# Service-Oriented Architectures for Embedded Systems Using Devices Profile for Web Services

Elmar Zeeb, Andreas Bobek, Hendrik Bohn and Frank Golatowski
*Institute of Applied Microelectronics and Computer Engineering, University of Rostock*
*(elmar.zeeb, andreas.bobek, hendrik.bohn, frank.golatowski)@uni-rostock.de*

## Abstract

*In recent years a movement from distributed systems controlled by users to automatic, autonomous and self-configuring distributed systems is noticeable. Web services is one approach but lacking the secure integration of resource-constraint devices.*

*This paper describes the Devices Profile for Web Services (DPWS), underlying protocols and a DPWS toolkit implementation based on C and gSOAP and discuss its current state. It has enormous relevance for embedded systems and industrial automation since DPWS targets resource-constraint devices explicitly, and has the potential to shift the industrial landscape which is characterized of heterogeneous devices.*

## 1. Introduction

In recent years a movement from distributed systems controlled by users to automatic, autonomous and self-configuring distributed systems is noticeable. Standardized self-describing interfaces (called *services*) and advanced separation of interfaces and implementation enhance the abstraction of component-based development and thereby paving the way for non-technical software engineers to develop complex, process-oriented software systems. *Service oriented architecture* (SOA) [1] is such an open concept supporting plug-and-play capabilities of heterogeneous software and hardware components. The SOA approach addresses issues such as service addressing, announcement, (self) description and discovering services as well as registering in and looking up a central service repository (service registry). The probably most popular implementation of SOA – *Web services* – is gaining increasing market penetration. The Web Services Architecture [2] provides a set of modular protocol building blocks that can be composed in varying ways (called *profiles*) to create protocol sets specific to particular applications. Profiles define which protocols are used, how they are adapted and in which way they are used to achieve a certain aim. Thus, profiles are meaningful instruments for achieving interoperability between software implementations of different vendors.

The *Devices Profile for Web Services* (DPWS) was developed to enable secure Web service capabilities on resource-constraint devices [3]. It allows sending secure messages to and from Web services, dynamically discovering a Web service, describing a Web service, subscribing to, and receiving events from a Web service. DPWS can be used for machine to machine communication whereas a specific client uses a specific service hosted on a device.

DPWS is not the first SOA that targets device-to-device communication. Technologies such as Open Service Gateway Initiative (OSGi), Home Audio/Video Interoperability (HAVi), Java Intelligent Network Infrastructure (JINI) and Universal Plug and Play (UPnP) are similar approaches.

The OSGi specification defines a service platform that relies on Java [4]. An OSGi service is a simple Java interface but the semantics of a service are not clearly specified. HAVi offers plug-and-play as well as Quality-of-service QoS) capabilities and is targeted for the home domain [5]. JINI was developed by Sun Microsystems for spontaneous networking of services and resources based on the Java technology [6]. Services/devices carry the code (proxy) needed to use them. UPnP supports ad-hoc networking for devices and services and it is easy to develop for [7]. It has a very similar functionality in comparison to DPWS but does not address security issues and is only applicable for small networks (no service registry/proxy).

The big advantage of DPWS compared to all other mentioned SOAs is the reliance on Web service which implies high acceptance among developers and platform as well as programming language independence. Microsoft has included DPWS in their latest operating system called Windows Vista.

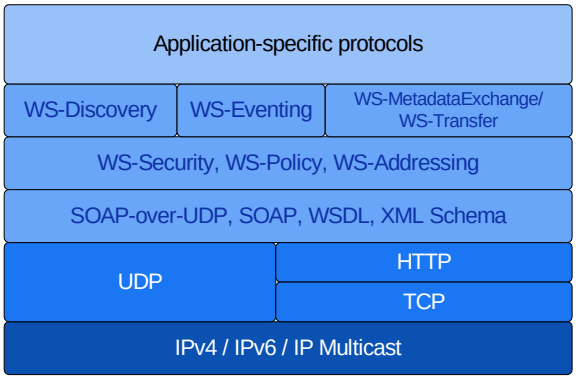


**Figure 1: Devices Profile for Web services as protocol stack**

This paper investigates the specification, implementation and use of DPWS for embedded systems and industrial automation. It was implemented and tested during the pan-European project SIRENA (Service Infrastructure for Real-time Embedded Networked Applications) [8] [9].

The paper is organized as follows. In the next section a short introduction to the underlying protocols used and adapted by DPWS are given. Section 3 deals with the DPWS specification. Adaptations and extension of the protocols used for DPWS are discussed. In section 4 our implementation of a toolkit for developing DPWS compliant services, devices and clients with the programming language C will be described. In section 5 our experience including the pitfalls of implementing DPWS will be presented. In section 6 the benefit of using DPWS for resource-constraint devices will be discussed. An overall conclusion and future work is summarized in section 7.

## 2. The Underlying Protocols of DPWS

DPWS is partially based on the Web Services Architecture (WSA) and uses further standards and draft specifications from the Web services protocol family (see Figure 1). SOAP (over HTTP), WSDL and XML-Schema are part of the WSA and will not be explained in this paper. The other standards and draft specifications are explained in the following sections.

### 2.1 WS-Policy

Since Web Services Architecture is platform, programming language and transport protocol independent, different service implementation have different capabilities, requirements and characteristics. WS-Policy provides a framework for expressing such declarations. If a Web service offers policies service users have to comply with the declarations found in the policy document. Service users have to choose one of the offered policy alternatives which consist of several policy assertions from the policy document.

Policies mostly define QoS characteristics and security considerations which are necessary for service communication.

### 2.2 WS-Addressing

The main objective of WS-Addressing which currently holds the state of a W3C submission is to provide an addressing mechanism for Web services as well as messages in a transport-neutral matter. By introducing both concepts *endpoint references* (EPR) and message information headers (MI) WS-Addressing overcomes the lack of SOAP's independence of underlying transport protocols (in most cases HTTP) and secondly support of asynchronous message exchange. Both limitations are historically caused by the default SOAP to HTTP binding.

Endpoint references are structures defined to address Web services endpoints. The address property within endpoint references is mandatory and its value could be an HTTP-based URL, a mail-to *Unified Resource Identifier* (URI) as well as a logical address expressed with an URI or any other *Internationalized Resource Identifier* (IRI). *Message information headers* are defined to address messages independent of the transport mechanism: Messages can be labeled with unique message IDs and as a result messages can be referenced by other messages, e.g. a request message can be referenced in a response message. These mechanisms release SOAP messages from the default synchronous message exchange pattern (see Figure 2) when using SOAP over HTTP. Further, message information headers are used to indicate sender and receiver endpoints as well as endpoints where to send responses and faults. The mandatory action property in the SOAP header tells the receiver

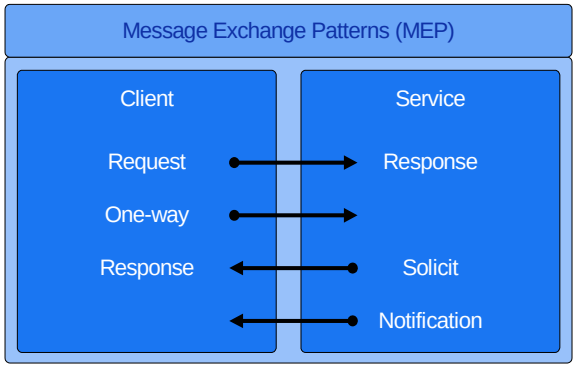


**Figure 2: Message Exchange Patterns**

about the meaning of the message hence its semantic.

The following Web services standards are based on WS-Addressing.

### 2.3 SOAP-over-UDP

As mentioned above, the trend of decoupling SOAP from its underlying layers leads to other binding protocols for SOAP. SOAP-over-UDP falls in this category and defines a binding for SOAP envelopes to UDP. In contrast to TCP the delivery of UDP packets can not be guaranteed.

SOAP-over-UDP uses WS-Addressing to support the same message exchange patterns as SOAP-over-HTTP. Unicast as well as multicast transmission is supported. Since UDP's unreliability message retransmission is encouraged, and for efficiency reasons a retransmission algorithm is provided which considers delayed repetitions of the same message.

### 2.4 MTOM

The idea of SOAP Message Transmission Optimization Mechanism (MTOM) is to provide an abstract feature for optimizing the transmission of SOAP messages. With MTOM parts of the envelope can be encoded outside of the envelope while still keeping the envelope intact. MTOM defines its feature in an abstract matter which means it hast to be adapted to the protocols used beneath.

The second part of this specification provides a binding for HTTP based on XML-binary Optimize Packaging (XOP) convention.

### 2.5 WS-Discovery

WS-Discovery is a discovery protocol based on IP multicast for enabling services to be discovered automatically. Discovery introduces three different endpoint types: target service, client and discovery proxy. *Target services* are Web services offering themselves to the network. *Clients* may search for target services and discover them dynamically. *Discovery proxy* is an endpoint enabling discovery in spanned networks since simple discovery is limited to a multicast group and hence to local managed networks only.

WS-Discovery defines four operations or messages to discover target services in a network. To explicitly discover target services in a network a client can use the *Probe* operation, send as multicast message. The message contains the requested target service type and scope. Matching target services will answer with the *Probe Matches* operation send as UDP unicast message to the client. To implicitly discover target services a client can listen for *Hello* and *Bye* messages. A target service announces its availability with these messages send as UDP multicast.

To resolve logical addresses introduced in WS-Discovery a client can use the *Resolve* operation send as UDP multicast message. The corresponding target service responds with the *Resolve Matches* operation send as UDP unicast to the client.

The discovery proxy does not need any additional operations.

### 2.6 WS-MetadataExchange / WS-Transfer

WS-MetadataExchange is a specification that defines data types and operations to retrieve metadata associated with an endpoint. This metadata describes what other endpoints need to know to interact with the described endpoint. WS-MetadataExchange defines the *MetadataSection* that divides the metadata into separate units of metadata with a dialect specifying its type. *MetadataSection*s can contain arbitrary XML-Data. There are predefined dialects for transferring WSDL-Documents, WS-Policy expressions, etc. Further user-defined *MetadataSection*s can be used to describe user-defined metadata.

Until the latest version of DPWS only WS-MetadataExchange was used for service and device description and retrieval. In the latest DPWS version of February 2006 [3] WS-Transfer is used to retrieve the metadata. Metadata is still structured as specified in WS-MetadataExchange. There is a slight functional difference in WS-MetadataExchange and WS-Transfer for retrieval of metadata. WS-MetadataExchange defines operations to retrieve all or parts of the metadata of an endpoint. WS-Transfer only can be used to retrieve all metadata of an endpoint.

WS-Transfer is very similar to HTTP since it defines *Create*, *Get*, *Put* and *Delete* operations which have almost the same semantics as HTTP request methods.

We expect WS-Transfer and WS-MetadataExchange to be merged closer in future releases.

### 2.7 WS-Eventing

WS-Eventing defines a protocol for managing subscriptions for a Web services based eventing mechanism. This protocol defines three endpoints: subscriber, event source and subscription manager. *Subscribers* request subscriptions on behalf of event

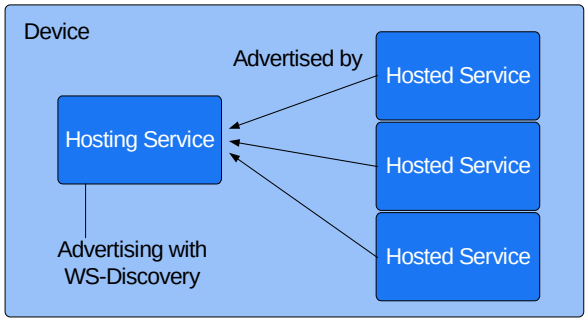


**Figure 3: Service hosting on a device**

sinks to receive events from *event sources*. Subscription requests contain an event *delivery mode* and *event filter* mechanism to negotiate an event source with an event sink. *Subscription managers* are responsible of holding subscriptions of event sources. Subscriptions must be requested and can expire over time. The subscription manager can be asked to renew or end a subscription, or to get the status of a subscription.

There are no limitations or restrictions in supporting other or user-defined event delivery and filter mechanisms.

### 2.8 WS-Security

This specification provides mechanisms for message integrity and confidentiality to SOAP messaging. These mechanisms are independent of the used technology. The specification defines the structures security header and security token to allow signing full or parts of SOAP messages or encryption of SOAP messages.

While the WS specifications define abstract mechanisms WS profiles define specific constraints and limitations. So the Devices Profile for Web Services describes how some specific WS specifications can be used on embedded devices.

## 3. Constraints, Limitations and Extensions specified by DPWS

In this section the DPWS specific constraints, limitations and extensions for above mentioned Web service specifications are described. They are divided into five parts covering the topics messaging, discovery, description, eventing and security. The specification uses the notation for requirement levels defined in RFC2119. For compliance only the “MUST” or “REQUIRED” level must be satisfied. In general it is hard to figure out the requirements that must be met for basic compliance with the specification. The specification contains lots of requirements that are not needed for compliance. But some of them lead to interoperability issues. Such problems are highlighted in the following sections.

### 3.1 Messaging

The messaging layer of DPWS relies on the SOAP 1.2, SOAP-over-UDP, HTTP/1.1, WS-Addressing, RFC 4122 (UUID) and MTOM specifications. DPWS does not use the full functionality of the underlying WS specifications but restricts them. The aim is to define only needed functionality so that DPWS can be implemented on small devices and that message sizes keep small. A service on a device must support the HTTP chunked transfer-coding and may support MTOM to receive or transmit messages exceeding this limits. In general messages exceeding this limit can be sent but then a service on a device may fail to process or may reject this message. This may result in incompatibility issues.

For basic interoperability a service on a device must at least support receiving and sending SOAP 1.2 envelopes over HTTP. It must at least support the request-response message exchange pattern and respond to one-way message exchange patterns.

A device must support WS-Addressing by including a relationship field in the message information header of each response or fault message. It must also generate a failure on request messages received over HTTP not including an anonymous reply endpoint. For device addressing an UUID or another identifier must be used that is stable, globally unique across an IPv4/v6 network and persistent across reinitialization or changes on the device.

In summary DPWS restricts the underlying WS specifications to restrict their functionality. But in some parts of the messaging section the specification is unclear and may lead to interoperability issues.

### 3.2 Discovery

There is a risk of misunderstanding discovery in DPWS with services discovery in the Web services world, when reading the specification for the first time. DPWS uses several steps to negotiate a client with a service. The first step is to discover devices, which is done with WS-Discovery. The actual service discovery is shifted to the client with the help of the device description metadata described later.

For limiting bandwidth usage in a network only a device, represented as a special service, must advertise

| Needed actions | Static case - Client knows everything | Client knows service address | Client knows device address | Generic case - Client knows nothing |
|---|---|---|---|---|
| Device **discovery** | | | | X |
| Device **description** | | | X | X |
| Service **description** | | X | X | X |
| Service **usage** | X | X | X | X |

**Figure 4: Steps needed for service usage**

itself with WS-Discovery. The actual services on a device are called hosted services and should not advertise themselves with WS-Discovery (see Figure 3).

For basic interoperability a device must support sending and receiving discovery messages over UDP unicast and multicast. For static scenarios, where the HTTP address of a device is already known, a device must support receiving discovery messages over HTTP and respond at least with *HTTP 202 Accepted* which is comparable to a ping operation. DPWS defines the *wsdp:Device* target service type that signals conformance with DPWS and should be included in discovery messages if types are included.

To simplify a resource-constraint device implementation a device must at least support the *rfc2396* and *strcmp0* scope matching rules.

To sum the discovery section up, it is important to not consider discovery in DPWS as service discovery. Furthermore, the semantic of device types is not clearly defined.

### 3.3 Description

DPWS relies on XML Schema, WSDL, Basic Profile Version 1.1, WS-MetadataExchange, WS-Policy, WS-PolicyAttachment and WS-Transfer to describe a device's or a service's metadata. The DPWS specification further divides the description metadata into characteristics, hosting, WSDL and policy description. All description of a (hosted) service or device (hosting service) can be retrieved with a *WS-Transfer Get* operation.

Figure 4 illustrates the several degrees how generic or static a client implementation may be. For highly-constraint circumstances, called static case, the DPWS specification assumes that a client knows all it needs to use hosted services on a device, so no discovery or retrieval of description is needed. For more generic usage, after a client has discovered a device, it must retrieve a devices description from its hosting service. This is the only reliable way for service discovery as the hosting description contains a device's *Relationship MetadataSection*. In the *Relationship* data structure defined by DPWS each hosted service is included with its address, ID and supported types as defined in WS-Discovery.

After retrieving the device description a client has to get the service description from a service. The service description contains the WSDL documents of a service and the policy description. Now the client knows all about the service to use it.

A client may retrieve a devices characteristics description to further inspect a device. This description is contained in a *ThisModel MetadataSection* and a *ThisDevice MetadataSection*, defined by DPWS.

To conclude description in DPWS defines the missing steps for service discovery. But the steps needed for service discovery depend on how static or dynamic a scenario is. In static scenarios most steps can be omitted.

### 3.4 Eventing

The devices profile includes an event mechanism based on WS-Eventing. A service, which provides events, must therefore have the attribute *EventSource=true* in its WSDL and be able to process subscribe request messages.

DPWS defines one event delivery mode and one event filter mechanism that has to be supported at least, to limit the functionality needed for resource-constraint device implementation. The event delivery mode is called *push delivery* mode and is already defined in WS-Eventing. This delivery mode must be supported at least for conformance to DPWS. The event filter is called *action filter* and lets the subscriber specify which events should be received.

An event source only must support subscriptions with duration type *xs:duration* but not *xs:date*, as synchronized clocks will not meet requirements of resource-constraint devices.

As a conclusion it is important to know that the only reliable way to detect if a service provides events is the *EventSource* attribute in its WSDL.

## 4. Implementation of DPWS

In this section the implementation of a toolkit, called WS4D-gSOAP [12], for developing devices and

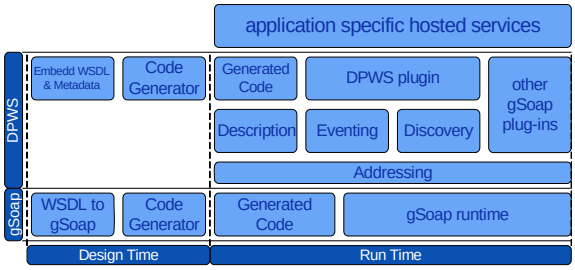


**Figure 5: WS4D-gSOAP toolkit**

clients conforming to the Devices Profile for Web Services will be introduced. WS4D-gSOAP is based on another toolkit to develop Web services in the programming language C/C++ called gSOAP. WS4D-gSOAP uses gSOAP's plug-in mechanism and implements several specifications like WS-Addressing, WS-Discovery, Description (WS-Transfer/WS-MetadataExchange) and WS-Eventing as plug-in independent of DPWS (see Figure 5). In the following subsection gSOAP, and the implementation of the devices profile will be described.

### 4.1 gSOAP

gSOAP [11] is a toolkit for building SOAP-based Web services in C/C++ developed by Robert A. van Engelen. gSOAP is designed to develop low footprint and high throughput Web services. gSOAP consists of a development and a runtime environment. gSOAP has defined its own service description language that is similar to the C syntax. This description is saved in special gSOAP files that are similar to C header files. The toolkit includes a tool to translate WSDL files into gSOAP files. The second part of the development environment is the gSOAP code generator. This generator generates XML schema to C data binding code and stub and skeleton code for a specific gSOAP service description. The XML schema to C data binding maps every type of the used schemas to a C structure and generates functions for marshalling and demarshalling. The skeleton and stub code generator maps WSDL operations to C functions.

gSOAP was chosen as basis for the implementation of the devices profile because it has an open source license, it meets the requirements for embedded systems, supports the Web services basic profile, has a plug-in concept and already includes a lot of plug-ins that implement Web service standards and other features.

### 4.2 WS4D-gSOAP

The WS4D-gSOAP Toolkit implements the constraints, limitations and extensions defined in DPWS based on gSOAP. It uses gSOAP's plug-in mechanism to implement WS-Adressing, WS-Discovery, WS-MetadataExchange / WS-Transfer and WS-Eventing.

It supports the three roles device, client and peer that can be switched at compile time. With the device role an endpoint implements the device side of the specification. To implement a client the client role must be enabled at compile time. In case of a peer endpoint, that means support for device and client role, the peer role must be used. WS4D-gSOAP is similar to gSOAP and thus uses a similar workflow.

This is only a brief description of the WS4D-gSOAP toolkit as a complete description is out of scope of this paper. Another toolkit which has been developed within SIRENA project is described by [10].

## 5. Pitfalls of implementing DPWS

When implementing DPWS the first pitfall is to figure out the basic functionality needed for compliance. As DPWS is designed for resource-constraint devices as well as for functionality-richer implementations the specification contains a lot of optional requirements and only a few mandatory requirements, respectively. Since most of these requirements are related to devices, this leads to a well-defined device specification but a vague client specification. Therefore, device implementation is straight forward and client implementation leaves a lot of open questions.

When implementing DPWS based on a Web services toolkit the choice which toolkit to use should be made carefully. A toolkit should support at least transmitting SOAP-over-UDP messages in form of unicast as well as multicast and support one-way, request-response and solicit-response message exchange patterns. Further the underlying IP stack must support IP multicast.

Before using a service on a device a client has to be aware of the service's description and endpoint. Assuming a high dynamical environment a client needs a maximum of three steps for accessing this description: (1) device discovery by device type and scope, (2) device description transfer, and (3) service description transfer. In case the client knows all it

needs in advance to communicate with a service (static case) these steps can be ignored. In all other cases only the needed steps have to be done. So discovery is only used for device discovery. To determine the capabilities and the endpoint of a requested service the description must be acquired with WS-Transfer. For example a service providing events has to include *wse:EventSource=true* as attribute in the portType definition of its WSDL.

The approach for service discovery that is at the bottom of DPWS can be best seen in the WSD-API from Microsoft [13]. The WSD-API is the programming interface to use Microsoft's client endpoint implementation integrated into next versions of their operating system. In this implementation all device and service description metadata is cached on the client side and discovery is used to keep this metadata up to date. The actual service discovery that is called function discovery is based on this cache. This approach shows that DPWS assumes resource-constraint devices and resource-rich clients.

To classify devices and services, DPWS uses a type system. Until now the semantic of this type system is not defined. It is important to not mistake these types with WSDL port types! DPWS only defines the device type *dpws:Device* that indicates compliance to the profile and support for retrieving descriptions.

Device discovery as defined in DPWS will cause interoperability problems, which may lead to clients that will not discover all requested devices. Length restrictions for message fields or whole messages (R0025, R0027, R0029, R0003, and R0026) defined in the messaging section may lead to interoperability issues. The client side could consider the restrictions when sending messages and the device side could reject messages that exceed the restrictions.

There are also technical pitfalls when implementing discovery for DPWS. WS-Discovery uses a special message exchange pattern that is comparable to the request-response message exchange pattern. The request is sent by the client by UDP multicast and the response is sent by the device by UDP unicast. The destination of the request is defined by WS-Discovery. The destination of the response is corresponding to the source address of the request. This means a discovery client, sending a request over UDP multicast to a device must listen on the same UDP port, the request was send for receiving the unicast response.

Another technical issue for a device is to obtain its IP address. A device can announce its real address with discovery or must be able to resolve its logical to its real address. Implementations relying on DNS to obtain its IP address will only work in a network where DNS is setup correctly.

## 6. Relevance for Embedded Systems and Industrial Automation

The devices profile has enormous relevance for embedded systems and industrial automation since DPWS targets resource-constraint devices explicitly, and has the potential to shift the industrial landscape which is characterized of heterogeneous devices.

The goal of the research project SIRENA was to develop a framework for embedded networked applications overcoming the obstacles of platform or language dependent architectures. SIRENA framework based on DPWS and was proven in and between four different domains: automotive, telecommunication, industrial and home automation.

One of the main problems identified while evaluating the profile was the client-side implementation. We found two aspects which have to be clarified in the future: The profile has too vague constraints and device types are necessary for efficient discovery.

The rules within the devices profile can be classified in mandatory and optional. Few mandatory constraints result in thin devices. This is a requirement of the specification. While optional rules can be left out by device implementations clients have to consider these cases if they are intended for use with unknown devices in dynamical environments. DPWS defines a lot of such optional rules for devices and this design issue lead to "fat clients".

WS-Discovery allows parameter-based searching for devices by specifying types and/or scopes. As explained in section 5 neither WS-Discovery nor DPWS introduce what is meant by types and how they are defined. However, this is a crucial point in order to exploit DPWS fully. We demand a mechanism similar to UPnP device templates or WSDL port types which allows definition of specific device types, for example "motor" or "audio renderer". This mechanism also should support some type of grammar, e.g. a specific XML schema which binds a QName, as required for discovery types, to WSDL port types. Devices marked with such type are known to support all port types bound in the device type definition.

As there is a transition from WS-MetadataExchange to WS-Transfer to retrieve a devices description, this part

of DPWS will certainly change in future versions of the specification

## 7. Conclusion and future work

In this paper we presented an overview of the Devices Profile for Web Services, its underlying Web services technology and its pitfalls when implementing it. Furthermore we introduced our gSOAP-based toolkit.

As we have shown in this paper the devices profile will need some slight changes to become a standard which allows implementing interoperable Web services on resource-constraint devices as well as clients. At the moment a devices description is only available at run time. A schema and/or template system for defining device types would improve the support of code generators towards easy service creation and allow a finer grained device discovery. We will further investigate how device templates can improve DPWS and complete the toolkit. For implementing Web services on devices with even lower resource constraints and for solving problems with too big message sizes XML-binary technologies like Fast Web Services will be investigated in future research.

## Acknowledgments

This work has been funded by German Federal Ministry of Education and Research under reference number 01|SF11H.

## References

[1] W. Dostal, M. Jeckle, I. Melzer, B. Zengler, *Service-orienterte Architekturen mit Web Services*, Elsevier, München, 2005.
[2] W3C, Web Services Architecture, 2004.
[3] 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, Y. Yarmosh, Devices Profile for Web Services, Microsoft Developers Network Library, February, 2006, http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf.
[4] OSGi Alliance, OSGi Service Platform Release 4 CORE, 2005.
[5] J. Teirikangas, HAVi: Home Audio Video Interoperability. Technical report, Helsinki University of Technology, 2001.
[6] Sun Microsystems, Jini Architecture Specification Version 1.2, 2001.
[7] UPnP Forum, UPnP Device Architecture v.1.0.1, 2003.
[8] H. Bohn, A. Bobek, F. Golatowski, "SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A service-oriented framework for different domains", in Proceedings of 5th International Conference on Networking ICN'06, 2006.
[9] F. Jammes, H. Smit, "Service-oriented architectures for devices – the SIRENA view", in Proceedings of 3rd IEEE International Conference on Industrial Informatics INDIN '05, 2005
[10] F. Jammes, A. Mensch, H. Smit, "Service-oriented device communications using the devices profile for web services", in Proceedings of 3rd international workshop on Middleware for pervasive and ad-hoc computing MPAC '05, 2005
[11] http://www.cs.fsu.edu/~engelen/soap.html
[12] http://www.ws4d.org
[13] http://windowssdk.msdn.microsoft.com/en-us/library/aa385800.aspx