Ref. Ares(2011)1133498 - 24/10/2011

Large Scale Choreographies for the Future Internet

ICT IP Project

Companion to Deliverable D3.1

Survey on DPWS

http://www.choreos.eu

Project Number	:	FP7-257178
Project Title	:	CHOReOS Large Scale Choreographies for the Future Internet

Deliverable Number	:	Companion to D3.1
Title of Deliverable	:	Survey on DPWS
Nature of Deliverable	:	Report
Dissemination level	:	Public
Licence	:	Creative Commons Attribution 3.0 License
Version	:	1.0
Contractual Delivery Date	:	30 September 2011
Contributing WP	:	WP3
Editor(s)	:	Sandrine Beauche (INRIA)
Author(s)	:	Sandrine Beauche (INRIA)

Abstract

Device Profile for Web Service (DPWS) is a specification to enable Web Services on resource constrained devices. In particular, tackling the mobility of these devices, this specification allows discovering services dynamically and accessing them. So, DPWS seems to be successor to UPnP, relying on a subset of the Web Services standards and SOAP messages over UDP in order to deal with resource constraints. The founders of this specification are Microsoft as well as the actors of the SIRENA European ITEA project, who aimed at leveraging SOA architectures to seamlessly interconnect embedded devices in the domain of industrial applications, telecommunications and automation. This project was a first promising contribution. As a follow-up to this, these actors are working on (1) extensions to add more features, like reliability and security, (2) new implementations to deal with a higher diversity of devices, from tiny devices to servers, and (3) tools to facilitate incorporating DPWS in other new technologies, and to improve device management in Windows or e-Management in factories. In this document, we survey the DPWS specifications, the existing implementations, and the related projects that apply or improve DPWS.

Keyword list

DPWS, Web Services, Devices, Implementation, Discovery.

Document History

Version	Changes	Author(s)
1.0	Creation	Sandrine Beauche

Glossary, acronyms & abbreviations

Item	Description
CA	Consortium Agreement
DL	Deliverable Leader
DOW	Description of Work
IAC	Industrial Advisory Committee
MST	Management Support Team
OSS	Open Source Software
PL	Project Leader
PMC	Project Management Committee
PO	Project Officer
PTC	Project Technical Committee
SL	Scientific Leader
WP	Work Package
WPL	Work Package Leader

Table of Contents

1. Introduction

The aims of this document are the presentation and the overview of Device Profile for Web Service (DPWS), a specification to enable Web Services on resource constrained devices. It aims also to explain how to use DPWS and how it can be involved in projects.

In section 2, we present the context and issues addressed by DPWS and a state of the art of the solutions provided to solve them.

In section 3, we give an overview of the DPWS specification.

In section 4, we provide a panorama of the existing implementations to build DPWS servers and clients. For each of them we will explain who to use them, their advantages and disadvantages. We also compare them according to their DPWS feature, development facilities and other characteristics that are important when choosing an implementation for a project.

In section 5, we show projects that are based on the DPWS technology, and we conclude in section 6.

2. Context and State of the Art

The last years, we have witnessed two major trends in the world of embedded devices. First, hardware is becoming smaller, cheaper, and more powerful. Many everyday objects already incorporate embedded microcontrollers and will increasingly include wireless interfaces. Typical microcontrollers incorporate a microcomputer, storage, software and interface sensor. The combination of these elements and the greater Internet makes it possible for the internet to evolve from a network of interconnected computers to a network of interconnected objects, that will have communication and computation capabilities, which they will use to connect, interact and cooperate with their surrounding environment. In addition, people and machines will monitor and control such objects from a distance, via the Internet. This new Internet is called Internet of things [1].

Second, the software industry is moving towards service-oriented integration technologies. Especially in the business software domain, complex applications based on the composition and collaboration among diverse services have been appearing. The most popular implementation of SOA architecture is the Web Services.

Devices may be abstracted by web services to enable their interoperability in the Internet of Things context and create real-world services. In the future Internet, real-world devices will be able to offer their functionality via SOAP-based web Services, enabling other components to interact with them dynamically. From a technology point of view, the challenge is how to discover, efficiently integrate the real world services into business application, and deal with the changing nature of the environment and the mobility of the devices.

In practice, devices may appear and disappear, discover the capabilities offered by the other devices in the subnet and invoke them. Let's imagine, as illustrated at the figure 1, a user that enter to its house with its PDA and want to print a document. He may discover the the printer that provides the needed capability and send the document to it. In another hand, the interaction with devices to control environment needs eventing: for example a mp3 may produce an event when a sing is finished.

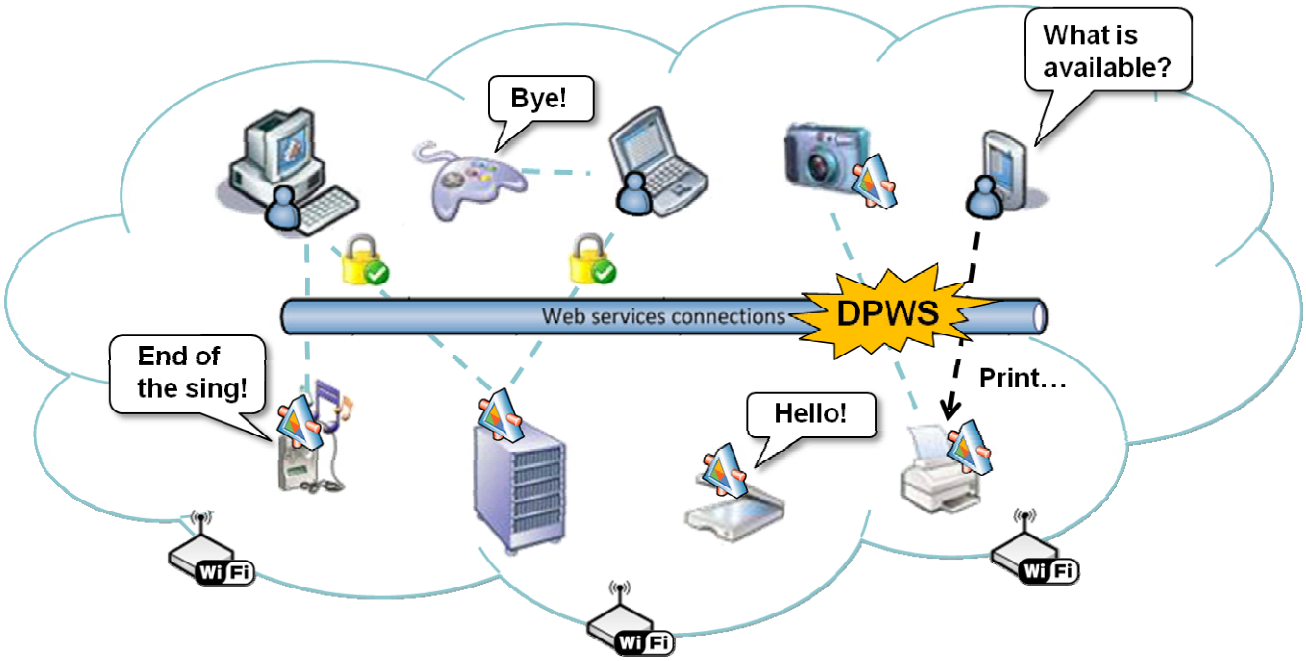

Figure 1: The internet of Things

Internet of things use cases may be from the simple home network to the ultra-large scale internet, involving multiple local network, large number of devices with an high heterogeneity, possibly several network media as Ethernet, Wifi,... The Choreos project proposes for example the DynaRoute use case, an ultra-large scale internet scenario. The DynaRoute use case depicts a realisitic situation of a person following a predefined itinerary on her way from the hotel to the airport. During this itinerary, her smartphone reserves dynamically a taxi and adapt its way according to unpredicted events, get notifications from the airport about a delay for her flight, from stores about interesting sales or from a monument for site-seeing.

To enable the internet of things, particularly in the scope of large scale internet scenarios, SOA faces the following challenges:

- Providing a **device support** to abstract devices and their drivers, make them accessible and controllable from the Internet.
- This device support should be **plug & play**, in order to discover dynamically new devices and their capabilities without knowledge about them. During this phase, the systems discovers dynamically how to interact with the new device.
- To deal with a high heterogeneity of the devices, the architecture should be able to integrate device software layer written in different programming language. It is especially true that some device manufacturers designed their product (as the Tahoe development board) for the Microsoft development suite with C#, while other products are designed for Java like the SunSpot sensor or the android smartphones. Thus, the architecture should be **programming language independant**
- In one hand the heterogeneity of the devices involves an heterogeneity of the used network media since some devices are accessible on the network only by wifi, while other machines are connected by Ethernet for example. Thus the architecture should be **network media independant** to avoid theses integration issues.
- In the context of large scale scenario, the architecture should be able to deal with **large area network** like internet, and should be able to integrate several local network to enable cross domain and enterprises cooperation.
- The second aspect of a **large scalability** is the capability to manage thousands of nodes and services to be coordinated and with a huge load.
- **Security** is an important concept in large area network to restrict the access to resources (i.e devices and services) to users, ensure privacy and secure communications between servers and clients. The architecture should support methods of authentication and secure channels.
- In order to integrate large number of cross domain devices and services, this architecture should be applicable in a lot of different domain and have a **high acceptance market**.

Numerous technologies are developed to enable dynamicity, flexibility, and integration of software and devices, in the context of the internet of things:

OSGi [4]

The OSGi technology is a dynamic module system for Java and provides a component-oriented, lightweight container framework to host dynamically managed services. This

platform provides the standardized primitives that allow applications to be constructed from small, reusable and collaborative components. It solves the dynamic loading, versioning and lifecycle management issues for Java-based services and also provides services to develop an ecosystem around it. In this way, services can be dynamically loaded and unloaded.

The OSGi alliance improved continuously its specification; starting in 1999 by targeting at the embedded java end networked device markets in home and telematics environments, including product series from Bosh, Siemens Home Appliance and BMW. Then in 2003, the alliance began to expanding its specification to accommodate the mobile device market, and was adopted by the open source communities as in the Apache Felix project, Knopflerfish and more. Today enterprise industry leaders are increasingly adopting OSGi technology. Diverse and new markets continue to implement and deploy products and solutions around the world: enterprise, open source, mobile, telematics, smarthome, e-health, vehicles [6]

OSGi can manage devices thanks to a device manager and by integrating device drivers as the other services [7]. OSGi provides plug & play mechanisms to manage them dynamically, as illustrated in the figure 2: the hardware is registered with its base driver to the device access bundle. This base driver is a generic driver with a low-level interface to enable the system to communicate with the device. Then the drivers that are more appropriate to communicate with the device are located thanks a driver database, installed, selected and attached to the device. For example, in the scenario with a USB printer, the USB driver registers the printer as an USB device and enables to communicate with it by the USB bus. Then the appropriate printer driver is selected and loaded by the device manager. However the plug and play perspective in the OSGi platform is restrained with the a priori knowledge of each device that may be connected.

The OSGi architecture provides a security layer based on the Java 2 specification. It enables to add permissions for the execution of the java code: access modifiers in the language restrict the visibility of the code to other programmers.

Although the OSGi components are originally designed to work on the same JVM, some implementation allows a distributed OSGi as the Apache Felix [8] and Apache CXF [9] implementations. But the devices that are discovered are not networked, so a OSGi node must be deployed at each point where we can connect a device. In the other hand, CXF implements the Remote Service functionality using Web Services, leveraging SOAP over HTTP and exposing the service over a WSDL contract. Java interfaces can also be exposed and consumed as RESTful JAX-RS services. In addition, the OSGi plateform is able to integrate external resources as web services, HTTP resources into a bundle in the OSGi architecture. But the Web Service cannot be discovered dynamically: each Web Service is explicitly exposed as a bundle.

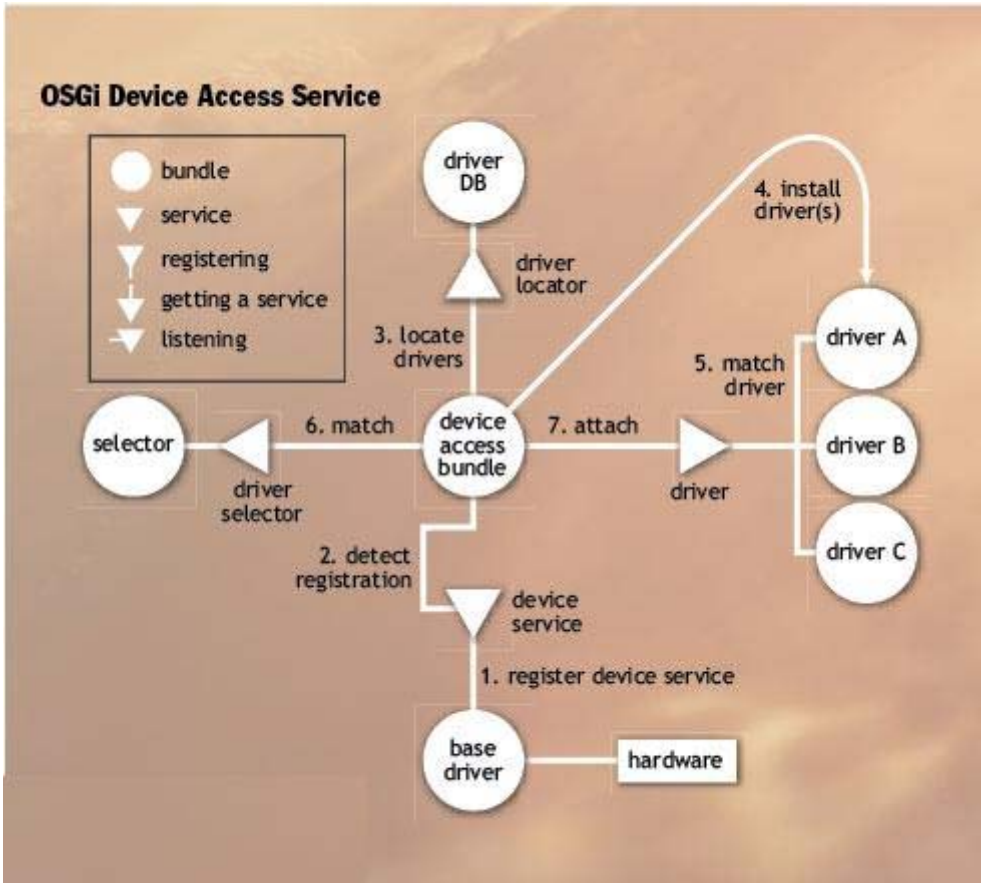

Figure 2: OSGi Device integration

HAVi [8]

HAVi is an open standard established by major electronics for home entertainment manufacturers as Grundig, Thomson, Toshiba, etc. It aims to bring true platform independent interoperability to consumer devices.

HAVi provides automatic discovery with adding or removing devices from the network, lookup service for discovering resources, automatic installation and configuration. It supports posting and receiving events and remote control of the devices.

This architecture is much closed to the home entertainment domain, since it is restrained to home entertainment devices that communicate with IEEE 1394 wiring as connecting medium. Consequently, networks managed with this infrastructure are often restrained to a home network, which does not represent a large scale network.

HAVi specifies a basic permission mechanism on the type of the messages: some devices have permissions to send only some types of messages.

JINI [9]

Jini is a system that aims to turn the network into a flexible, easily administrative tool with which resources can be found by human and computational clients. Resources can be implemented as either hardware devices, software programs, or a combination of the two.

It provides though an SOA architecture based on the Java RMI technology, mechanisms for devices, services and users to join and detach from a network. The key element of this architecture is the lookup service; service providers register themselves to this service and client invoke it to discover dynamically the available service. Then the client can invoke the discovered services by loading its proxy class code as we should do with a classic RMI invocation.

The Jini system extends the Java application environment from a single virtual machine to a network of machine. This architecture leads still with local network, as virtual machines are connected explicitly before deploying the distributed application. In addition, the centralized registry is not well adapted to big network but for local network.

The design of the security model for Jini is built on top of the twin notion of a principal and an access control list. Jini services are accessed on behalf of some entity (e.g. the principal) which generally traces back to a particular user of the system. Services themselves may request access to other services based on the identity of the object that implements the service. Whether access to a service is allowed depends on the contents of an access control list that is associated with this object.

UPnP [13][14]

UPnP, which bases on TCP/IP, is a widely accepted, powerful and yet simple approach to connect and connect and control internet gateways and multimedia devices in home. UPnP defines an architecture for pervasive peer-to-peer network connectivity of intelligent appliances, wireless devices and PCs in the home, in a small business, public spaces.

It is designed to support automatic discovery for a wide range of vendors. A device can dynamically join a network, obtain a IP address, convey its capabilities and learn about the presence and capabilities of the other devices. Finally, a device can leave a network smoothly and automatically without leaving any unwanted state behind.

UPnP specifies just the exchanged messages needed to perform the discovery and other dynamic functionalities. Thus this specification does not depend on the languages used to generate these messages. Moreover since this specification is built on top of TCP, it does not depend on the media network.

The main drawbacks of this approach is the design only for local networks: only devices o the same subnet are accessible and the UPnP protocol generate a lot of IP multicast messages while IP multicast does not scale very well on bi networks.

WS [15]

Web Services is the most popular implementation of the SOA architecture. Web Services has a lot of implementations from open source as Apache Axis [16] or Apache CXF [17], to industrials products as .NET [18], IBM Websphere [19].

It enables services accessibility from internet on top of HTTP [20]. So they address networks of any size and any type.

The Web Services architecture provides, on top of HTTP SOAP [21] (Simple Object Access Protocol) messages, a set of modular protocol building blocks that can be composed in varying ways to create protocols specific to particular application.

The most important building block is WSDL [22], a protocol that allows to specify the interface (e.g. the provided operations and their signatures) of a service. Thus a Web Service can be seen from 2 views: the abstract view where the Web Service exposes

how to interact with its capabilities, and the implementation that is hidden to the client. Web Service solves heterogeneity issues between services, like the different programming language.

There exists a building devoted the security and that allows authentication and secure channels.

The main drawback of the Web Services architecture is the lack of support to devices.

DPWS [23][24]

Device Profile for Web Services (DPWS) is a new specification attending to be a successor of UPnP by using the existing standards in the Web Services architecture in order to inherit their advantages in terms of features and advantages.
DPWS was developed to enable secure web service capabilities on resources-constrained devices, (i.e. everyday objects, sensors,...).

The SIRENA [25][26] project, which started in 2003 to leverage SOA architectures to seamlessly interconnected embedded devices and between distinct domains (the industrial, telecommunication, automotive and home automation domains), was a precursor for a first proposal of DPWS in 2004 [23] and its oasis standardisation in 2009 [24]. It was required to support plug-and-play connectivity, to base open standards, to apply down to sensor and actuator level and to be technology neutral regarding programming languages, operating systems and network media.

When the SIRENA project ended in the beginning of 2006, 2 initiatives were created by the partners of SIRENA to follow up the development of DPWS toolkits from the preceding results:

- The Web Services for Device (WS4D) [27] initiative, founded by the university of Rostock, the university of Dortmund and Materna Information and Communications [29]. This initiative allows to maintain the collaboration between these partners and building up on an open community around DPWS.
- The SOA4D [28] initiative, founded by Schneider Electric.

The 2 initiatives were established by academic and industrial partners and can be considered as non-profit follow-up projects to preserve and extend the SIRENA results. They aims to participate in further development processes to

- Improve available stacks by providing them as open source software
- Developing an ecosystem for the development of service-oriented software components adapted to the specific constraints of embedded devices. The elements of this ecosystem are test suites, DPWS browser and builder.
- Promoting standardization process, e.g delivery of standardized devices and basic services such as management services.

The following table summarizes the comparison of the different approach to build an SOA architecture and integrate devices into it.

	OSGi	HAVi	JINI	UPnP	WS	DPWS
Plug and Play	-	X	X	X	-	X
Device support	X	X	X	X	-	X
Programming langage independent	-	X	-	X	X	X
Network media independent	-	-	X	X	X	X
Large scalability	X	-	-	-	X	X
Security	X	X	X	-	X	X
High market acceptance	X	-	X	X	X	X

Figure 3: DPWS comparison

3. DPWS Overview

DPWS is a specification that enables a SOA architecture, in particularly Web Service as the most popular implementation, on resources constrained devices with all the features provided by Plug and Play and it aims to be large scalable. Devices and their provided capabilities are both considered as Web services with different roles: devices are services that encapsulate a real device and hosts other services that represent their capabilities.

DPWS is organized around 3 axes:

- The description of the devices and their provided services in order to exchange metadata about them with other devices.
- The alignment of a specification with Web Services standards for dynamic discovery of devices and services and the support for eventing and security. So DPWS is designed as a subset of Web Services standards.
- Providing rules to specifiy a minimal set of implementation constraints to deal with resources-constrained devices.

In the following subsection we will present the DPWS description format (section 3.1), then the subset of Web Services standard that constitute DPWS and their utilization (section 3.2) and an overview of the existing rules in DPWS (section 3.3). We finally compare the different versions of DPWS (section 3.4).

3.1. Description format

The first axe of DPWS is the description of the devices and their provided services. For this purpose, DPWS provide a XML schema to enable the specification of the devices and their services as a XML dialect.

Devices are described as a product model where all its copy built by the manufacturer share the same properties such as the manufacturer name and URL, the model name, number and URL or the presentation URL. The following code is the description of the TOSHIBA e-Studio model printer with DPWS.

```
<ThisModel >
   <Manufacturer>TOSHIBA TEC</Manufacturer>
   <ManufacturerUrl>http://www.toshibatec.co.jp</ManufacturerUrl>
   <ModelName> TOSHIBA e-STUDIO Series</ModelName>
   <ModelNumber>STUDIO351c</ModelNumber>
   <ModelUrl>http://www.toshibatec.co.jp/e-STUDIO</ModelUrl>
</ThisModel>
```

Figure 4: DPWS model device description

Devices are also described with properties that are unique to this copy, such as a friendly name, the version of the installed firmware and the serial number. The following code is a example of printer of the preceding described model.

```
<ThisDevice >
   <FriendlyName xml:lang="EN">TOSHIBA e-STUDIO</FriendlyName>
   <FirmwareVersion>T410Y0350</FirmwareVersion>
   <SerialNumber>04647420</SerialNumber>
</ThisDevice>
```

Figure 5: DPWS device description

The second part of a DPWS device description is its relations to its hosted services. This part describes into a Relationship element, a host relation between a mandatory host (e.g. the device) and one or several hosted services. These descriptions express only the minimum to discover efficiently devices and hosted services: an endpoint reference that enables access to the service, a type to classify services and an Identifier. As Devices are also considered as services and can be discovered, they also have an endpoint reference and a type. The following code is an example for a printer device that provide printer and scanner services.

```
<Relationship Type= "http://docs.oasis-open.org/ws-
dd/ns/dpws/2009/01/host" >
    <Host>
        <wsa:EndpointReference>…</EndpointReference>
    </Host>
    <Hosted>
        <wsa:EndpointReference>…</EndpointReference>
        <Types>PrinterServiceType</Types>
        <ServiceId>…/Printer</ServiceId>
    </Hosted>
    <Hosted>
        <wsa:EndpointReference>…</EndpointReference>
        <Types>ScannerServiceType</Types>
        <ServiceId>…/Scanner</ServiceId>
    </Hosted>
</Relationship>
```

Figure 6: DPWS device and hosted services relations description

Although the elements and properties exposed these examples are exhaustive, the description of the devices and their relations to hosted services can be extended with new elements and attributes that are not specified in the DPWS XML Schema. Effectively, the DPWS XML Schema contains <any> and <anyAttributes> elements that enable to add new elements in an instance.

3.2. Subset of Web Services standards

The force of the Web Service specification is the combination of building specification block for each aspect of the Services. DPWS is built on top of Web Services standards in order to inherit their advantages in term of technology neutrality, scalability, security and high acceptance market.

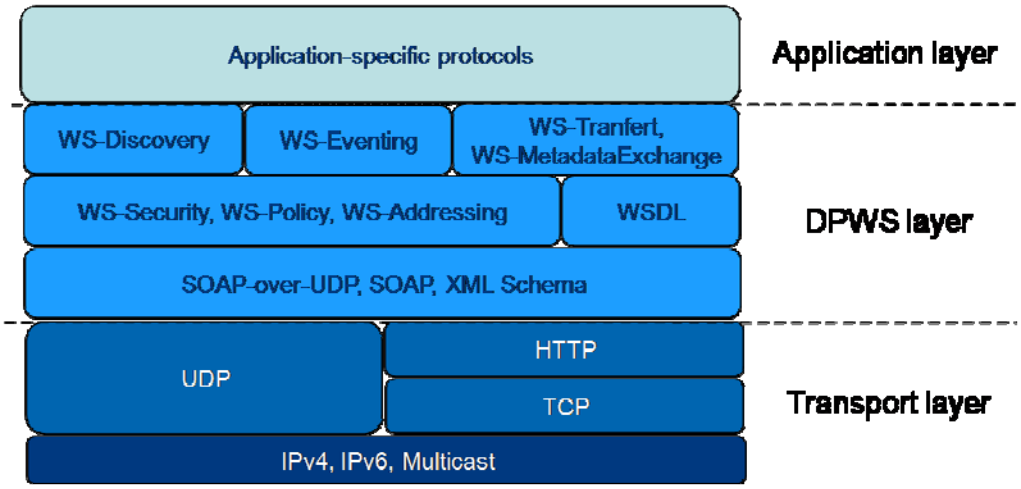

Figure 7: The DPWS stack

As illustrated in the figures 3, DPWS is built on top of UDP and TCP/HTTP transport protocols which are commonly used in web applications and discovery. In the DPWS stack, we can see that the set of constituting Web Services standards are organized in “3 main layers” (not official in the DPWS specification):

- A **message layer** intending to encapsulate and structure the messages, thanks to the SOAP and SOAP-over-UDP protocols
- A **properties layer** intending to provides description elements for services or to add complementary information about how to route messages. WS-Security enables security mechanisms on the SOAP messages such as encryption, digital signature. WS-Policy can inform the policies of the client and the server, for example the used version of DPWS. WS-Addressing can inquire information about the routing and the dispatching of the message.
- A **DPWS features layer** which provides the main high level features of DPWS, namely the discovery for devices and services detection, the exchange of metadata in order to invoke these services and the eventing. For each phase of a DPWS conversation, the DPWS rely on the 2 previous layers.

The DPWS conversation pattern is composed of the conversation patterns of DPWS features layer protocols and we can split this conversation in 3 phases:

- The **discovery** where devices notify their arrival (Hello messages) and departure (Bye messages) from the subnet and client search the available device and their address transport (Probe and Resolve messages). In this phase clients get the minimal information to identify a device.
- The **metadata exchange** thanks to the WS-Transfer and WS-MetadataExchange protocols where clients ask to devices (DPWS) and hosted services (WSDL) the necessary information to invoke services.
- The **service consuming** where clients can invoke a hosted service as a classical Web Service invocation, or receive notification from them thanks to the WS-Eventing protocol, which enables to manage subscription about events.

The figure 4 shows a pattern overview of a DPWS conversation. In the next subsections we will detail each protocol by describing their conversation pattern.

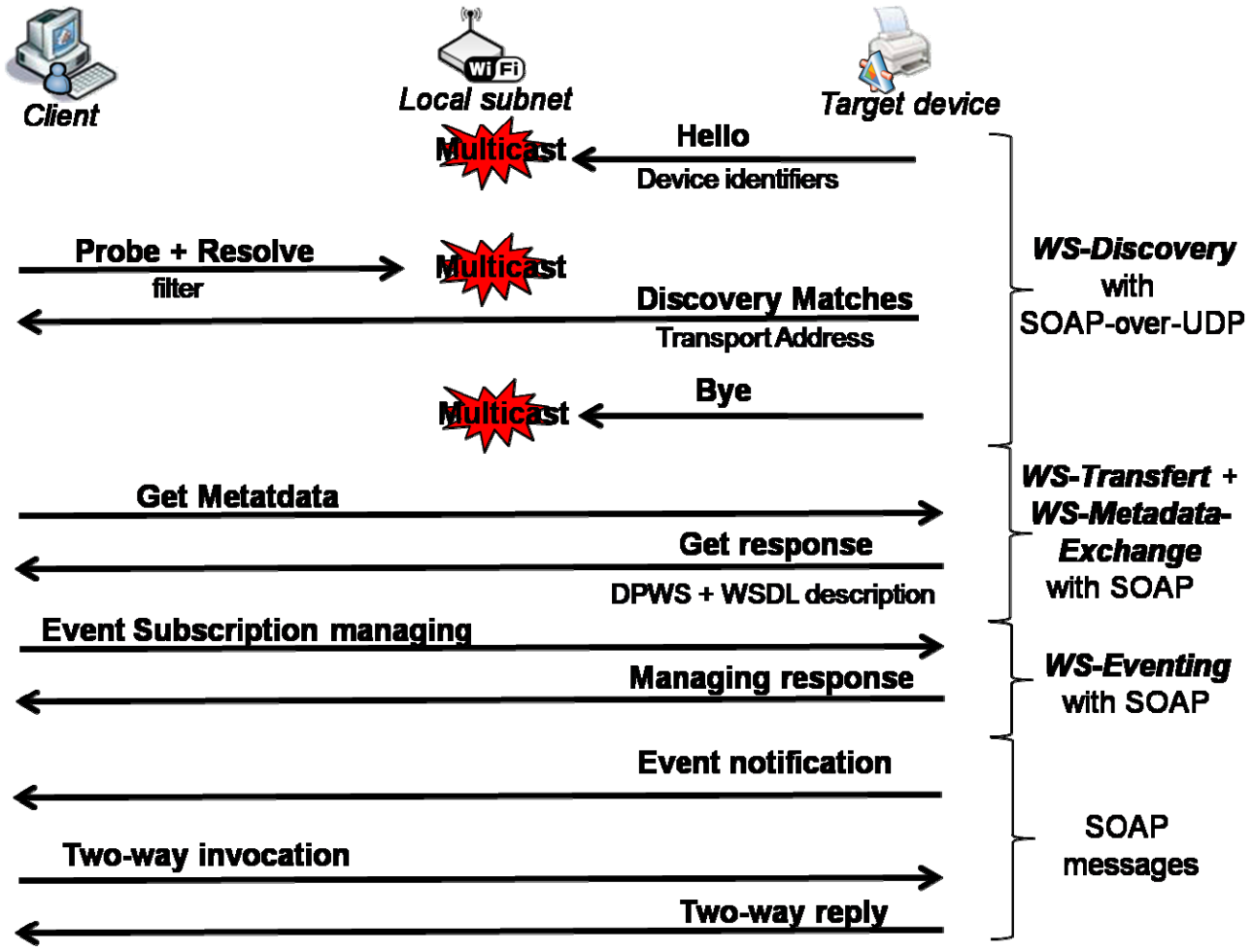

Figure 8: the DPWS conversation overview

3.2.1. SOAP and SOAP-over-UDP

Web Services use SOAP [31] as their messaging protocol. It is a simple, extensible, XML-based protocol that represents as a SOAP envelope composed of 2 parts:

- The **SOAP Header** enables to control how the message is transmitted from its sender to the intended receiver, possibly going through some intermediate nodes. Others blocks can be added to specify routing, reliability, security,...
- The **SOAP Body** contains the application-specific message payload. The body can contain for example service invocation and then its result, or other data for describing services.

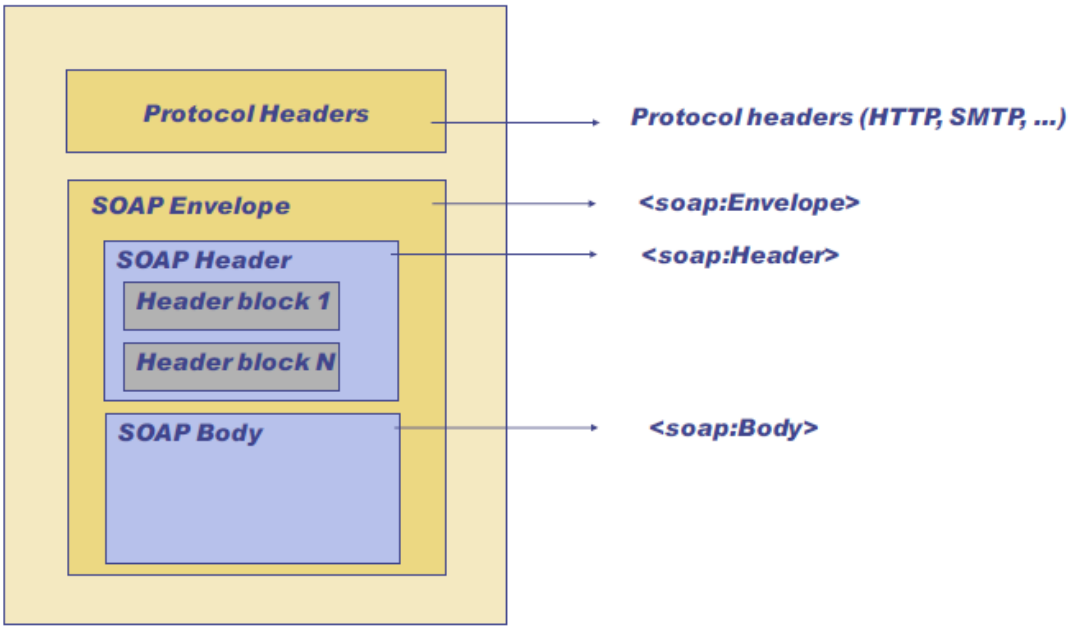

Figure 9: Structure of a SOAP message

The SOAP protocol is transport agnostic, and can therefore be used in combination with different transport layers. SOAP was firstly used with the TCP/HTTP protocol. This transport protocol enables to split big messages into several chunks to be reconstituted by the receiver. Chunks are reassembled in the correct order thanks to a sequence numbers mechanism. TCP provides too guarantees for the delivery of the messages. This implies for each chunk or messages, that sender and receiver exchange acks and synchronization about their sequence number. HTTP is the protocol that allows to transport text content, particularly XML documents, on top of TCP. HTTP provides an additional header that specifies the kind of content that is transported.

In the other hand, many application protocol patterns do not require the guarantees of TCP while others make use of multicast transmission, such as WS-Discovery. The need of low connection overhead, particularly for DPWS where devices are resources constrained, makes UDP a natural choice to transport SOAP message. This is the goal of the SOAP-over-UDP specification, which indicates how to map SOAP envelopes to UDP messages.

The figure 10 is an example of SOAP message for an operation invocation. The header contains metadata on the message to identify it, add a semantic and inform the destination; while the body contains arguments of the invocation. Note that no information allows to differentiate a message with SOAP from a message with SOAP-over-UDP [32]. So for the two protocols, the messages have the same form.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="…">
   <soap:Header>
      <wsa:Action>http://.../TwoWayRequest</wsa:Action>
      <wsa:MessageID>
         urn:uuid:042c5e00-0b84-11e0-8022-08f93903c3d9
      </wsa:MessageID>
      <wsa:To>http://192.168.1.101:4322/SimpleService</wsa:To>
   </soap:Header>
   <soap:Body>
        <n1:deg_fahrenheit xmlns:n1="…">45</n1:deg_fahrenheit>
   </soap:Body>
</soap:Envelope>
```

Figure 10: A SOAP message example

3.2.2. WSDL

The Web Service Language (WSDL) [33] is a specification of a XML language used for describing services and how to invoke them. Thins specification is often used as a contract between the client and the server, for generating stubs or SOAP invocation messages

The WSDL specification is composed of two part (figure 6):

- The **abstract part** that describes the sets of available operations (*port types*) and their signatures. Each *operation* is specified with a name and inputs and output arguments. Input arguments are gathered into an input *message* while output arguments are gathered into an output *message*. Arguments are typed thanks to a XML Schema section at the beginning of the WSDL document. The expressiveness of XML schema allows to specify simple types such as integers or complexes types with sequences for example.
- The **concrete part** that describes how to invoke the operations specified in the abstract part. Each provided *binding* describes for each abstract operation the concrete format and protocol used to invoke them. Commonly bindings enable the operation invocation for several SOAP versions. The SOAP binding specifies the transport protocol (commonly the HTTP transport), a SOAP action that is specified in the SOAP header to indicate the semantic of the message, the message encoding and an operation style (either document or rpc). The style indicates how the arguments of the invocation are presented in the SOAP body. Then *services* and *ports* represent the association of a network address and a specific binding.

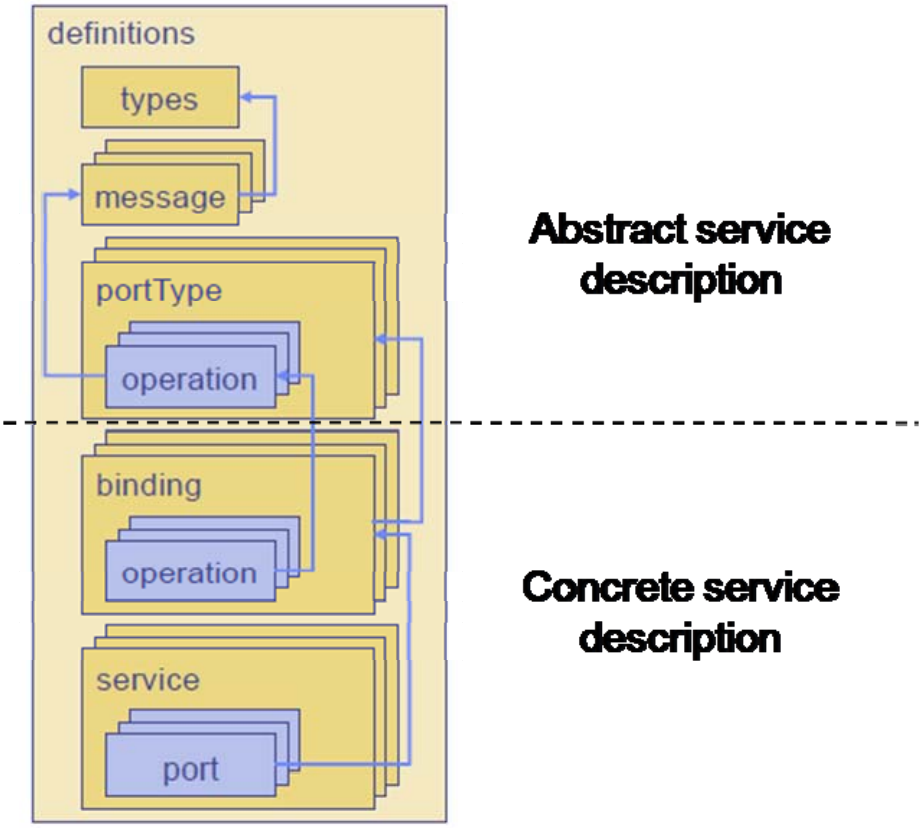

Figure 11: Structure of a WSDL specification

The invocation patterns we can specify are the followings:

- **One-way**: this is an invocation with a possible input argument and without response. This invocation is equivalent to a procedure in the programming world. This pattern is specified with only an input message
- **Two-ways**: this is an invocation with inputs and outputs arguments. It is the most common way to invoke a service. This invocation is equivalent to a function in the programming word. This pattern is specified with an input message and then an output message.
- **Notification**: this is a message from the server without solicitation from the client. This pattern is used when a service want to notify an event. For example, a service may notify when a resource is available. Nevertheless this pattern may require a subscription from the client to indicate to the server that he is interested to receive this kind of notification. This pattern is specified with only an output message that represents the content of the notification.
- **Solicit-Response**: this is an invocation from the server to client. This pattern is specified with an output message and then an input message.

Note that the order between input and output message is important to differentiate a two-ways invocation and a solicit-response invocation. In addition, exceptions during the execution of the operation can be thrown thanks to a fault Message instead of an output message. This message is specified with the fault XML element in the operation. This fault XML element has an attribute to specify the type of the message, in the same way than for the input and output messages.

The DPWS specification enforce that at least the document/literal web service style is supported by the device and the client. This means that message parts are specified with an element that is declared in the XML schema. A message can have several parts, but generally all the arguments are encapsulated in a complex type and the message has only one part.

The following code is an example of WSDL with a one-way, two-way, notification and solicit-response provided operations. The one-way operation use for its input argument an integer while the SolicitResponse operation uses a complex output argument. The binding specifies that these operations are accessible with http transport protocol and the document literal style.

```
<wsdl:types>
   <xsd:schema targetNamespace="…" >
        <xsd:element name="OneWayIntValue" type="xsd:int" />
        …
        <xsd:element name="SLNotif" type="tns:SolicitNotifType"/>
        <xsd:complexType name="SolicitNotifType">
             <xsd:sequence>
                  <xsd:element name="notif_x" type="xsd:int" />
                  <xsd:element name="notif_y" type="xsd:int" />
             </xsd:sequence>
        </xsd:complexType>
        …
   </xsd:schema>
</wsdl:types>

<wsdl:message name="OneWayMessage">
   <wsdl:part name="body" element="tns:OneWayIntValue" />
</wsdl:message>
<wsdl:message name="TwoWayRequestMessage">…</wsdl:message>
<wsdl:message name="TwoWayResponseMessage">…</wsdl:message>
…

<wsdl:portType name="SimpleService">
   <wsdl:operation name="OneWay">
        <wsdl:input
             message="tns:OneWayMessage"
             wsa:Action="http://.../OneWay" />
   </wsdl:operation>
   <wsdl:operation name="TwoWay">
        <wsdl:input
             message="tns:TwoWayRequestMessage"
             wsa:Action="http://.../TwoWayRequest" />
        <wsdl:output
             message="tns:TwoWayResponseMessage"
             wsa:Action="http://.../TwoWayResponse" />
   </wsdl:operation>
   <wsdl:operation name="SimpleNotification">
        <wsdl:output
             message="tns:SimpleNotificationMessage"
             wsa:Action="http://.../SimpleNotification" />
   </wsdl:operation>
   <wsdl:operation name="SolicitResponse">
```

```
        <wsdl:output
             message="tns:SolicitResponseSolicitMessage"
             wsa:Action="http://.../SolicitResponseSolicit" />
        <wsdl:input
             message="tns:SolicitResponseResponseMessage"
             wsa:Action="http://.../SolicitResponseResponse" />
   </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="SimpleServiceBinding"type="tns:SimpleService">
   <n1:binding
        style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />

   <wsdl:operation name="OneWay">
        <n1:operation sopaAction="" />
        <wsdl:input>
             <n1:body use="literal" />
        </wsdl:input>
   </wsdl:operation>
   …
</wsdl:binding>
```

Figure 12: A WSDL specification example

3.2.3. WS-Addressing

The purpose of WS-Addressing [34][35] is to provide a language to specify addressing information. It mainly enables to define appropriate SOAP headers to store the message addressing information that is usually stored in transport protocol headers (such as those used in HTTP), thereby decoupling the message content from the transport and enabling more complexes messages exchanges patterns than the HTTP request-response model. Ws-addressing provides a well-defined way to do asynchronous one-way messaging, with the ability to correlate messages. WS-addressing headers can add the following message addressing properties to a SOAP message:

- **To**: URI address of the receiver of the message
- **Action**: identifies the semantic of the message.
- **Reply endpoint**: an endpoint reference where the receiver can reply
- **Message Id**: a unique identifier for a message
- **Source endpoint**: the endpoint reference of the sender endpoint
- **Fault endpoint**: the endpoint reference where the receiver can reply if a fault occurs
- **RelatesTo**: correlates the message with another request message by message ID
- **ReplyTo**: correlate the message with a request sender by its address.

To send a request, client using WS-Addressing must add a destination and an action and should put other useful information for the addressing and dispatching. To reply, the server must add correlation information such as RelatesTo and ReplyTo to perform the expected conversation.

The key abstraction underlying WS-Addressing is the endpoint reference (EPR), composed of an address and optional reference parameters and metadata description, which identify a resource. The address can be logical or physical, while reference parameters are state information that the service uses to disambiguate resources. A caller obtains an EPR for the resource and uses the Address field for sending a message to the resource.

The figures 8 and 9 are SOAP headers with WS-Addressing fields for the TwoWay operation invocation and its reply. We can see that the RelatesTo field take back the value of the MessageID field.

```
<soap:Header>
   <wsa:Action>http://.../TwoWayRequest</wsa:Action>
   <wsa:MessageID>
        urn:uuid:042c5e00-0b84-11e0-8022-08f93903c3d9
   </wsa:MessageID>
   <wsa:To>http://192.168.1.101:4322/SimpleService</wsa:To>
</soap:Header>
```

Figure 13: An invocation SOAP header example with WS-Addressing

```
<soap:Header>
   <wsa:Action>http://.../TwoWayResponse</wsa:Action>
   <wsa:RelatesTo>
        urn:uuid:042c5e00-0b84-11e0-8022-08f93903c3d9
   </wsa:RelatesTo>
</soap:Header>
```

Figure 14: A response SOAP header example with WS-Addressing

The figure 10 is an example of endpoint reference with a physical address.

```
<wsa:EndpointReference>
   <wsa:Address>
        http://192.168.1.101:4322/SimpleService
   </wsa:Address>
</wsa:EndpointReference>
```

Figure 15: An endpoint reference example

3.2.4. WS-Discovery

The WS-Discovery [36][37] specification defines a multicast discovery protocol to search for and locate resources or, more specifically, Target Services available on the subnet. In WS-Discovery, the device and service metadata information are limited to the strict minimum, in order to reduce the size of the messages and traffic overload. The missing needed metadata information is then exchanged thanks to other protocols.

The primary mode of discovery is a client searching for one more target services. The search can either specify the type of the target or a service scope in which the target resides or both, and is materialised as a probe message (see figure 16) sent to a multicast group. Target Services that matches the probe, sends a Probe Match response (see figure 17) in unicast mode. In the probe match response, the service can specify its endpoint reference, its types and scopes and its physical transport address.

Each modification of these metadata creates a new version that is identified by a number. This identifier is incremented for each new version and allows to differentiate the last version. This version number is specified by the MetadataVersion field as illustrated in the figure 17.

```
<soap:Header>
   <wsa:Action>…/Probe</wsa:Action>
   …
</soap:Header>
<soap:Body>
   <wsd:Probe>
        <wsd:Types>i1:SimpleDevice</wsd:Types>
   </wsd:Probe>
</soap:Body>
```

Figure 16: Probe message example

```
<s12:Header>
   <wsa:Action>…/ProbeMatches</wsa:Action>
   …
</s12:Header>
<s12:Body>
   <wsd:ProbeMatches>
        <wsd:ProbeMatch>
             <wsa:EndpointReference>
                  <wsa:Address>urn:2345</wsa:Address>
             </wsa:EndpointReference>
             <wsd:Types>i1:SimpleDevice i0:Device</wsd:Types>
             <wsd:Scopes>http://www.ws4d.org</wsd:Scopes>
             <wsd:XAddrs>
                  http://192.168.1.101:4321/SimpleExampleDevice
             </wsd:XAddrs>
             <wsd:MetadataVersion>4</wsd:MetadataVersion>
        </wsd:ProbeMatch>
   </wsd:ProbeMatches>
</s12:Body>
```

Figure 17: ProbeMatches message example

To minimize the need for polling, when a Target Service joins the network, it announces itself by sending a multicast Hello Message (see Figure 18). By listening to the multicast group, clients can detect newly-available Target Service without repeated probing. When leaving the network, a Target Service announces this through by a Bye Message. Hello and Bye messages contain the current metadata of the device. By receiving Hello and Bye message, clients update their knowledge of the available devices. To this aim, devices sent Hello message each time they do a modification in their metadata.

```
<soap:Header>
   <wsa:Action>…/Hello</wsa:Action>
   <wsa:To>
        urn:docs-oasis-open-org:ws-dd:ns:discovery:2009:01
   </wsa:To>
   <wsd:AppSequence InstanceId="1292772221" MessageNumber="1" />
</soap:Header>
<soap:Body>
   <wsd:Hello>
        <wsa:EndpointReference>
             <wsa:Address>urn:2345</wsa:Address>
        </wsa:EndpointReference>
        <wsd:Types>i1:SimpleDevice i0:Device</wsd:Types>
        <wsd:Scopes>http://www.ws4d.org</wsd:Scopes>
        <wsd:XAddrs>
             http://192.168.1.101:4321/SimpleExampleDevice
        </wsd:XAddrs>
        <wsd:MetadataVersion>4</wsd:MetadataVersion>
   </wsd:Hello>
</soap:Body>
```

Figure 18: Hello message example

It is not mandatory that the metadata give the transport address, which is needed to access to the device. In this case, the client can ask explicitly the transport address of a device identified by his endpoint, through a Resolve multicast message (see figure 19). The device that has this endpoint, answers with his transport address in a unicast message.

```
<soap:Header>
   <wsa:Action>…/Resolve</wsa:Action>
   …
</soap:Header>
<soap:Body>
   <wsd:Resolve>
        <wsa:EndpointReference>
             <wsa:Address>urn:2345</wsa:Address>
        </wsa:EndpointReference>
   </wsd:Resolve>
</soap:Body>
```

Figure 19: Resolve message example

WS-Discovery proposes 2 operational modes:

- The **ad hoc mode** (see Figure 19) is the mode provided in the earlier versions of WS-Discovery. This mode establishes a discovery conversation between a client and a Target Service through multicast messages and unicast responses through the preceding described messages. Multicast messages are sent to the IP address 239.255.255.250 and the discovery port (3702) with the SOAP-over-UDP protocol. The figure 19 shows the discovery pattern conversation for the ad hoc operational mode.
- The **managed mode** is a mode proposed in the last version of WS-Discovery in order to be scalable to enterprise-wide scenarios. This mode introduces the notion of Discovery proxy that becomes an intermediate between clients and Target Services. A discovery proxy has two functions: suppressing multicast discovery to reduce network traffic, and extending the network reach for the discovery protocol beyond the local subnet. Instead of sending message by multicast, they are sent by unicast to the discovery proxy, as illustrated in the figure 20. This mode allows to send messages with the SOAP with HTTP protocol instead of SOAP-over-UDP, since the multicast is removed.

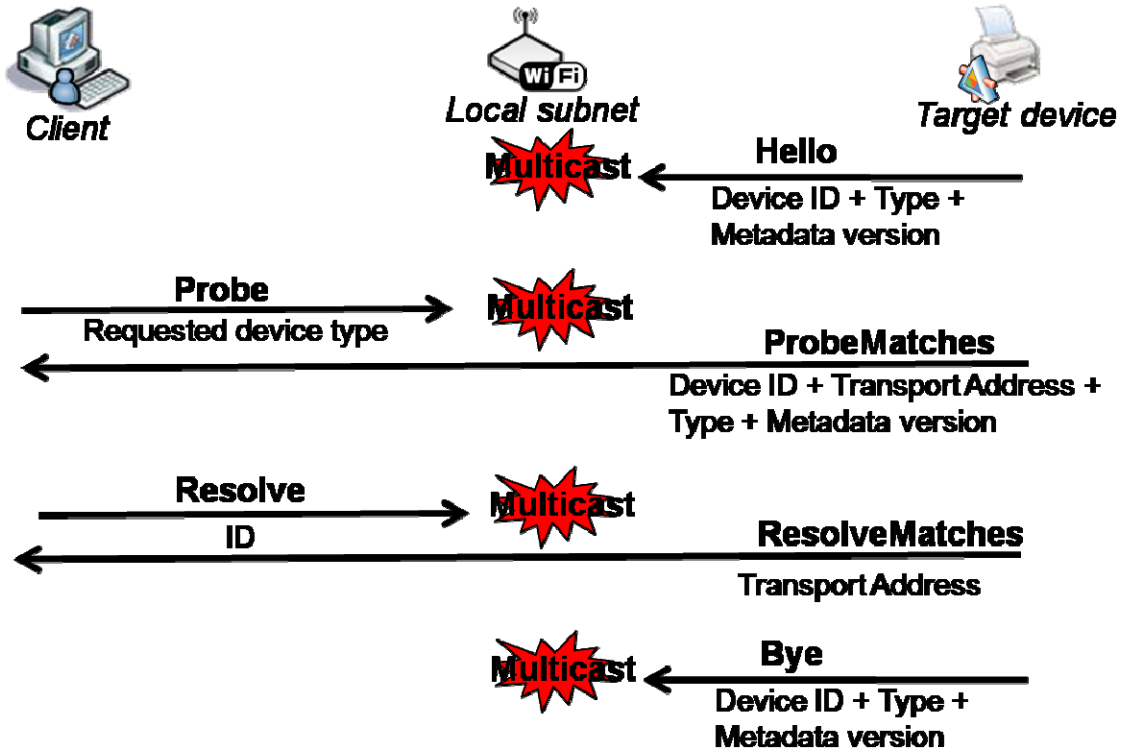

Figure 20: Ad hoc WS-Discovery conversation

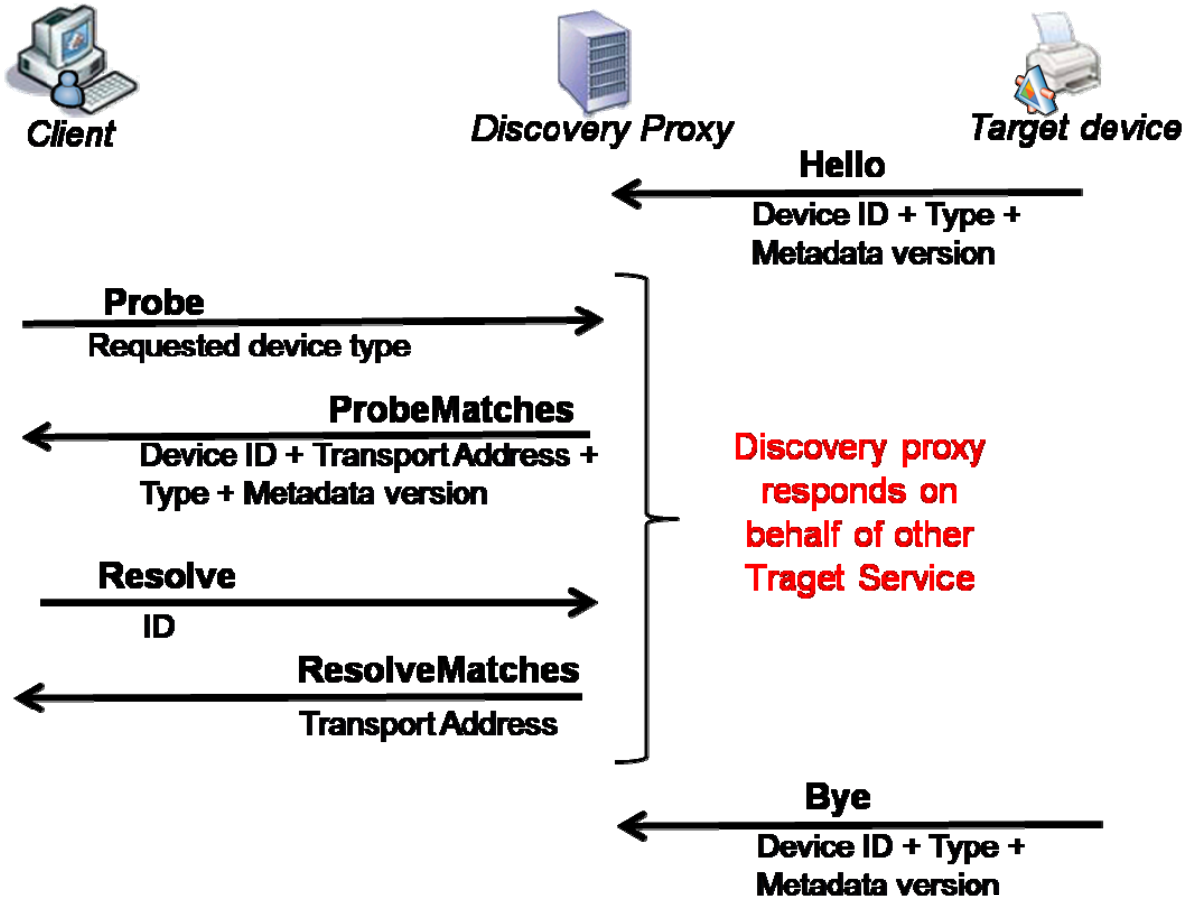

Figure 21: Managed WS-Discovery conversation

WS-Discovery allows switching dynamically from the ad hoc to the managed mode. When WS-Discovery is used in ad hoc mode, the new discovery proxy is firstly considered as a Target Service of type DiscoveryProxy. Thus it sends multicast Hello and Bye messages to announce themselves. They listen for multicast messages and can store the metadata of the other Target Services. When they receive a probe or resolve message, they can respond for the wanted Target Service and proposes to the client to switch to the managed mode, as now the client knows the discovery proxy's address.

3.2.5. WS-Transfert & WS-MetadataExchange

WS-Transfert [38][39] and WS-MetadataExchange [40][41] enables services and client to exchange metadata. In the context of DPWS, they exchange metadata that describes the devices properties such as their manufacturer, and their hosted services with their available operations and signature. Firstly, to discover the properties of a device and the list of hosted services, the client needs to retrieve his DPWS description (see section 3.1). From the list of hosted services, we obtain their endpoint that allows the client to retrieve their WSDL description (see section 3.2.2). In this document we present only the messages that are used for these purposes.

Metadata associated to a Web Service are considered as resources: a Web Service addressable by an endpoint reference and represented by an XML infoset. This representation can be retrieved by 2 ways:

- A **Get operation invocation** with the **WS-Transfert** protocol on the endpoint of the resource. This operation corresponds to the standard way to retrieve any resource on the Web.
- A **GetMetadata operation invocation** with the **WS-MetadataExchange** protocol on the endpoint of the Web Service itself. WS-MetadataExchange is a protocol that defines how metadata can treated as WS-Transfer resources for retrieval purposes, how metadata can be embedded in Web Service endpoint reference and, how Web Service endpoints can support a request-response interaction for the retrieval of metadata. This mechanism can be used if we have no information on the metadata endpoint.

In the context of DPWS, the device metadata information is available on the endpoint specified by the device, while the hosted service metadata information is not properly available at the endpoint of the service itself. That is why for retrieving the device metadata, the client performs a Get WS-Transfer request while he performs a GetMetadata request to get the service metadata.

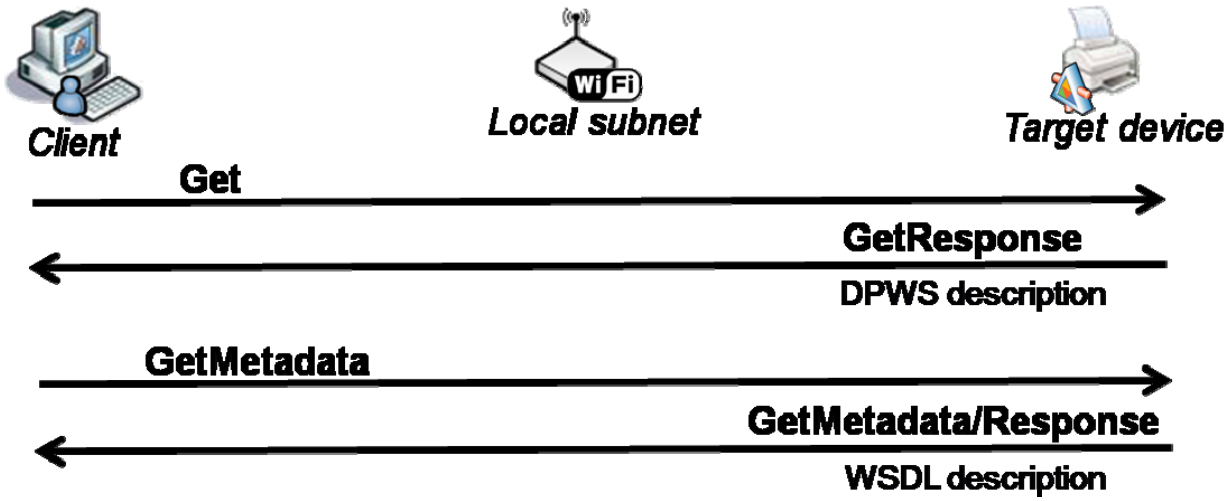

Figure 22: WS-Transfert/WSMetadataExchange conversation for DPWS

For the WS-transfert and WS-MetadataExchange metadata retrieval request, the important information are contained in the SOAP header while the body is almost empty. In the header is specified the WS-Addressing action and the destination that represent the address where the metadata are available or the concerned Web Service. The figure 22 is an example of Get WS-Tranfert request to retrieve the metadata of the device identified by the endpoint urn:2345. The figure 24 is an example of GetMetadata WS-MetadataExchange request to retrieve the WSDL description of the service http://192.168.1.101:4322/SimpleService.

For the metadata retrieval responses for WS-Transfert or WS-MetadataExchange, metadata units are contained in MetadataSection where the dialect attribute specifies the type of metadata by providing the URL of the root element. The figure 23 is an example of Get WS-

Transfert response: the response provides ThisModel, ThisDevice and Relationship section for a device (see section 3.1). Instead of providing the document, a metadata section can just give the location where the resource metadata is available, as illustrated in figure 25 for the WSDL description.

```
<soap:Header>
   <wsa:Action>…/Get</wsa:Action>
   <wsa:MessageID>…</wsa:MessageID>
   <wsa:To>urn:2345</wsa:To>
</soap:Header>
<soap:Body />
```

Figure 23: A DPWS description Get WS-Transfert request example

```
<soap:Header>
   <wsa:Action>…/GetResponse</wsa:Action>
   …
</soap:Header>
<soap:Body>
   <wsx:Metadata>
        <wsx:MetadataSection
             Dialect="http://docs.oasis-open.org/.../ThisModel">
             <dpws:ThisModel>
                  <dpws:Manufacturer>Materna</dpws:Manufacturer>
                  …
             </dpws:ThisModel>
        </wsx:MetadataSection>
        <wsx:MetadataSection
             Dialect="http://docs.oasis-open.org/.../ThisDevice">
             <dpws:ThisDevice>
                  <dpws:FriendlyName>SimpleDev</dpws:FriendlyName>
                  …
             </dpws:ThisDevice>
        </wsx:MetadataSection>
        <wsx:MetadataSection
             Dialect="http://docs.oasis-open.org/../Relationship">
             <dpws:Relationship Type="http://.../host">
                  <dpws:Host>
                       …
                  </dpws:Host>
                  <dpws:Hosted>
                  …
                  </dpws:Hosted>
                  <dpws:Hosted>
                       …
                  </dpws:Hosted>
             </dpws:Relationship>
        </wsx:MetadataSection>
   </wsx:Metadata>
</soap:Body>
```

Figure 24: A DPWS Get WS-Transfert response example

```
<soap:Header>
   <wsa:Action>…/GetMetadata/Request</wsa:Action>
   <wsa:MessageID>…</wsa:MessageID>
   <wsa:To>http://192.168.1.101:4322/SimpleService</wsa:To>
</soap:Header>
<soap:Body>
   <wsx:GetMetadata />
</soap:Body>
```

Figure 25: A WSDL description GetMetadata WS-MetadataExchange request example

```
<soap:Header>
   <wsa:Action>…/GetMetadata/Response</wsa:Action>
   …
</soap:Header>
<soap:Body>
   <wsx:Metadata>
        <wsx:MetadataSection
             Dialect="http://schemas.xmlsoap.org/wsdl">
        <wsx:Location>http://.../description.wsdl</wsx:Location>
        </wsx:MetadataSection>
        <wsx:MetadataSection
             Dialect="http://.../Relationship">
             <dpws:Relationship Type="http://.../host">
                  <dpws:Host>
                  …
                  </dpws:Host>
                  <dpws:Hosted>
                  …
                  </dpws:Hosted>
             </dpws:Relationship>
        </wsx:MetadataSection>
   </wsx:Metadata>
</soap:Body>
```

Figure 26: A WSDL description GetMetadata WS-MetadataExchange response example

3.2.6. WS-Eventing

WS-Eventing [42][43] describes a protocol allowing one Web Service (called an “Event sink”) to register interest (called “subscription”) with another Web Service (called an “Event source”) in receiving messages about events (called “notifications”).

A subscription can expire over time. This time limit can be specified by duration or a date by the client when he subscribe. Then the status of this subscription can be consulted. The subscription end is notified when it expires. WS-Eventing allows to renew a subscription to extend the expiration limit.

WS-Eventing provides a set of messages to manage a subscription for its initialisation (Subscribe messages), its renewal (Renew messages), the consultation of its status (GetStatus message) and its termination (Unsubscribe message). The figure 26 illustrates the WS-Eventing conversation.

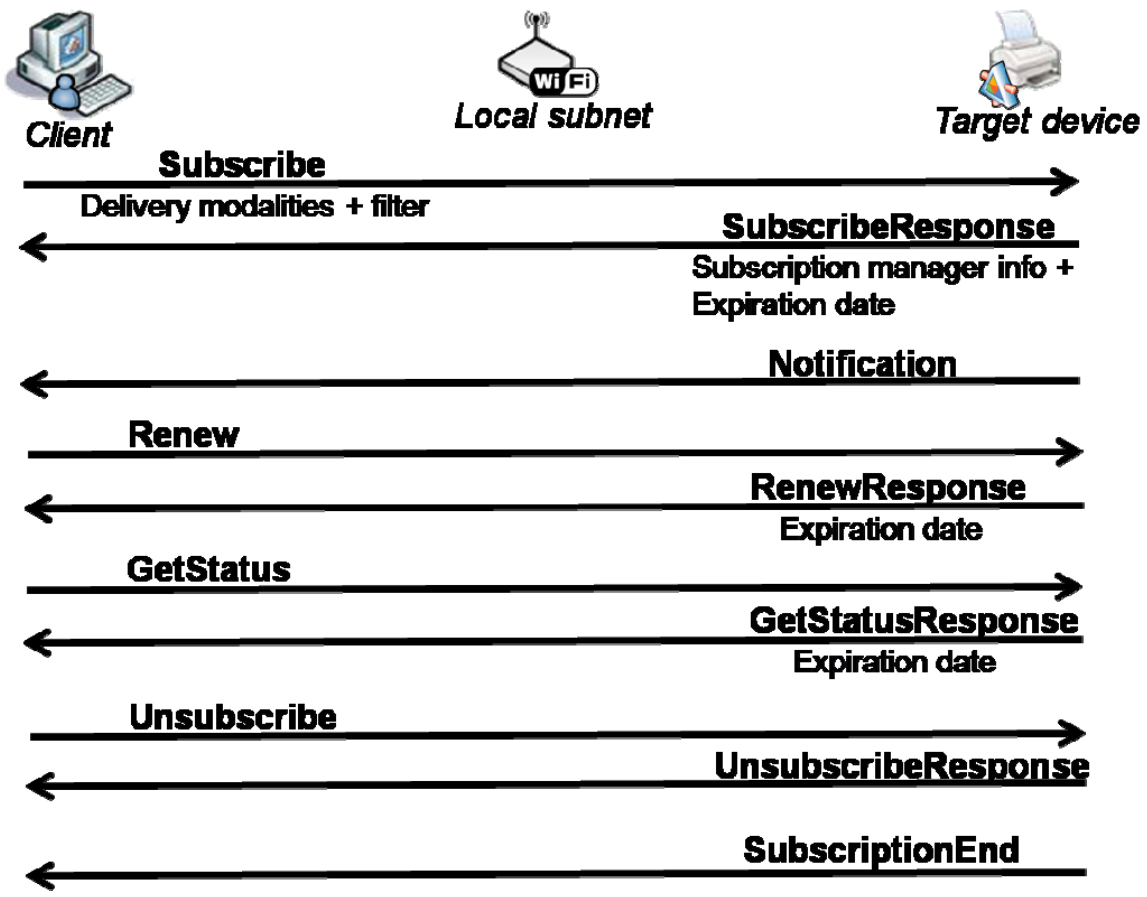

Figure 27: WS-Eventing conversation

When a client sends a subscribe message to the event source, it specifies the notification modalities (i.e the address that will receive the notifications and other custom parameters), an expiration limit, and can specify a filter in order to receive only some kind of notification. The figure 27 is a subscribe message example where the client subscribe to the event with the WS-Addressing action SimpleNotification from the service http://192.168.1.101:4323/EventingService. The client will receive the notification at the address http://192.168.1.100:50886/EventSink and this subscription will expires in 60 seconds.

```
<soap:Header>
   <wsa:Action>…/Subscribe</wsa:Action>
   …
   <wsa:To>http://192.168.1.101:4323/EventingService</wsa:To>
</soap:Header>
<soap:Body>
   <wse:Subscribe>
        <wse:Delivery Mode="…/Push">
             <wse:NotifyTo>
                  <wsa:Address>
                       http://192.168.1.100:50886/EventSink
                  </wsa:Address>
                  <wsa:ReferenceParameters>
                       <wse:Identifier>
                  urn:uuid:11f59290-0b84-11e0-8023-08f93903c3d9
                       </wse:Identifier>
                  </wsa:ReferenceParameters>
             </wse:NotifyTo>
        </wse:Delivery>
        <wse:Expires>PT60S</wse:Expires>
        <wse:Filter
             Dialect="http://.../Action">
                  http://.../SimpleNotification
        </wse:Filter>
   </wse:Subscribe>
</soap:Body>
```

Figure 28: A Subscribe message example

The service answers by giving the subscription manager for this subscription. Each subscription has a unique subscription manager with a unique identifier. This identifier will be used for each managing action on the subscription such as a unsubscribe, a status consultation or a renewal. Note that the address of the subscription manager may be different of the service that send the notification if the service delegates the subscription managing to another service. The figure 28 is the response of the message in figure 27, while figure 29 is an unsubscribe message on the same subscription manager. Responses to these requests confirm the unsubscribe (see figure 30), or the renewal with a new expiration limit.

```
<soap:Header>
   <wsa:Action>…/SubscribeResponse</wsa:Action>
   …
</soap:Header>
<soap:Body>
   <wse:SubscribeResponse>
        <wse:SubscriptionManager>
             <wsa:Address>
                  http://192.168.1.101:4323/EventingService
             </wsa:Address>
             <wsa:ReferenceParameters>
                  <wse:Identifier>
                  urn:uuid:11145a50-0b84-11e0-bf2f-da194d83aab9
                  </wse:Identifier>
             </wsa:ReferenceParameters>
        </wse:SubscriptionManager>
        <wse:Expires>PT60S</wse:Expires>
   </wse:SubscribeResponse>
</soap:Body>
```

Figure 29: A SubscribeResponse message example

```
<soap:Header>
   <wsa:Action>…/Unsubscribe</wsa:Action>
   …
   <wsa:To>http://192.168.1.101:4323/EventingService</wsa:To>
   <wse:Identifier
        wsa:IsReferenceParameter="true">
        urn:uuid:11145a50-0b84-11e0-bf2f-da194d83aab9
   </wse:Identifier>
</soap:Header>
<soap:Body>
   <wse:Unsubscribe />
</soap:Body>
```

Figure 30: An Unsubscribe message example

```
<soap:Header>
   <wsa:Action>…/UnsubscribeResponse</wsa:Action>
   …
</soap:Header>
<s12:Body />
```

Figure 31: An Unsubscribe response message example

3.2.7. MTOM

SOAP Message Transmission Optimization Mechanism (MTOM) [88] is a method to efficiently sending binary data to and from Web Services. The efficiency claim of MTOM refers to the size of the messages sent across the wire. Since SOAP uses XML, any binary data in SOAP message will be encoded as text. This is usually done by using Base64 encoding which increases the size of the binary data by 33 %. MTOM provides a way to send the binary data in its original binary form, avoiding any increase in size to encoding it into text. MTOM does not address the efficiency of serializing or deserializing the messages.

MTOM specification is defined as three related features:

- The **Abstract SOAP Transmission Optimization Feature** for sending and receiving SOAP messages that contain binary data. This features introduces the concept of sending the binary data separately of the rest of the XML infoset.
- The **Optimized MIME Multipart/Related Serialization of SOAP Messages** defines that the serialized XML infoset will include XML-Binary Optimized Packaging (XOP) in place of the binary data, and the binary data will be represented together in a MIME container.
- The **HTTP SOAP Transmission Optimization Feature** defines how the above MIME and XOP SOAP messages are sent over HTTP.

Using MTOM with Web Service allows for example to take in input or output a binary data, such a picture. So such an argument is specified with the type Binary64 which is a XML Schema primitive type.

3.2.8. WS-Security

WS-Security [44][45] is a standard set of extensions that can be used when building secure Web Service. This specification proposes mechanisms to verify integrity and confidentiality of the exchanged messages. Although it is flexibly designed to be used in conjunction with a variety of security models including PKI, Kerberos and SSL, we will present only the mechanisms used in DPWS.

The security mechanisms used in DPWS aims to preserve integrity, confidentiality, authentication, availability and target the following attacks:

- **Message alteration:** message content may be changed by an attacker.
- **Message disclosure:** an attacker may get the content of a message. The message is no longer confidential.
- **Denial of service:** an attacker may send messages that consume resources at application level or network level and make the service unavailable.
- **Spoofing (man-in-the-middle):** an attacker successfully masquerades as another by falsifying data and gaining illegitimate advantage.
- **Illegal access** to a resource.
- **Non-repudiation:** an actor that sends a message may deny later that he sent it.

The DPWS security mechanisms are WS-Discovery compact signature for the discovery phase, and Secure Socket layer (SSL) for the other phases of a DPWS conversation to avoid these attacks. These techniques are mainly based on the concepts of certificates and asymmetric cryptography.

Certificates

The certificate aims to be credentials that authenticate the sender of a message. A certificate is assigned to the sender by a certification authority that is trusted by everybody. Thanks to this certificate, the receiver can verify the identity of the sender. There exist several forms of certificates, but in the specification of DPWS, only X.509 certificates are used. A X.509 certificate contains information about the sender and public key that can be used to decrypt encrypted contents. When the sender wants to attach a certificate to a SOAP message, he specifies it in a BinarySecurityToken sub-block of the Security bloc header. The X.509 certificate is not a XML content; thus it is inserted as a text value after being encrypted. The figure 31 is an example of X.509 certificate encapsulated in a WS-Security sub-block.

```
<wsse:BinarySecurityToken
   ValueType="...#X509v3"
   wsu:Id="X509Token"
   EncodingType="...#Base64Binary">
        MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
</wsse:BinarySecurityToken>
```

Figure 32: A X.509 certificate example

Assymetric cryptography

The principle of asymmetric cryptography is the random generation of a pair of keys: a private key and a public key, where one of the key can decrypt only the content crypted by the other key. The public key is then registered in a certificate by a certification authority to prove that this key belong to the given actor. This mechanism can be used in two ways:

- **To prove the identity of the sender**: the sender generates the private and public keys. He crypts the message with his private key. Then if the receiver is able to decrypt the message with the public key, the sender is safely authenticated, since that the certificate proves that the public belong to the sender. The receiver must have a way to verify that the decrypting is correct.
- **To ensure that only the receiver can read the message**: the receiver generates the private and the public key. The sender crypts his message with the certificated public key. Since only the private key can decrypt this message, the sender is insured that only the targeted receiver can read the message.

WS-Discovery Compact signature

The WS-Discovery Compact signature intervene during the discovery phase on the multicast or unicast SOAP-over-UDP messages in order to prevent message alteration, denial of service at the service level, replay and spoofing attacks. The principle is the generation of a

signature by hashing the message content and encrypt. The crypting of the signature and its decrypting are based on the asymmetric cryptography to prove the identity of the sender, in conjunction with the X.509 certificate to transport the sender's public key. The sender sends the message with the obtained signature and his certificate to provide his public key. With the received public key, the receiver can decrypt the signature and compare it to the hash of the message content. The figure 32 illustrates the signing and verification processes. Since the value of the signature depends on the message content, by comparison between the decrypted signature and the rehashed content, the receiver can detect if the content have been modified. In a second time the comparison between the digest obtained by the receiver and the decrypted signature allows him to verify that the decrypting is correct. Thus he can verify the identity of the receiver.

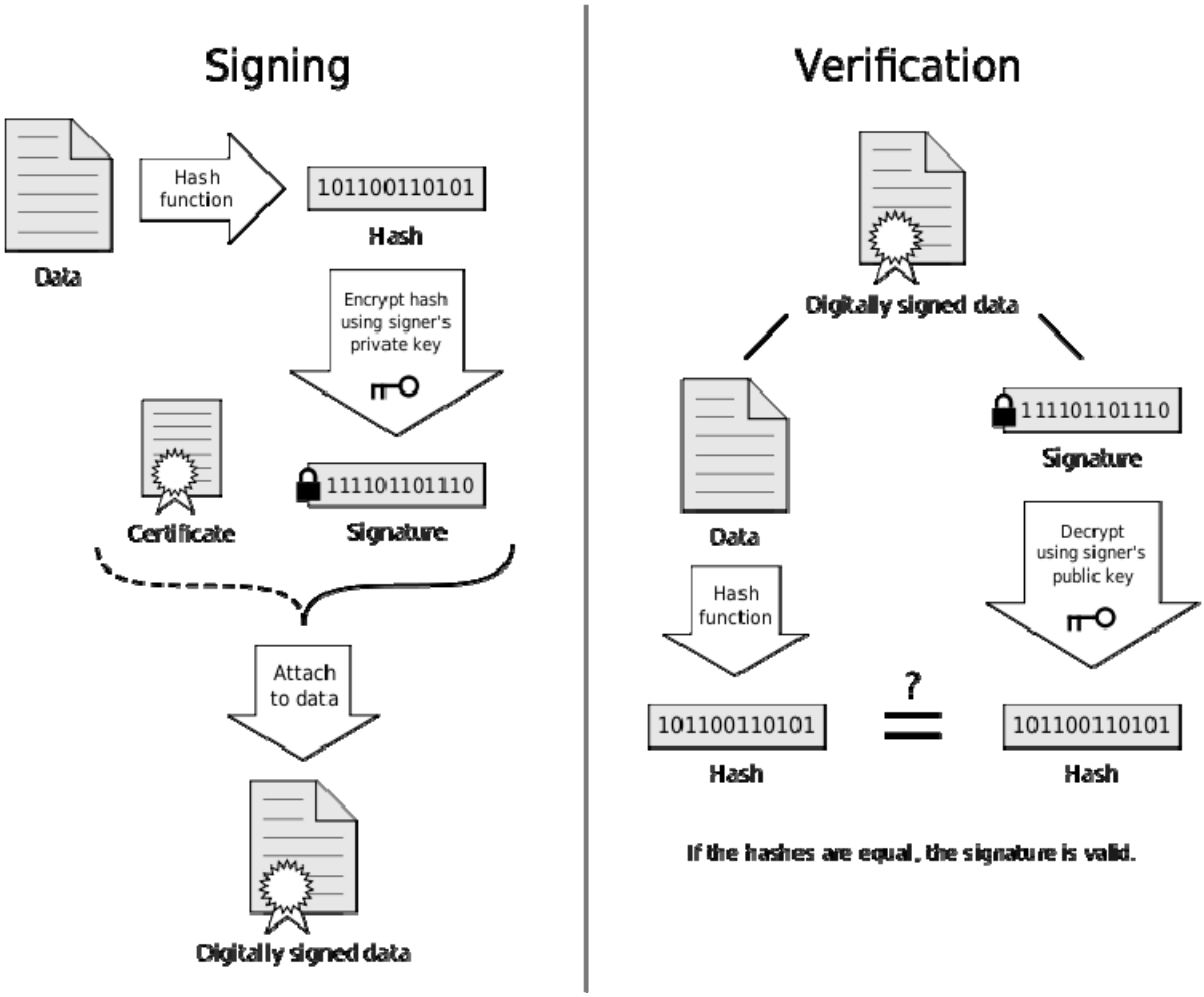

Figure 33: The digital signature mechanism

The figure 34 is an example of XML digital signature using the X.509 certificate of the figure 31. The SignedInfo element gives information about how compute the digest. In this example, the digest is computed by applying the xml-exc-c14n canonicalization algorithm and then the rsa-sha hashing algorithm on the body part of the message. The value of the encrypted signature is contained in the SignatureValue element while the KeyInfo element provides a reference to the used X.509 certificate.

```
<ds:Signature>
   <ds:SignedInfo>
```

```
        <ds:CanonicalizationMethod Algorithm="…/xml-exc-c14n#"/>
        <ds:SignatureMethod Algorithm="…/xmldsig#rsa-sha1"/>
        <ds:Reference URI="#body">
             <ds:Transforms>
                  <ds:Transform Algorithm="…/xml-exc-c14n#"/>
             </ds:Transforms>
             <ds:DigestMethod Algorithm="…/xmldsig#sha1"/>
             <ds:DigestValue>LyLsF094hPi4wPU…</ds:DigestValue>
        </ds:Reference>
   </ds:SignedInfo>
   <ds:SignatureValue>Hp1ZkmFZ/2kQLXDJbchm5…</ds:SignatureValue>
   <ds:KeyInfo>
        <wsse:SecurityTokenReference>
             <wsse:Reference URI="#X509Token"/>
        </wsse:SecurityTokenReference>
   </ds:KeyInfo>
</ds:Signature>
```

Figure 34: A signature example

Secure Socket Layer (SSL)

DPWS uses through the WS-Security specification, the SSL protocol to create secure channels between a device and a client, after the client has verified the identity of the server. The secure channel is ensured by the encryption by common key between the client and the server in order to crypt and decrypt. The goal of the first phase to establish the secure channel is the sending of this symmetric key (called a session key) from the client to the server (i.e. the server and the client share the same key to encrypt and decrypt). For this purpose, SSL uses the asymmetric cryptographic mechanism to ensure that only the receiver can read the message. The server has the private and public key and he firstly sends the public key through a X.509 certificate. Then the client sends the session key to the server by crypting it with the public key of the server. After that the client and the server can exchange crypted messages with the session key. This process is illustrated in the figure 33.

With the first phase for establishing a SSL connection, it provides authentication features while the secure channel prevent message disclosure attacks.

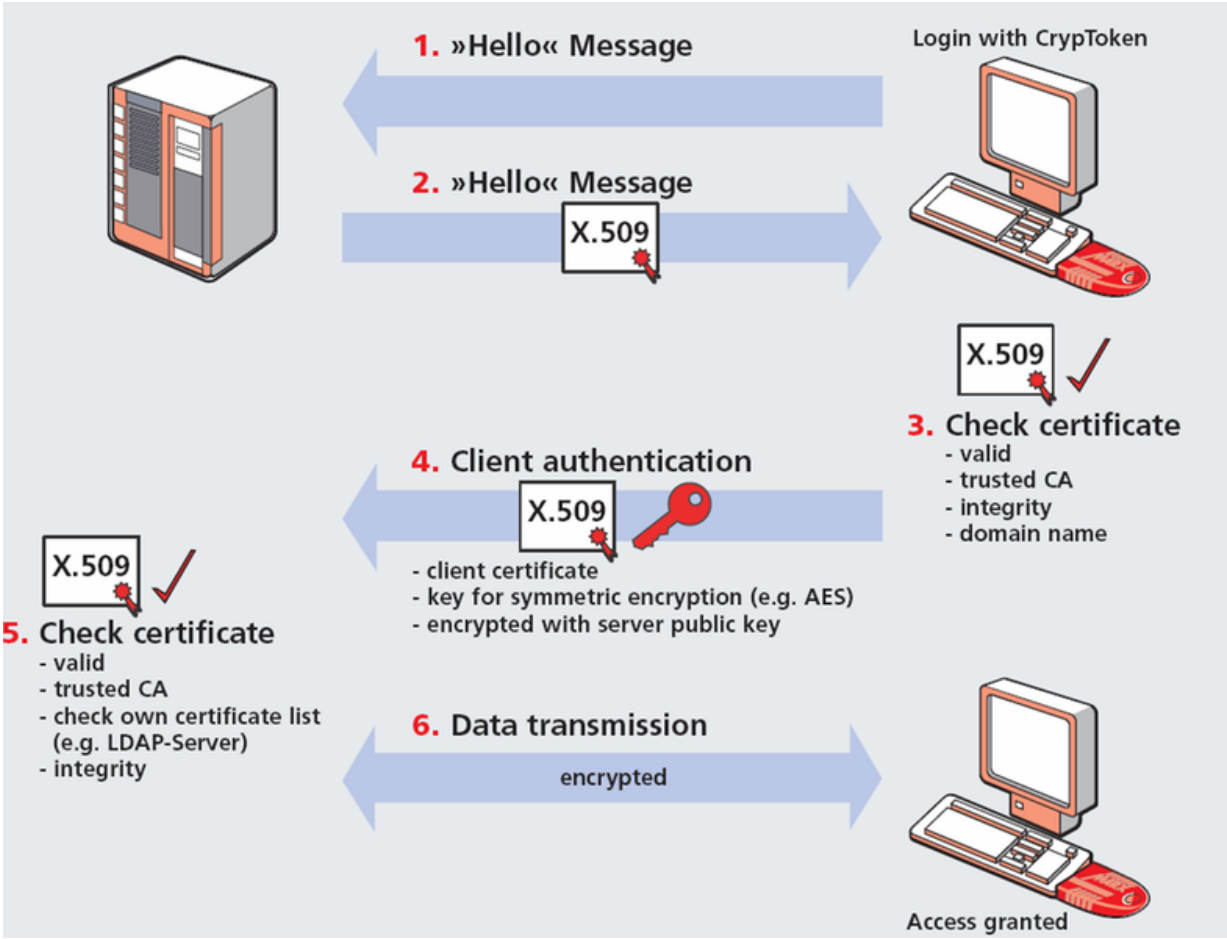

Figure 35: SSL mechanism

3.2.9. WS-Policy

Web Service Policy [46][47] defines a framework for expressing assertions that represents requirements, capabilities or other property or behaviour. In DPWS, WS-Policy is used to specify the compliance of the service with a version of DPWS. The figure 35 is an example of a policy to indicate that the service is compliant with DPWS version of 2009.

```
<wsp:Policy
   xmlns:dpws="http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01"
   xmlns:wsp="http://www.w3.org/ns/ws-policy">
   <dpws:Profile />
</wsp:Policy>
```

Figure 36: A DPWS policy example

3.3. Specification of a set of implementation constraints

DPWS is mainly a specification based on Web services standard. Because the conditions for an SOA plateform to be efficient are not the same on computers and on smart devices, DPWS aims also defining a set of implementations constraints to satisfy the devices

requirements. Moreover the used Web Services specifications allows a lot of different options that are not necessarily compatible with all the devices. So DPWS provides a set of minimal compatibility constraints to ensure that all the devices can communicate.

The main points discussed in the DPWS specification are the following:

- The **max size of the data**
  - The **message size** must be enough small to be delivered in one UDP packet for the discovery conversation.
  - Max **URI size**
  - Max **field size**
- On a **service EPR**:
  - Recommendation on his form and on the address property
  - The EPR must be fix
- A service must **accept the following standard version or options**
  - SOAP 1.2
  - HTTP Binding as specified in its WSDL
  - Document literal style for the WSDL. This implies, according to the WS-I compliance [65] that each message has only one part and multiple arguments are wrapped into a complex type.
  - UTF-8 encoding
  - Chunked transfert coding for HTTP messages
  - Push delivery mode for eventing
  - Cypher suite RSA, RC4 on 128 bits key and SHA algorithms for the security
- The **required information in the SOAP header**
- **Behaviour** of the devices and hosted services
  - Only devices participates to the discovery
  - Devices must be able to match the main matching rules manage the WS-Discovery messages.
  - The device must increment its metadata version number if its metadata are modified.
  - The service should send a fault message if the received WSDL description is inconsistent

3.4. DPWS versions

There exist two versions of DPWS: an early version during the SIRENA project in 2006 (DPWS 1.0), and another OASIS standard version released in 2009 (DPWS1.1). The main difference between theses versions are the versions of the used Web Service standards, as illustrated in the figure 36.

DPWS 1.0	DPWS 1.1
Namespace: http://schemas.xmlsoap.org/ws/2006/02/devprof	Namepsace: http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01
WS-Discovery [36] – April 2005	WS-Discovery 1.1 [37] – July 2009
WS-MetadataExchange [40] – Sept. 2004	WS-MetadataExchange 1.1 [41] – August 2005
WS-Transfert [38] – Sept. 2004	WS-Transfert [39] – Sept. 2006
WS-Addressing [34] – August 2004	WS-Addressing [35] – August 2006
SOAP-over-UDP [31] – Sept. 2004	SOAP-over-UDP [32] – July 2009
WS-Policy [46] – Sept. 2004	WS-Policy 1.5 [47] – Sept. 2007
WS-Eventing [42] – Aug. 2004	WS-Eventing [43] – March 2006
WS-Security 1.0 [44] – March 2004	WS-Security 1.1 [45] – Feb. 2006
SOAP 1.2	SOAP 1.2
WSDL 1.1	WSDL 1.1

Figure 37: DPWS versions comparison

Globally the different versions of the used Web Service standard offer the same features. However WS-Discovery 1.1 adds a new feature compared the previous version: the discovery proxy that enables to avoid multicast messages and allows the communication between devices that are not in the same subnet. This new functionality is important in the context of Choreos to deal with large scale web service composition on devices.

Another difference between DPWS 1.0 and DPWS 1.1 is the adding of implementation constraints and details on the behaviour of devices and hosted services. For example

4. Prototypes & implementations

The DPWS prototypes aim at developing DPWS devices and their hosted services on various platforms such as computers or resources constrained devices like PDA, phones or sensors. They have to provide at least the features specified in the Web Service standards used in the DPWS specification. In addition they must satisfy the implementation constraints of the DPWS specification. During our experimentations with the existing DPWS implementations, we see in hindsight that the high-level principles of them are quite similar. Although the syntax of some artefacts is different, some parts of the different DPWS implementation are designed in a similar way. That is why in the first subsection we are going to expose general principles that we retrieve on many DPWS implementations.

In the next subsections, we are going to present in a more detailed way the existing prototypes that enable to develop DPWS services. The explanation in these subsections will refer to the general explanations of the first subsection. For each implementation, we will present its principles of use for the main DPWS features through an example with a counter device that exposes a service for incrementing its internal value. The device will have the following characteristics:

- FriendlyName: Counter Device
- Uuid: 123456
- Type: CounterDeviceType
- Scope: CounterScope
- IP address: 192.168.2.100
- Port: 4320

The following WSDL (figure 38) specifies the operations of the hosted service. The operation *Increment* takes in input an integer that corresponds to a step used to increment, then the operation returns the new value of the counter.

```
<wsdl:types>
   <xsd:schema targetNamespace="…" >
        <xsd:complexType name="TwoWayReq">
             <xsd:sequence>
                  <xsd:element name="Value" type="xsd:int" />
             </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="TwoWayResp">
             <xsd:sequence>
                  <xsd:element name="Value" type="xsd:int" />
             </xsd:sequence>
        </xsd:complexType>
   </xsd:schema>
</wsdl:types>

<wsdl:message name="IncrementInput">
   <wsdl:part name="TwoWayReq" element="tns:TwoWayReq" />
</wsdl:message>
<wsdl:message name="IncrementOutput">
   <wsdl:part name="TwoWayResp" element="tns:TwoWayResp" />
```

```
</wsdl:message>


<wsdl:portType name="CounterPortType">
   <wsdl:operation name="Increment">
        <wsdl:input
             message="tns:IncrementInput"
             wsa:Action="http://.../IncrementRequest" />
        <wsdl:output
             message="tns:IncrementOutput"
             wsa:Action="http://.../IncrementResponse" />
   </wsdl:operation>
</wsdl:portType>
```

Figure 38: WSDL for the counter hosted service

We are going to show the use of the DPWS implementations through the following scenario:

1. The server starts
2. The client starts and probes with a device type. The server answers with a probeMatches.
3. The client gets the metadata of the device and prints its friendlyName.
4. The client invokes the two-way operation Increment.

Then we will detail the following points:

- The data-binding possibilities,
- the principles for using the other advanced features,
- the test on different platforms,
- the documentation,
- The licence,
- The advantages and limits of the implementation.

In the last subsection, we summarize each implementation presentation and compare them with comparative tables.

4.1. General principles

Development of DPWS devices with the existing implementations is quite similar to the development of simple Web Service, since a DPWS service is foremost a Web Service. Most often the implementation of a Web Service is based on stubs and skeletons that encapsulate the invocation mechanisms while the developer focuses on business code to perform an operation or react to a notification.

In the rest of this document, we use the following definitions to refer to the different parts of a DPWS service implementation:

- **Interface**: a contract between the client and the server about the provided operations and their signature.
- **Handler**: a function that processes received events at the client side.

- **Callback**: a function that processes an invoked operation of the Web Service at the server side. The signature of this callback is designed according to the signature of the corresponding operation exposed in the interface.
- **Data-binding**: the code that allows marshalling data which is exploitable in handlers and callbacks into XML format. The data-binding perform also the inverse operation.
- **Stub**: a proxy at the client side that provides methods exposed in the interface in order to invoke transparently a remote operation on the server. Thus this proxy is in charge of marshalling requests and unmarshalling responses. The stub can also trigger a handler when an event is received.
- **Skeleton**: a proxy at the server side that dispatches requests to the appropriate callback. Thus this proxy is in charge of unmarshalling requests and marshalling responses.

Stubs and skeletons can be generated according to a WSDL specification in the case of a top-down approach (e.g. implementation from a specified interface).Then the generated data-binding corresponds to the XML schema used as types in the input and output message of the operations. These are re-generated each time the WSDL is modified. Stubs and skeleton can also be generic for all signatures of the operations. This is often the case of a bottom-up approach which consists of generating the interface from an implementation.

The next step to develop a Web service is the implementation of the callbacks and handlers. These callback and handler are actually an implementation of the interface that is shared by the server and the client. The input and output of these operations are encapsulated by using the data-binding provided by the skeleton and the stub. The data-bindings can be classified into 2 categories:

- **Bean-like data-binding** that maps XML Schema to Bean-like classes or structures (e.g. classes that represents XML Schema types and exposes as attributes or fields the sub-elements if it is a complex type). The body soap is entirely parsed and translated into an object that allows to manage inputs and outputs as it was a local method call. The advantage of this data-binding is the ease of use but its disadvantage is the need for a generation phase from a WSDL specification. So this data-binding cannot be dynamic.
- **Document-like data-binding** that leaves the responsibility to the developer to interpret or build the input or output messages. The encapsulation of the message can be more or less sophisticated: from the string that represent the soap body, to the DOM-like structure to naviguate among the XML element.

In the next subsections, we will detail the used data-bindings for each DPWS implementations. The client main program is also implemented by using its data-binding to prepare the invocation arguments and interpret the response. Generally the data-binding are shared by the stub and skeleton. Thus the preparation of the input parameters by the client is similar to the preparation of the reply by the callback, and inversely for the interpretation of the input by the callback and the interpretation of the output by the client. This is also valid for the notification and solicit-response exchange patterns. So in the presentation of the different existing DPWS stacks, we only present in detail the callback implementation.

Often client and server need to initialize a stack that manages the stubs and skeletons, and allows configuring the running environment. This point will be particularly detailed in each DPWS stack presentation

Simple Web Services can also provide advanced features such as security on the HTTP messages thanks to WS-Security, or attachments in the SOAP messages thanks to MTOM.

The development of DPWS services differs of Web Services by adding a service that represents a device and hosts other Web Services. In addition device must be dynamically discovered and exposes metadata about them and their hosted services. As a consequence the development of DPWS devices and services add the management of a discovery and metadata exchange conversation between the client and the server. Since the DPWS stack are almost the only implementation to support WS-Eventing as a main feature, we can consider that subscription management conversation is new in DPWS comparing to the other Web Services stacks.

Discovery and metadata exchange implies for the developers that he has to provide device and hosted service descriptions for the discovery (e.g. id, types and scopes) and for the metadata exchange (e.g. information about the model and the device). The figure 39 illustrates a generic way to implements DPWS services at the server-side and at the client-side. Elements in purple represent the libraries provided in the DPWS stack. Elements in grey represent code that are generated from a contract such as a WSDL file. Elements in yellow represent artefacts that are manually edited by the developers. In the rest of this document, we will keep this colours code to expose a schema about the principle of use of a given DPWS implementation.

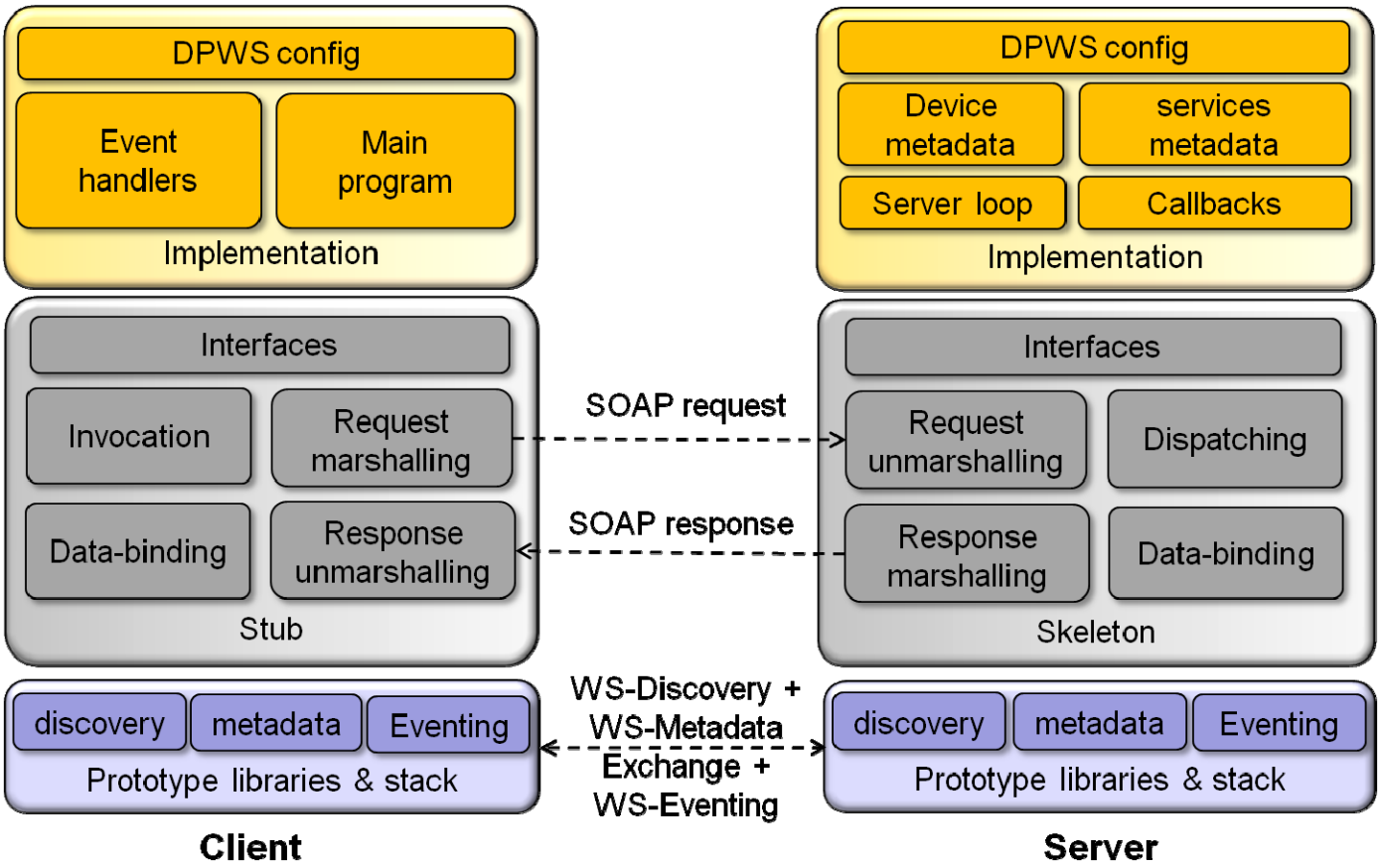

Figure 39: General principles for DPWS stacks

The coding of the server main program, when it is explicitly given by the developer, always follows the same pattern:

1. Initializing the dpws stack, possibly with the interface (i.e. eth0), IP address, and binding to be used.
2. Instantiating a device and services

3. Setting the discovery metadata for the device and services data such as uuid, the types and scopes.
4. Setting the metadata of the device for the parts ThisModel and ThisDevice.
5. Add the services as hosted services to the device
6. Starting the device. From this point, the server loops to accepts requests and process them. This loop can be already managed by the DPWS stack or remaining the responsibility of the developer.
7. The server has also to provide a possibility to leave the loop, in order to shutdown the device. The developer can for example monitor when the user hit ‘q’ and then break the loop.

The coding of the client, when developed, follows always this pattern:

1. Initializing the dpws stack, possibly with the interface and binding to be used.
2. Preparing and sending a probe request thanks to the provided dpws API. After this step, if there are device that respond to the probe, the developer obtains a list of device references that are generally encapsulated either in a class or a structure.
3. Preparing and sending, for each interesting device reference, a request to obtain the device metadata thanks to the provided dpws API. After this step, the developer obtains the device metadata, generally encapsulated in class or a structure that exposes the ThisModel, ThisDevice and RelationShip fields.
4. Obtaining the service proxy from the metadata in order to invoke its operations.

The objects or structures (Figure 40) to manipulate devices, services and metadata at the server side and client side are generally designed in a similar way in the different DPWS prototypes. ThisModel, ThisDevice, Service are generally shared between the server and the client, and represent transparently the metadata.

At the server side, these classes are accessible from the Device and HostedService classes. So the developer has to instantiate these classes and put ThisModel and ThisDevice with the correct values.

At the client side, these classes are accessible from the DeviceProxy, DPWSMetadata and ServiceProxy classes. The proxy for the device is a result of the operation that orders a probe from the stack and can be used to get the metadata and the proxy for the hosted services. The proxy for the hosted service encapsulates the invocation to the provided operations.

For implementations that are based on the C language, these classes are replaced by C structures. Then the manipulation of these structures is quite similar with the objects. Structures are often allocated before the call to the DPWS stack, and then put as an argument by reference to the call to the stack. The stack fills the structure as a result, while the stack returns an integer code that indicates if the operation succeeded or failed.

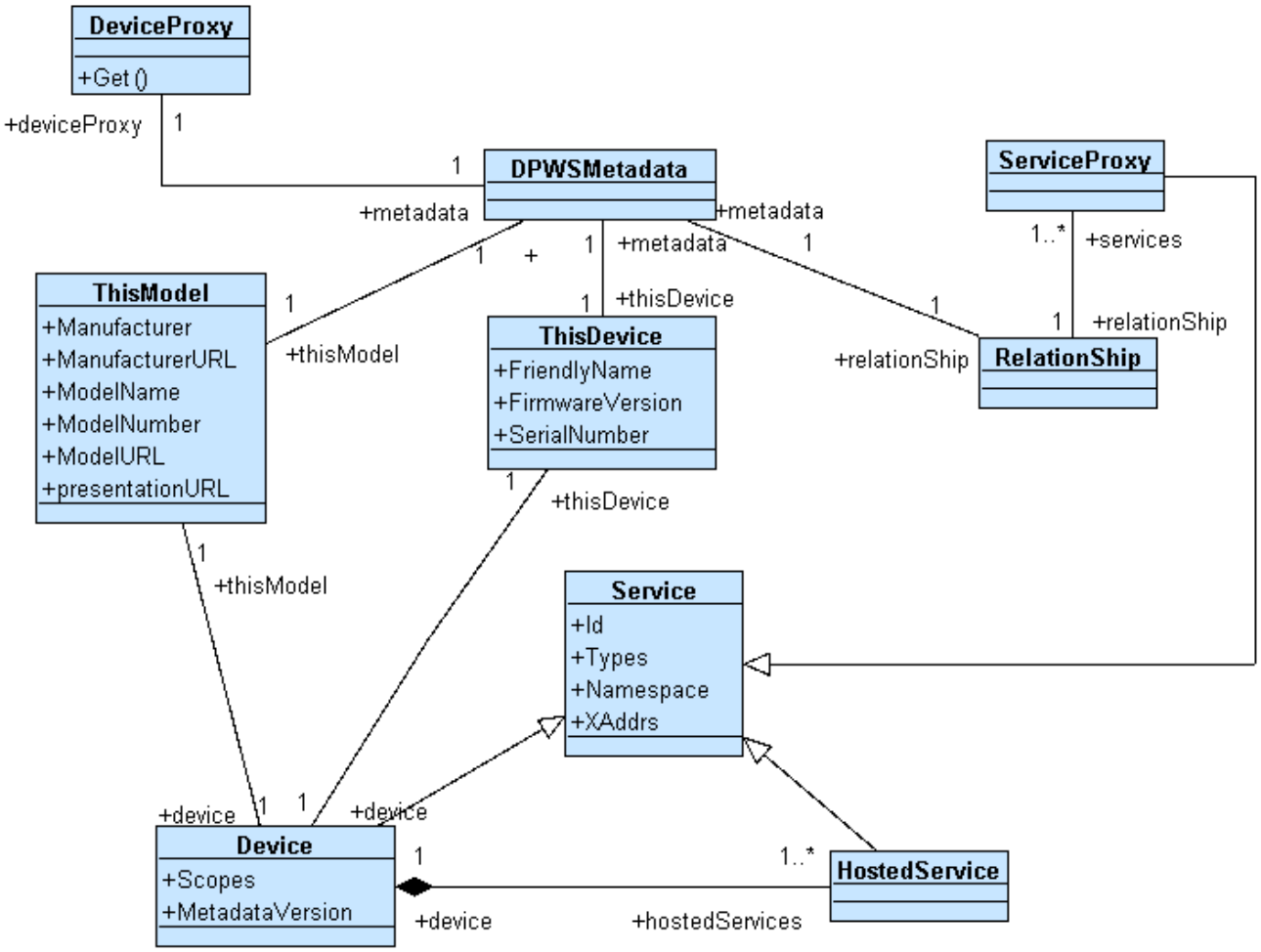

Figure 40: Object model to set and access DPWS metadata

4.2. .NET frameworks & libraries

The .NET frameworks and libraries developed by Microsoft provide support to DPWS and target different platforms from the tiny resource constrained devices to the personal computer or servers:

- Windows Vista/7 and Windows Server 2008 support natively the use of DPWS into the windows applications. These applications can be developed thanks to the **windows SDK** corresponding to the targeted Windows version. Each Windows SDK embeds the **Web Service on Device API (WSDAPI) [50]**, a C++ implementation of DPWS.
- The **.NET Framework** targets personal and enterprise computers with windows XP, Vista and 7, while the **.NET Compact Framework** targets PDA and other rich Smartphone with Windows CE and Windows Mobile. The .NET framework is composed of a virtual machine called CLR, and a set of libraries including **WSDAPI for .NET** [48], an adaptation of the WSDAPI with managed code like C# or Visual Basic. DPWS is supported by these frameworks since the .NET framework 3.5 and .NET Compact Framework 3.5. Unfortunately, although they can be downloaded for free of charge, they do not provide directly the WSDAPI. The API is available in the Windows Embedded Standard 7, a componentized version of Windows 7 that enables to build custom iso image of this OS.

- **.NET micro framework** [49] is a framework developed in order to enable programming high level applications on tiny devices (e.g. resources limited devices as sensors, development platform boards, etc). A very small memory footprint is obtained by eliminating the OS layer and deploying the framework and the applications as a unique firmware on the device. The figure 41 provides an overview of the .NET micro framework architecture. The runtime component layer contains the .NET micro framework CLR and the drivers that abstract the hardware. The Class Library layer contains all libraries usable by the programmer to build his application. The .NET micro framework represents a recast of the .NET frameworks by selecting the essentials packages and reorganizing them. That is why the DPWS is supported, since the .NET micro framework 2.5, with a different API than WSDAPI for .NET.

The table of the figure 42 compares the use of the different frameworks.

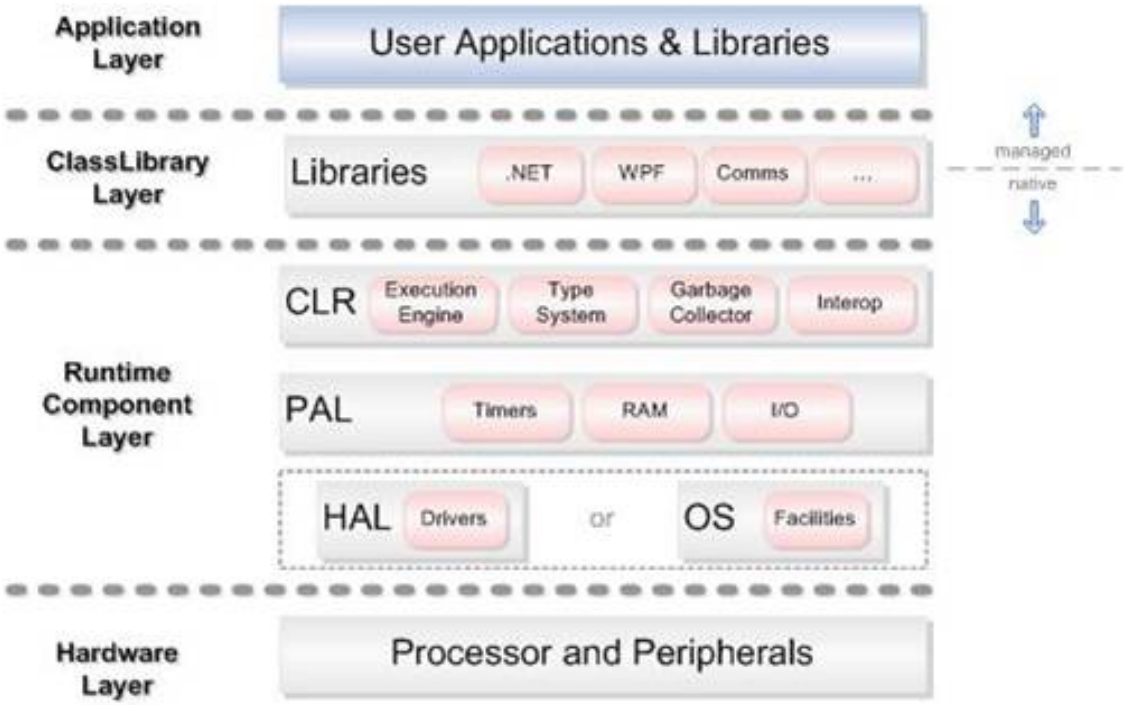

Figure 41: .NET micro-framework architecture overview

	.NET Micro Framework	.NET Compact framework	.NET Framework	WSDAPI
Targeted products	Sensor, robotic, sideshow	GPS, PDA, Automotive	Computers, servers workstations	Computers with Windows Vista/7
Memory footprint	250 – 500 Kb	More than 300 Kb	More than 40 Mb	?
CPU	ARM7, ARM9, Cortex	X86, MIPS, SH4, ARM	X86	X86
Power	Very low power	Low power	Mains power	Mains power

	.NET Micro Framework	.NET Compact framework	.NET Framework	WSDAPI
OS	No OS	Windows CE	Windows XP/Vista/7	Windows Vista/7
Managed vs. native code	Managed via .NET Micro Framework	Managed by .NET Compact framework	Managed by .NET framework	Native

Figure 42: .NET framework target architecture comparison [51]

Principle

The principle of work of these DPWS solutions (see figure 43) is very similar: a top-down approach where stubs and skeletons are generated from the WSDL, while the developer just implements callbacks, handlers and main programs for server and client. In addition of the .NET Micro Framework Web Services data-bindings, stubs and skeleton integrate directly the WSDL description as generated C# code. In the contrary, device and service metadata descriptions are set in the server main program by using the DPWS API.

The main idea of the .NET Micro Framework API, beyond providing object model that encapsulates the manipulable objects related to DPWS or Web Services such as bindings or metadata, is the implementation of default devices, services and clients. Theses default implementations corresponds to the behaviour of the server and client according to the DPWS specification. Thus it is easy to inherit these default implementations and add them the customized parts of our DPWS applications.

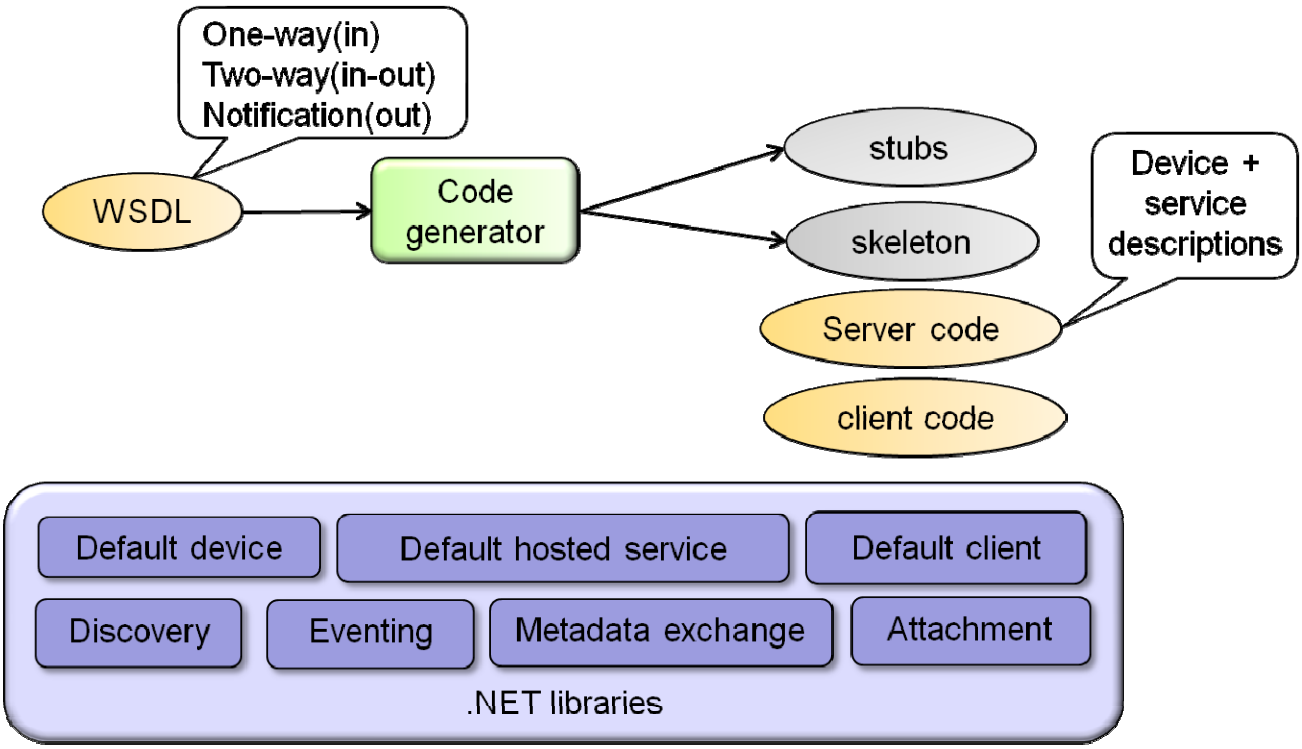

Figure 43: DPWS with .NET frameworks principle

To implements the callback, the developer inherit the interface generated in the stub and skeleton. Then their implementation is as transparent as the call of a local function. The generated data-binding translates the used types into classes and primitives C# types according to the Bean-like data-binding type. In the figure 43, we implement the *Increment* callback by inheriting the *ICounter* interface which has been generated. *TwoWayReq* and *TwoWayResp* are the data-binding class to encapsulate the input step value and the output counter value.

```
namespace schemas.example.org.Counter
{
   class CounterImpl : ICounter
   {
        private int current;

        public TwoWayResp increment(TwoWayReq req)
        {
             int step = req.value;

             current += step;

             TwoWayResp result = new TwoWayResp();
             result.value = current;
             return result;
        }
   }
}
```

Figure 43: Callback with .NET Micro Framework

Each type in the XML Schema is mapped with a class in the C# code to express arguments of the callback:

- Primitive types are mapped with built-in C# structures and types. For example, the *int* primitive XML schema type is mapped with the Int32 structure when it used for the type of the root element. Otherwise this type is mapped with the *int* built-in type.
- Complex types are mapped with new generated class that provides public attributes for each sub-element of the type. This data-binding supports sequence, choice, restricted types with enumerations.

In addition of this class that encapsulates the types of the arguments of the callback, new class are generated for marshalling and unmarshalling. The *readObject* and *writeObject* methods adapt the interpretation of the attributes with a sequence or a choice for example. The generated data-bindings are common for the client and the server and are used in the same manner than in the callback implementation. So to invoke the *Increment* operation implemented in the figure 43, the developer has to instantiate a *TwoWayReq* object and fills the *value* attribute.

To develop main programs for servers and clients, .NET provides default implementations of services and clients to manage the discovery, metadata exchange and eventing conversations. The Device structure represents the device with its default implementation, DPWS metadata, and contains the Host service and the hosted services. So firstly, the developer prepares the Host service and hosted services by inheriting the

DPWSHostedService class. The host service consists simply to store the WS-Discovery metadata like the type and scope. This is done in the figure 44. The hosted services are actually automatically generated from the WSDL, and contain the WSDL information. They are then parameterized with the class that implements the callback (figure 43). Secondly, in the server main program (figure 45), after initializing the Device structure with a binding and a protocol version, the DPWS metadata and host service are set, and the hosted service is added. Finally the device is started with its Start method. Note that the server loop is automatically managed by the .NET Micro Framework DPWS stack after the starting of the device, until its stopping.

```
public CounterDeviceHost: DpwsHostedService {
  public SimpleDeviceHost(ProtocolVersion v): base(v){
     ServiceNamespace = new WsXmlNamespace("cnt", "http://..."));
     ServiceID = "urn:uuid:abcdef";
     ServiceTypeName = "CounterDeviceType";
  }
}
```

Figure 44: Device discovery properties in .NET Micro Framework

```
/* initialization */
string guid = "urn:uuid:123456";
ProtocolVersion version new ProtocolVersion10();
Device.initialize(new WS2007HttpBinding(
                    new HttpTransportBindingConfig(guid, 4320),
                    version);

/* sets the metadata */
Device.ThisModel.Manufacturer = "Microsoft Corporation";
Device.ThisModel.ManufacturerUrl = "http://www.microsoft.com/";
Device.ThisModel.ModelName = "CounterDev";
Device.ThisModel.ModelNumber = "1.0";
Device.ThisModel.ModelUrl = "http://www.microsoft.com";
Device.ThisModel.PresentationUrl = "http://www.microsoft.com";

Device.ThisDevice.FriendlyName = "Counter Device";
Device.ThisDevice.FirmwareVersion = "alpha";
Device.ThisDevice.SerialNumber = "123";

Device.Host = new CounterDeviceHost();
Device.HostedService.Add(new Counter(new CounterImpl()));

/* starting */
ServerBindingContext ctx = new ServerBindingContext(version);
Device.start(ctx);

…

Device.stop();
```

Figure 45: main program at server side with .NET Micro Framework

The client side reuses the file implementing the marshalling and service interface, in addition of the generated service proxy. The client code consists of

- Instantiate the service proxy generated in the stub. This proxy provides default clients for discovery, metadata exchange. So they encapsulate the management for the discovery and metadata exchange conversations. In addition, it encapsulates the invocation to the provided operations.
- Invoking from this proxy the features provided by the different default client of the DPWS stack. The figure 46 shows the invocation of these default clients from the proxy p, to send a probe request, and get metadata request. Finally the proxy is used to invoke the Increment operation. Note that since the service transport address is discovered at runtime, you have to set the Endpoint property of the proxy class with the address of the appropriate hosted service before invoking the provided operations.

```
/* proxy instantiation */
ProtocolVersion version = new ProtocolVersion10();
WS2007HttpBinding binding = new WS2007HttpBinding(
        new HttpTransportBindingConfig("urn:uuid:123456", 15000));

CounterClientProxy p = new CounterClientProxy(binding, version);

/* probes */
DpwsServiceTypes typeProbes = new DpwsServiceTypes();
typeProbes.Add(new DpwsServiceType("CounterDeviceType",
                                   "http://...",));
DpwsServiceDescription desc = p.DiscoveryClient.Probe(typeProbes);

/* gets metadata */
String deviceTransportAddr = "http://192.168.1.42:1234/...";
DpwsMetadata metadata = p.MexClient.Get(deviceTransportAddr);

Debug.Print("friendly name:" + metadata.ThisDevice.FriendlyName);

/* invoke service */
DpwsMexService hosted = metadata.Relationship.HostedServices[0];
p.EndpointAddress = hosted.EndpointRefs[0].Address.AbsoluteUri;
TwoWayReq req = new TwoWayReq();
req.Value = 1;
TwoWayResp result = p.Increment(req);
```

Figure 46: Client probe with .NET Micro Framework

The difference between these frameworks relies on these points:

- The class naming. For example, in the .NET Micro Framework, the discovery client is named DiscoveryClient while it is named ClientDiscovery in the WSDAPI for .NET. In C++ WSDAPI, the naming difference are much different: the discovery client is called

IWSDiscoveryProvider and the Probe and Resolve features are respectively ensured by the SearchById method and SearchByType method.

- In the .NET Micro Framework, the DPWS libraries are organized in the packages Dpws.Device and Dpws.device.Service (assembly MFDpwsDevice.dll) to deal with devices and hosted services, and in the package Dpws.Client (assembly MFDpwsClient.dll) to dela with DPWS clients. The WSDAPI for .NET is contained in the package Microsoft.Web.Service.Dpws and offers, as in the .NET micro framework, classes to build default DPWS devices, hosted services and clients. These classes manage all the DPWS conversation for WS-Discovery, WS-MetadataExchange and WS-Eventing.
- Since WSDAPI and .NET Micro framework are based on quite different languages, the code may be quite different as they use different syntaxic structures. For example the C++ WSDAPI is based on C structures while C# API exploits the Get and Set properties.
- In the contrary of WSDAPI for .NET and .NET Micro framework, an IDL file is generated as a contract to specify input and output while C# annotation allows adding these information directly on the generated code.
- In the .NET Micro Framework, the code generator is MFSvcUtil.exe in the Tools subdirectory of .NET Micro Framework install directory. For the WSDAPI, the code generator is the wsdcodegen.exe in the Bin subdirectory of the SDK install directory. For the WSDAPI for .NET, the code generator is DpwsCodeGen.exe. These code generator take simply in input the WSDL file and generate

Advanced features

- **Hello/Bye listening**: the Hello and Bye message can be listened by setting the events specified in the device proxy, thanks to the super class DpwsClient. These events are like placeholders to specify a handler that react to the receiving of these messages. Since C# has functional programming features, the function to be called as a handler can be directly used as a parameter to build such events.
- **Eventing**: In the same model than the discovery and the metadata exchange, the proxy provides a default client for eventing, that allows to manage a subscription with subscribe, renew or unsubscribe. The notification and solicit-response handler are designed with the same way than the callback for one-way and two-way operations: the stub is generated and contains the needed data-binding and the C# interface that specifies the interface of the handler. The signature of theses handler and the data-binding are similar to those for the server callbacks.
- **MTOM**: To use the MTOM features, the client needs to have more control about the way of sending the messages. So the developer should use the API with a lower level by building himself a channel and needed headers and invoking the marshaller and unmarshaller to build message body and process response body. In the server-side, there is no change with classical invocations.
- **Managed discovery**: The default discovery proxy provides another operation, DirectProbe, to perform a managed probe instead of a broadcast probe. This operation works in the same way that the probe method, but the developer specifies an endpoint.

Target platforms

The .NET DPWS stack was tested with the .NET Micro Framework and Visual Studio Express on a Windows 7 plateform. Visual Studio provides freely an emulator that enables to test execution of DPWS server and client on emulated devices.

DPWS with .NET should work on any platform that with a Windows system and a .NET framework, or devices that are designed to work with .NET Micro framework. In the contrary, this stack is not adapted to Java platforms. So this platform does not work on Android or SunSpot.

Documentation

The .NET framework and the WSDAPI are quite well documented to start and develop DPWS services on computers or devices. They provide:

- A general user manual for WSDAPI [53], WSDAPI for .NET [48] and .NET Micro Framework [57]. The documentation for .NET Micro Framework is not specific to DPWS but includes all the aspects of this framework (i.e. customizing emulator, ...)
- tutorials for getting started with WSDAPI for .NET [55] and .NET Micro Framework [58]
- Complete API references for WSDAPI [52], WSDAPI for .NET [54] and .NET Micro Framework [56].
- Samples showing the discovery, metadata exchange, eventing and MTOM features.

Licence

The Windows SDK, including the C++ WSDAPI, is distributed freely under a Microsoft Software licence that allows a non-commercial use into software that add significant features. The redistributed software must display the unchanged Microsoft licence. However the sources are not open.

Since Microsoft is concerned with spreading the .NET Micro Framework for the embedded programming on devices, .NET Micro Framework has an open-source licence in order to facilitate the portage of the framework by the manufacturers on their devices. .NET Micro Framework is thus distributed with the Apache Licence 2.0 [61]. This licence authorize to modify, use in another software and redistribute possibly with another licence. However the Apache licence requires that each modification on a file must be specified and all provided NOTICE file must be present in the new distribution.

Advantages

- Allows to develop DPWS services on devices that work with Windows Mobile, such as Pocket PC, Windows Phone.
- Allows to deploy DPWS services on resources constrained devices that are designed for .NET Micro Framework, such as Tahoe III, FEZ development boards with sensor and Wifi connections.
- The API is intuitive, well designed, and well documented.

Limits

Since in the context of Choreos, we are more interested by an open source solution on resource constrained devices, we have only tested the .NET Micro Framework version. In conclusion of our experiences, this framework provides all the features for discovery, metadata exchange, eventing and attachment. However it remains some gaps:

- The namespaces of the Web Services standards used in DPWS are not respected, in particularly when switching to DPWS 1.1. The WS-Discovery namespace satisfies the DPWS specification, but in contrary the namespace of DPWS, WS-Eventing and WS-Transfert stay always at the namespace of the DPWS 1.0 version. Thus we consider that .NET Micro Framework support only the DPWS 1.0 version.
- .NET Micro Framework allows using the DPWS client with a Discovery Proxy thanks to the DirectProbe and DirectResolve methods in DiscoveryClient object. But the framework does not enable the implementation of discovery proxy.
- The security is not provided directly by the different DPWS stack proposed by Microsoft. As a consequence, the security features are ensured by the other libraries embedded in the framework: with the WSDAPI, we have to use the secure channels of the HTTP server API [59] while .NET Compact Framework provides API to establish secure communications [60]. In the other side .NET Micro Framework does not provide the security features.
- .NET Micro Framework does not support IPv6. Thus IPv6 for DPWS is not available.
- Although the generation tools is designed to take account of complex types when invoking an operation, they are not well supported: the generated code produce malformed SOAP message because the namespaces and their prefix are not properly managed.

4.3. µDPWS

µDPWS represents the most lightweight implementation of DPWS in term of resource consumption. It has been designed especially for constrained networked embedded systems and wireless sensor networks. In one hand, µDPWS works on top of Contiki [62], a very lightweight and open source operating system for tiny devices like embedded systems on microcontrollers or sensor network motes. Effectively Contiki needs only few kilobytes of code and few hundred bytes of RAM. In another hand, the native code implementing the DPWS stack, devices and services are contained in 20 Kb and consumes 3Kb of RAM. This very small memory footprint makes this stack interesting to target very constrained devices, while other implementation needs too memory.

µDPWS is a contribution of the WS4D initiative and was implemented within a German master thesis in 2010. So this project is in its early versions and is yet a prototype. Just the more important features are supported in this implementation. Despite of his lack of features and the presence of many bugs, the next versions may be promising.

Principle

The development of devices and hosted services consists of building an image or executable that contains the contiki OS adapted for DPWS, the DPWS stack with its customized configuration, and the device and services implementation. For this purpose, µDPWS is distributed with 2 sources directories:

- The **Contiki directory** with its sources that are modified to support multicast for IPv4 and adding some IP configurations.

- The **µDPWS directory** that is organized as following:
    - The *core* subdirectory contains the **DPWS stack sources**. It provides routines to help building and sending DPWS messages, and automate the device behaviour.
    - The *platform* subdirectory contains the **DPWS platform configuration** in order to set the ID of the platform and the port that is used to exchange messages.
    - The *device* subdirectory contains the **device configuration and implementation** in order to set the device description, the access information and possibly implementing a custom behaviour.
    - The *services* subdirectory contains the **service configuration and implementation** in order to provide WSDL description and the implementation of the provided operations.

These sources allow implementing DPWS server on devices on top of the Contiki OS. The implementation of DPWS server is made almost exclusively by hand to interpret input and outputs arguments of the callback or linking the different platform, device and service sub-projects in the corresponding makefiles. However all the basic behaviour of the server for the discovery and the metadata exchange is already implemented in the dpws core subproject. So the developer needs to focus only on the customized part of the DPWS application. In addition, metadata description for the device and the service are described in plain text files for facility, but are integrated into C code in the platform subproject thanks to the file2C tool. The figure 47 summarizes the principle of use of µDPWS.

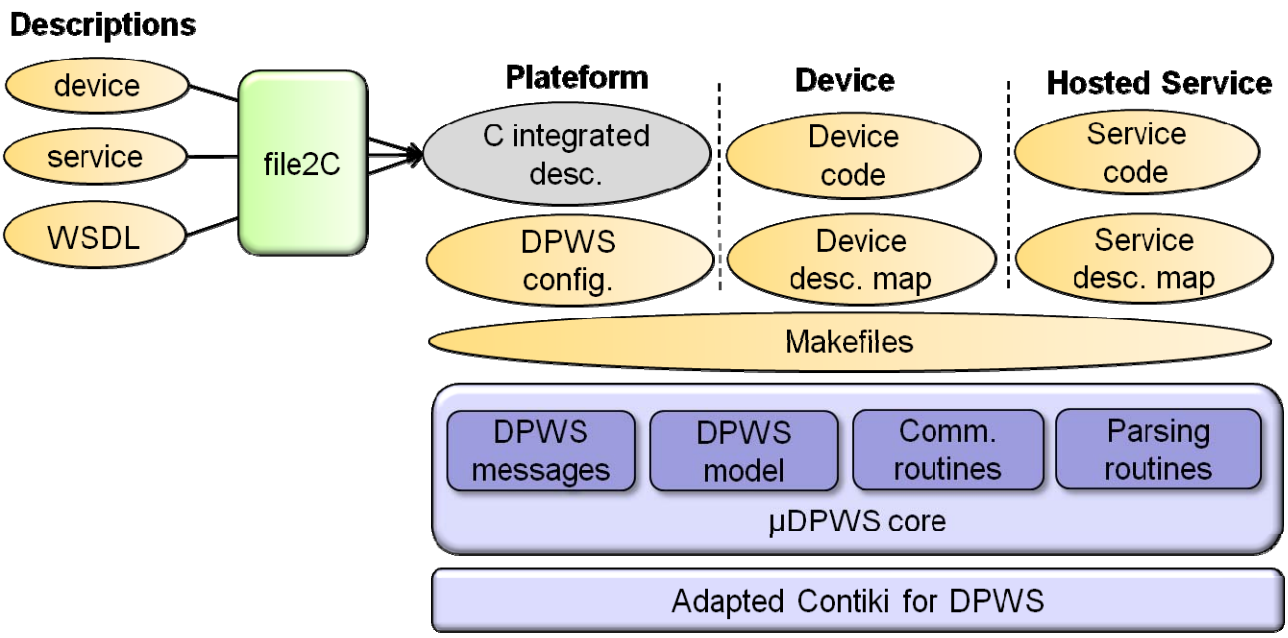

Figure 47: DPWS with µDPWS principle

The implementation of the callback remains very low-level: the parsing of the soap message remains the responsibility of the developer, with the help of XML functions provided by the µDPWS stack. So the arguments of the callback correspond to the header and body of the SOAP request, and the response message, represented by a structure. The last argument of the callback is a structure that represents the µDPWS stack. It enables for example the conversion of the values into the required format to put it in a SOAP response message. Conversely they can be converted in the appropriate type with the usual C functions (i.e.

atoi(...)). The figure 48 is an example of callback implemented with µDPWS for the increment operation. The function *xml_next_tag and xml_cont_as_str* and the constants XML_TAG_TYPE_EMPTY are used to parse the body by obtaining the next tag from the current position, and gets the value of the current tag. The response message is then prepared by generating the header with the µDPWS routine *dpws_gen_header*, and by generating the body by adding XML elements and values thanks to the µDPWS routines *sendm_add_end_rom* and *sendm_add_end_ram*.

```
int current = 0;

int cnt_increment_callback(struct soap_header_s *sh, char* body,
int body_len, struct sendm_s *sm, struct stack_s *stack) {

    int err;
    char* body_end = body + body_len;
    char* p = body;
    uint8_t tag_type;

    /* Parse the body */
    /* go in Body element */
    if (xml_next_tag(p, body_end, NULL, &tag_type, …, &p))
        return DPWS_ERR_PARSE;

    /* go in TwoWayReq element: Error if empty */
    if (xml_next_tag(p, body_end, NULL, &tag_type, …, &p))
        return DPWS_ERR_PARSE;

    /* go in Value element: Error if empty */
    if (xml_next_tag(p, body_end, NULL, &tag_type, …, &p))
        return DPWS_ERR_PARSE;


    /* here must be the value string */
    if (tag_type == XML_TAG_TYPE_EMPTY || (p = xml_cont_as_str(p,
body_end)) == NULL) {
        /*No target temp in there*/
        return DPWS_ERR_PARSE;
    } else {
        step = atoi(p);
        current += step;
    }

    int err;
    err = dpws_gen_header(SH_RELATESTO | SH_TO , …, sm, sh, NULL);

    char * resultValue = stack_pop(stack, 4);
    itoa(current, resultValue);

    sendm_add_end_rom(sm, cnt_TwoWayResp, cnt_TwoWayResp_len);
    sendm_add_end_rom(sm, cnt_value, cnt_value_len);
    sendm_add_end_ram(sm, resultValue, resultValue_len);
    sendm_add_end_rom(sm, cnt_xvalue, cnt_xvalue_len);
    sendm_add_end_rom(sm, cnt_xTwoWayResp, cnt_TwoWayResp_len);
```

```
    DPWS_FINISH_SOAP(sm);

   return 0;
}
```

Figure 48: a two-way operation implementation with µDPWS

This implementation of DPWS does not really provide data-binding, since the callback arguments just encapsulate the input and output SOAP messages. Thus all the XML schema and Web Service styles are supported by µDPWS, provided that the developer adapts the input parsing and output building according to the WSDL.

To implement the server side, the programmer intervenes mainly on the device and services directories while the other directories are rather sources and headers to be included. The development process may be summarized with the following unordered steps:

- **Configuration of the DPWS platform** by editing the udpws-config.h for setting the port used for DPWS, the initial UUID or debug levels. The figure 49 is an excerpt of the udpws-config.h file, where the properties are defined C macros.
- The **description of the device and the services** by putting each field value in a plain text file, which has an arbitrary name. These files are placed in a gen subdirectory of the concerned section (e.g. device or services). In order to link them to the expected description fields, the developer has to write a structure that contains the name of the files. The figure 50 shows the C source for a device and illustrates the link between the implementation and the plain text metadata. The structure *udpws_acd_temp_device* provides the file names (highlighted in green) for the device UUID, the *ThisDevice*, *ThisModel*, *MetadataVersion* sections. Then the description fields are integrated into the C code in the gen directory of the DPWS platform thanks to the file2C tool. This transformation adds a facility to the programmer by setting the properties in a readable format and avoids the load of a text file during the execution. Instead the fields become C variables that can be reused in the device and service implementations. In a similar way, the *udpws_counter_service* structure is edited to receive hosted service metadata. In addition, a field of this structure references the callback function for each provided operation.
- The **specification of the relationship between the device and the services**. The code in the figure 50 declares 2 structures representing hosted services: *udpws_aircon_service* and *udpws_temp_service*. These structures are then referenced in the structure that maps the device and text file description. In order to take account of these services, their makefile have to be included in the device makefile.
- The **implementation of the device,** optionally the **device behaviour** may be completed by a thread that performs a custom processing. The device thread is controlled by macros to begin, end or wake up the thread. Since the counter device is not a good example, we provide an example with a air conditioner device that regulate the temperature to reach a target temperature. The code in the figure 51 is a device implementation of the air conditioner device.

Note that the developer does not need to implement the server loop to accept request and dispatch them; this is automatically managed by the µDPWS stack. So the developer worries only about the business code for the callbacks and device additional behaviour, and about

the description of the device and hosted services. The sources are then compiled thanks to the makefiles that take account each part of the sources. The main makefile to launch all the compilation is in the platform implementation.

```
#define UDPWS_INITIAL_UUID        "urn:uuid:123456"
#define UDPWS_PORT                4320
```

Figure 49: µDPWS plateform configuration

```
extern const struct dpws_device_s udpws_cnt_counter_device;
#define UDPWS_DEVICE                &udpws_cnt_counter_device

extern const struct hosted_service_s udpws_counter_service;

#define DEVICE_SERVICE_COUNT    1
#define DEVICE_SERVICES    {&udpws_counter_service}

const struct dpws_device_s udpws_cnt_counter_device = {
        cnt_counter_UUID,
        cnt_counter_UUID_len,
        cnt_counter_ThisDevice,
        cnt_counter_ThisDevice_len,
        cnt_counter_ThisModel,
        cnt_counter_ThisModel_len,
        cnt_counter_MetadataVersion,
        cnt_counter_MetadataVersion_len,
        cnt_counter_ServiceId,
        cnt_counter_ServiceId_len,
        &counter_process,
        DEVICE_SERVICE_COUNT,
        DEVICE_SERVICES
};
```

Figure 50: µDPWS device configuration and hosting service

```
extern int ac_target_temp;
extern int ac_current_temp;

PROCESS(aircon_process, "AirConditioner_process");

static struct etimer periodic_timer;

PROCESS_THREAD(aircon_process, ev, data)
{
    PROCESS_BEGIN();
    etimer_set(&periodic_timer, 3*CLOCK_SECOND);

    while(1) {
      PROCESS_YIELD();
      if (etimer_expired(&periodic_timer)) {
        if (ac_target_temp < ac_current_temp){
```

```
                ac_current_temp--;
                DBG_PRINT_INFO("cooling");
            }
            else (ac_target_temp > ac_current_temp){
                ac_current_temp++;
                DBG_PRINT_INFO("heating");
            }
            etimer_restart(&periodic_timer);
        }
    }
    PROCESS_END();
}
```

Figure 51: µDPWS device implementation

Target Plateform

µDPWS was tested on

- linux Ubuntu and the gcc toolchain. The device must be started with the superuser rights (with the sudo command)
- Windows XP with MinGW and cygwin. Cygwin is required to deal with the provided makefile. Since sudo does not exist in cygwin, cygwin has to be started with the administrative rights. µDPWS is not dependant of the Windows version as it can be complied with the MinGW toolchain.

µDPWS could work on Android thanks the Native Development Kit (NDK) provided by Android. However µDPWS is not a good choice for this platform because the NDK is less advanced than the SDK based on Java, and applications are more complex to design with C/C++ and NDK than with Java. µDPWS implies useless difficulties whereas the Android platforms are often powerful and work well with a heavier DPWS stack.

Documentation

Since µDPWS is only in its early version, there is not much documentation:

- A google-code site [63] that exposes a concise presentation of µDPWS and few tutorials to getting started with µDPWS on linux or 2 models of device sensors.
- Few samples are provided in the µDPWS distribution, with a client able to send DPWS messages in order to test the µDPWS features.

Licence

µDPWS is available under the New BSD Licence [64] that allows redistributing the source or the binary form with or without modifications, provided that the licence is unchanged. Thus this project is open-source and free of charge but have to be redistributed with the New BSD licence.

Advantages

The main advantage of this implementation is its very small memory footprint that enables this stack to target very resource-constrained devices.

Limits

However µDPWS presents many limits, because of its prototype state or the goal of the project:

- µDPWS supports only few main features. The eventing, attachments, security or adavanced features for the discovery are not yet supported.
- The µDPWS stack enables only to implement server since the team take the hypothesis that tiny devices contains only server parts and there exists other stack to build a client on computers. Nevertheless the stack is compatible only with clients that support DPWS 1.1 and deliver HTTP messages built in a specified form. The problem is the high flexibility of the HTTP protocols with the form with the header: some client can specify the size of the HTTP message in the header while other put this information at the end the message. The µDPWS is not compatible with messages where the size message is specified in the header.
- There is a bug when receiving a Resolve message: a internal error occurs.

4.4. JMEDS

JMEDS is a Java-based implementation of DPWS developed by Materna – Information & Communications in the context of the WS4D initiative. The version 2 of this project started in 2009 and is regurlarly updated to bring new features. Now JMEDS represents the most advanced and active work of the WS4D initiative concerning the DPWS implementation. This framework enables almost all the existing DPWS features. In addition this stack proposes very usefull additional features as the interpretation on the fly of WSDL descriptions, in order to enable more flexibility when discovering devices. Since the JMEDS framework targets Java SE and Java CLDC platforms, DPWS service based on JMEDS can be deployed on a huge variety of machine, from personal computers with Windows or Linux to phones supporting java, android phones or SunSpot, a sensor that embeds Java. Thanks to its usability, runtime flexibility and its portability to a wide range of platforms, JMEDS is one of the most interesting implementation of DPWS.

Principle

The principle of use of JMEDS is a bottom-up approach where the WSDL specification is automatically generated at runtime to be sent to the client. So the developer has just to write a Java code using the provided high-level DPWS library in order to develop a device with hosted services and clients. The JMEDS API provides all the necessary objects to encapsulate and manipulate low-level elements for each DPWS features and Web Service. In addition, JMEDS provides for facility default implementations for devices, hosted services and clients. So the developer has just to inherits these classes and redefine the methods that corresponds to the behaviour to be customized.

The server and client code are completed by a plain text file configuration to set properties about the DPWS configuration and device and services description without recompiling the application. Figure 52 illustrates the DPWS development process with JMEDS.

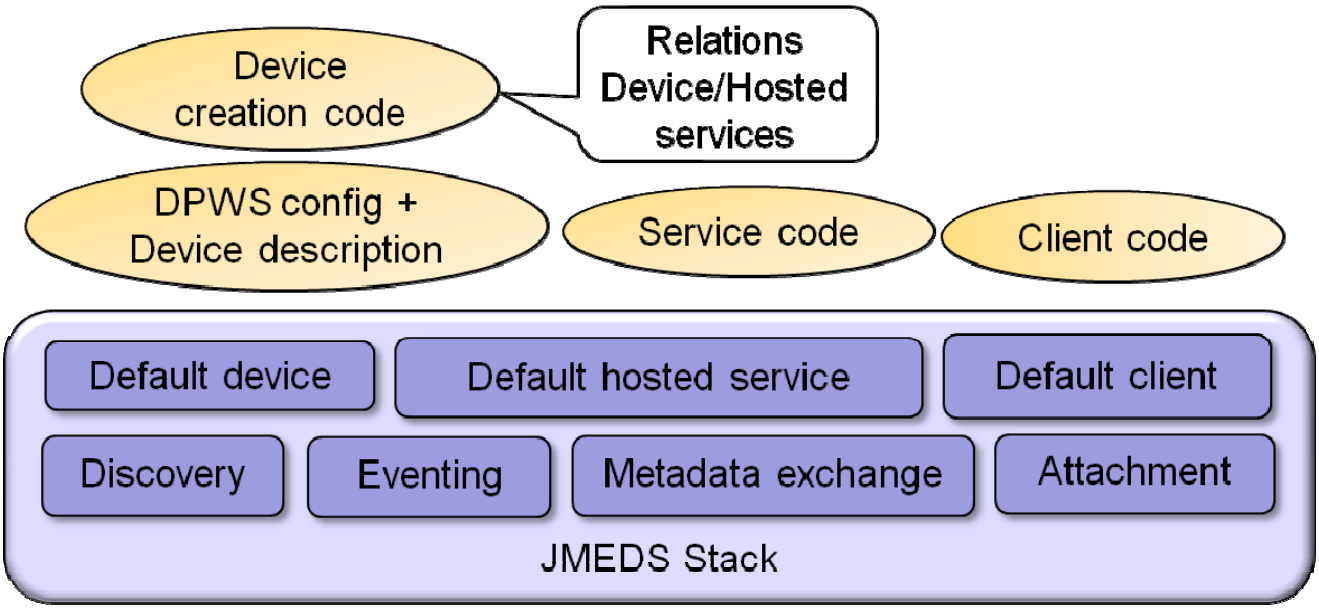

Figure 52: DPWS with JMEDS principle

The JMEDS API represents Web Services and their constituting elements with a class model: there are for example classes to encapsulate a service, an operation, parameter values and XML Schema declarations. So the implementation of the callback is performed by refining the Operation class and implementing the *invoke* abstract method. This method takes in input and output a *ParameterValue*, a generic encapsulation of the arguments that are received and returned by the service. A *ParameterValue* can be considered as an instantiation of the declared XML schema. Thus the business logic consists too in interpreting the inputs arguments and building the output values. However JMEDS provides facilities to interpret the arguments: the *setString* and *getString* methods of *ParameterUtil* allow using a path like XPath expressions to reach the right XML element and get or set its value. This facility is particularly interesting when using complex types or several arguments wrapped in a complex type. The figure 53 exposes the declaration and the callback of the *Increment* operation. The *getString* and *setString* methods points on the *TwoWayReq* and *TwoWayResp* XML elements. So the XPATH expression used to get the set is “value” since it is the direct sub-element.

```
public class TestService extends DefaultService{
   public TestService(int configurationId){
        super(i);
        DPWSFramework.start(new String[]{“example.properties”});
        this.addOperation(new TwoWayOperation());
   }
}

class TwoWayOperation extends Operation{

   protected int current = 0;

   TwoWayOperation(){
        super(“Increment”, new QName(…));
```

```
        int t = SchemaUtil.TYPE_INT;
        Type tInt = SchemaUtil.getSchemaType(t);

        Element val = new Element(new QName("Value",…), tInt);

        ComplexType tVal = new ComplexType(new QName("val",…),
                                   ComplexType.CONTAINER_SEQUENCE);
        tVal.addElement(val);

        Element in = new Element(new QName("TwoWayReq",…), tVal);
        this.setInput(in);

        Element out = new Element(new QName("TwoWayResp",…, tVal);
        this.setOutput(out);
    }

    public ParameterValue invoke(ParameterValue inputs){
        String val = ParameterUtil.getString(inputs, "Value");
        int step = Integer.parseInt(val) ;

        this.current += step ;
        String result = Integer.toString(this.current);

        ParameterValue returnVal = this.createOutputValue();
        ParameterUtil.setString(returnVal, "Value", result) ;
        return returnVal;
    }
}
```

Figure 53: Service implementation with JMEDS

There are 2 ways for the declaration of a service and their operations:

- With the JMEDS API, thanks in one hand to the classes that encapsulates the XML Schema constituting element, and in the other hand thanks to the class that enables to declare services and operations. In the figure 53, the class *TestService* is a service that declares the operation *Increment*. This operation is represented by the class *TwoWayOperation* where the constructor add the XML element *TwoWayReq* and *TwoWayResp* respectively as input and output of the operation. During the metadata exchange, the WSDL of this service is automatically generated from this Java model in order to send it to the client. The client is able to interpret the WSDL and reconstitute the Java model at the client side.
- By reading and interpreting the WSDL at runtime. Instead of inheriting the Operation class to declare a new Web Service operation, the developer defines the service by setting the URL of the WSDL. The JMEDS will create an internal object model of the service with its operation, input and output. The developer can then retrieve the defined operation from the service, and set its callback by building a new *InvokeDelegate* that provides the *invoke* method. The figure 54 shows the same declaration than in the figure 53, but with the WSDL interpretation method. *Service.define* reads firstly the WSDL file *description.wsdl*. The *Increment* operation stub is retrieved with the *getOperation* method and the action name of the specified,

as specified in the WSDL. The callback is finally defined with the *setDelegate* method of the operation stub

Both the 2 methods has data-binding that deal with simple types, complex types such as sequence, choice or enumeration. In the case of a declaration with the JMEDS API, the class ComplexType encapsulates complex type and can be parameterized to represent a sequence or a choice for example. The manipulation of such types for the parameter values in input and output of the callback are transparent by using the XPATH expressions.

```
DefaultService service = new DefaultService();
service.define(new URI("local:/…/description.wsdl"));
OperationStub op = service.getOperation("http//…/Increment");

op.setDelegate(new InvokeDelegate() {
   public ParameterValue invoke(Operation op, ParameterValue in){
        …
   }
});
```

Figure 54: Service implementation with JMEDS

ParameterValue object are used in the same manner to implement callback or clients. So to invoke the increment operation declared in the figure 53, the developer builds a parameterValue that encapsulates the inputs and sets the value with the *setString* method and the path "*value*". Inversely the parameterValue that correspond to the output message is interpreted with the *getString* method, as in the callback implementation in the figure 53.

Developing the DPWS server part with JMEDS is quite simple: firstly the developer provides the DPWS properties such as the IP address and port to use for this server, and the device's metadata. These properties are expressed thanks to a plain text file organized in sections and subsections. In these sections, properties have the form of a key associated with value. Among the possible properties, we have to specify the handler class that will interpret them, the port for the HTTP binding used by the device and the hosted services and the device metadata. The figure 55 shows the section that configures the DPWS framework for using the IP address 192.168.2.100 on port 4320. The figure 56 shows the sections for the DPWS metadata specification. Note that the binding of the device and the hosted service are linked to the bindings of the figure 51 by their identifier. With the same idea, the hosted service is linked to the device by the configuration identifier.

```
[Global]
PropertiesHandler=org.ws4d…configuration.GlobalPropertieshandler

[Bindings]
PropertiesHandler=org.ws4d.java.configuration.BindingProperties

[[HTTPBindings]]
PropertiesHandler=org.ws4d…HTTPBindingProperties
Address=192.168.2.100
Port=4320

[[[HTTPBinding]]]
```

```
BindingId=1
Port=4321
Suffix=SimpleExampleDevice

[[[HTTPBinding]]]
BindingId=2
Port=4322
Suffix=SimpleService
```

Figure 55: JMEDS DPWS Framework configuration

```
[Devices]
PropertiesHandler=org.ws4d…DevicesPropertiesHandler
MetadataVersion=1
FirmwareVersion=1
ModelNumber=2

[[Device]]
ConfigurationId=1
DeviceSecured=false
DeviceUuid=urn:uuid:1234
MetadataVersion=2
Types=http://…/Device http://.../CounterDeviceType
Scopes=http://www.ws4d.org
Binding=1

ManufacturerUrl=http://dpws.materna.de
ManufacturerName=Materna
ModelName=CounterDev
ModelNumber=3
ModelUrl=http://dpws.materna.de/model
PresentationUrl=http://dpws.materna.de/model/presentation.html

FriendlyName=Counter Device
FirmwareVersion=2
SerialNumber=123

[Services]
PropertiesHandler=org.ws4d…ServicesPropertiesHandler

[[Service]]
ConfiguationId=1
ServiceId=CounterService
Binding=2
```

Figure 56: Device description with JMEDS

The second step is the java implementation of the device, its initialization and starting. Note that before other actions, the server has to start the DPWS framework with as argument the configuration file presented above, in order to set the properties concerning the address, port, and device description. To implement the device, JMEDS already provides a default

implementation that manages the discovery, metadata exchange and eventing conversations. After instantiating, the developer can add to it an instance of the service previously declared, and start it. The figure 57 shows the main program for a server, that instantiate the device and a hosted service, and then start it. The server loop is automatically managed by the JMEDS stack, so the developer has just take care of start and stop the loop.

```
DPWSFramework.start(new String[]{"exemple.properties"});

DefaultDevice device = new DefaultDevice(1);
LocalService service = new TestService(1);
device.addService(service);

try{
   device.start();
   …
   Device.stop();
catch(IOException ex){
   e.printStackTrace();
}
```

Figure 57: Server main program with JMEDS

Implementing the client is very simple also: firstly the DPWS framework have to be initialized with the configuration file indicating the address and port to be used. Secondly JMEDS provides also a default implementation of a client where methods to react to hello messages, events or research results can be overridden. In addition JMEDS encapsulates all the elements that represent devices and services references. The figure 58 shows a class for a simple client that search a device of type *CounterDeviceType*, and then invokes the operation *Increment* of the hosted service *CounterService*. After initializing the DPWS framework, the client constructor launches a search with a device type filter. When a device is found, the method *deviceFound* is called with the reference of the device. This reference contains all the metadata of the device, and allows obtaining the hosted services references. The service and device are accessible from theses references. Finally the *invokeService* method performs the *Increment* operation invocation by obtaining the operation and building the parameter values in the same way than the service.

```
public class TestClient extends DefaultClient{
   TestClient(){
        DPWSFramework.start(new String[]{"example.properties"});

        /* search a device */
        QName devT = new QName("SimpleDevice", "http://...");
        SearchParameter params = new SearchParameter();
        params.setDeviceTypes(new QnameSet(devT));
        this.searchDevice(params);
   }

   /* when a device is found */
```

```
    public void deviceFound(DeviceRefence devRef, …){
         Iterator serv = devRef.getDevice().getServiceReferences();
         while(serv.hasNext()){
              ServiceReference s = (ServiceReference) serv.next();
              if(s.getServiceId == "CounterService"){
                   Service service = (Service) s.getService();
                   invokeService(service);
              }
         }
    }

    public void invokeService(Service s){
         Operation op =
                s.getAnyOperation(new QName(…), "Increment");
         ParameterValue inVal = op.createInputValue();
         ParameterUtil.setString(inVal, null, Double.toString(5));

         ParameterValue outVal = op.invoke(inVal);

         Double result = ParameterUtil.getString(outVal, null);
    }
}
```

Figure 58: Client implementation with JMEDS

Advanced Features

- **Hello/Bye listening:** The default client used in the preceding example for sending probe request provides two methods that handle the receiving of Hello or Bye messages. The default behaviour is the display of a message. So the developer can overwrite these methods to react to a Hello message by launch request to get metadata for example.
- **Eventing:** The obtained stub for a service provides always methods to manage subscription to events. So after the discovery and getting the service stubs, the developer has just to use them. The subscription to an event implies that the developer build an event sink, i.e. a configuration of the binding and the port for receiving the events. Then to process the notifications and solicit-response, the eventReceived method of the default client has to be overwritten to process them. The same methods process all the possible events. Thus the kind of event has to be checked each time.
- **MTOM:** JMEDS leads with attachment in a transparent way, as any data. Instead of using the *getString* and *setString* methods of the ParameterUtil class, the developer uses the methods *getAttachment* and *setAttachment*. In addition JMEDS support streaming by providing an InputStreamReader with the obtained attachment for the client.
- **Security:** The security with JMEDS is based on the Java security framework that provides an implementation of SSL. The Java security framework enables to put certificates in keystores and truststore. The keystore is used by the server to send the first certificate, while the truststore is used by the client to verify the received certificate. This exchange of certificates is entirely encapsulated in the Java API, when the developer builds SSL sockets. But a first step for using the security is the generation of the keystore and truststore with the keytool, provided by the JDK. In

addition, these stores hava to be referenced in the configuration file of JMEDS in a new section concerning the security.

Target platforms

JMEDS was tested on the following platforms:

- **Java SE**: the stack works well after modifying the source to solve a bug during the initialization of the stack.
- **Android 2.1, 2.2 and 2.3**: the stack works well on Android for the server and client side. The latest version of JMEDS is not completely compatible with Android 2.2 because it uses some methods that are available only since Android 2.3. These methods are used to verify that a found interface is up and support multicast. However on android phones or tablets, we often have one WiFi interface and we can easily make the assumption that the user verifies that the WiFi is up, and we can make the assumption that this interface supports multicast. After a modification for avoiding these tests, the latest version works well on Android 2.2.
- **SunSPOT**: JMEDS released his stack for J2ME and provided in addition an extension to work with the SunSPOT sensors. Effectively, SunSPOT sensors communicate via infrared instead of WiFi or Ethernet. The idea was to allow the SunSPOT to join the network through another SunSPOT that we call “base” and is connected to a computer by USB. Consequently JMEDS released a program for the sunspot base that perform the conversion between the infrared and networked communication. This extension works well for the discovery conversation, but unfortunately there are problems with the other conversation based on HTTP. Effectively the UDP messages for discovery are sent with only one packet, while the HTTP messages are chunked. Problems occurs when the sunspot base want reassemble the packet to send the sunspot dpws server.

Documentation

Although the advanced state of this implementation, the documentation is very sparse:

- a section of the ws4d website [67] presents quickly the architecture and the supported features
- samples are available on the JMEDS sourceforge [68] to demonstrate the use of discovery, eventing and invocation.
- The Javadoc can be generated from the source and gives some indications about how to use some features of JMEDS and helps to understand the structure of the code. However this javadoc is quite incomplete.

Fortunately this limit is remedied by an API that is quite intuitive and easy to use.

Licence

JMEDS is distributed under the Eclipse Public Licence (EPL) [69]. The EPL was originally used for the distribution of the Eclipse EDI and is now extended to any open source programs. Like other open source licence, the EPL allows getting freely a program and its sources, and then modifying and redistributing. All programs that are derived from these sources must be distributed with the EPL licence. However in the contrary of the GNU Public Licence, this licence allows to distribute a commercial application that uses libraries which are available with a EPL licence.

Advantages

- Supports a lot of DPWS features
- Targets several editions of Java. Thus DPWS services based on JMEDS can be deployed on computers, Java and Android phones or other tiny devices such as SunSpot.
- Interpretation of the WSDL at runtime. This allows to develop DPWS browser for example where we can discover DPWS services and invoke them without knowledge of them before the execution.
- The API provided by this DPWS stack is high-level and easy-to-use.

Limits

- The new version (2.0 beta 4) solves issues of initialization when we start, then stop and restart the DPWS stack. But this new version works after a modification on the initialisation on the list of discovery bindings.
- The last version of JMEDS does not work directly on android 2.2 because the stack uses methods that are available only on android 2.3. These methods are used to verify that a network interface is up and supports the multicast. The issue can be avoided by modifying the JMEDS sources and replace the call to these methods by true. As a consequence, when we use DPWS on Android 2.2, we take the hypothesis that the first network interface is always up and support multicast.
- Poor documentation
- WS4D announce that JMEDS support managed discovery with a discovery proxy but there is no available example with a discovery proxy.

4.5. WS4D-Axis2

WS4D-Axis2 [71] is another DPWS implementation by the WS4D initiative and is based on the well known Axis 2 Web Service engine and framework. The Axis engine is designed with a core that provides the basic features to dispatch messages or deploy Web Services. It accepts a set of modules for the required extensions to manage the Web Services. The idea of WS4D-Axis2 was to add new modules to Axis 2 to support DPWS, in particular SOAP-over-UDP, WS-Discovery and WS-MetadataExchange. As there exists modules to support eventing (SAVAN [70]), security (Apache Rampart [72]) and attachments (MTOM/SwA [73]), these features are reused instead of redeveloping again. This approach is interesting thanks to its capabilities of extensions with other Axis2 based projects.

Principle

The principle of use of this stack is very simple: the hosted services are already deployed services on the Axis 2 engine. Then the developer has just to deploy a new service that represents the device and is linked to the hosted services. This device is simply implemented by an Axis2 archive containing a XML file with DPWS configuration and metadata information. The figure 59 illustrates the development of DPWS devices with WS4D-Axis2.

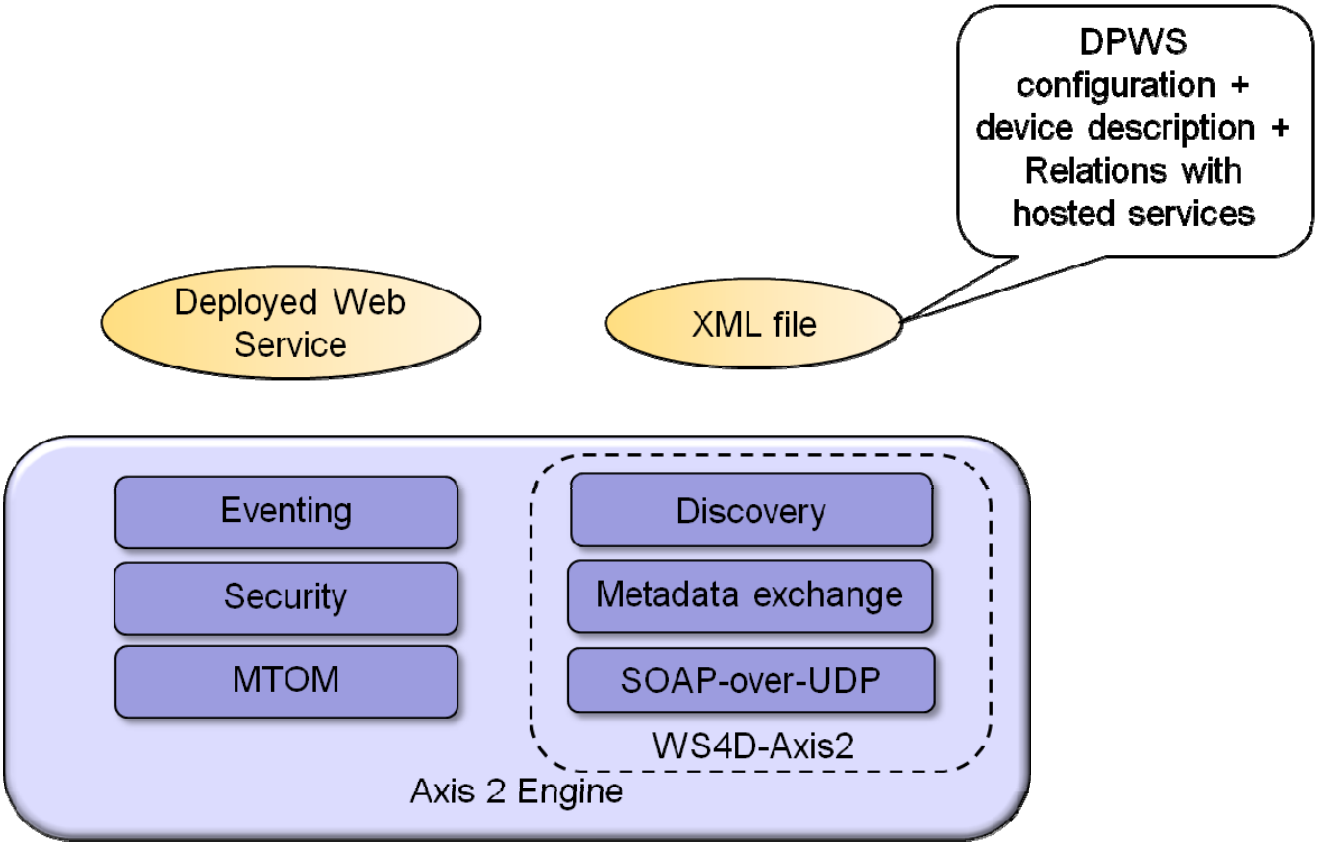

Figure 59: DPWS with Axis 2 principle

Axis2 proposes several ways to create a Web Services with a top-down or a bottom-up approach. The POJO and AXIOM methods enable to deploy a Web Service from a Java class and to generate automatically the corresponding WSDL at runtime. POJO and AXIOM methods address simple and complex types. They present the input and output arguments in a different way: POJO allows to write the callback as if it was a local call, while AXIOM presents input and output with the AXIs Object Model, a DOM-like structure that is based on the StAX API. So AXIOM gives the developer the initiative to interpret input and output by manipulating the XML structure.

In the contrary, the ADB, XMLBeans and JiBX frameworks allows to create a Web Service by generating Java code from a WSDL. According to the chosen framework, the XML schema specification is more or less well supported and the input/output parameters are presented as a DOM model, or Beans (e.g. classes representing the XML element according to their name and have accessors to manipulate direct children) that are mapped to the XML schema. In the same idea, Axis2 supports also the creation of Web Services thanks to JAXB. ADB is provided by Axis 2, while JiBX, XMLBeans and JAXB are well known frameworks for accessing XML by binding it to java types.

The device implementation consists of an Axis2 archive composed with an XML file that presents configuration parameters and the device description with the DPWS format described in the section 3.1. The figure 60 is an example of XML file to deploy a device that hosts the Version service, a Web Service that deployed by default at the installation of Axis2. In the first part of this file are the usual Axis2 parameters: useOriginalwsdl and modifyUserWSDLPortAddress. The second part contains the information needed for the

discovery phase of the DPWS conversation, namely the UUID, the types and scopes of the devices. Finally the third part contains the metadada information with the ThisModel, ThisDevice and Relationship metadata exchange dialects.

```
<serviceGroup name="sampledevice">
  <service name="sampledevice">
    <parameter name="useOriginalwsdl">false</parameter>
    <parameter name="modifyUserWSDLPortAddress">true</parameter>

  <!-- -->
  <!-- Discovery definitions -->
  <!-- -->
    <parameter name="Discovery" auto="true ">
      <enableAutoDiscovery>true</enableAutoDiscovery>
      <UUID>urn:uuid:123456</UUID>
      <types>
        <type ns="http://..." prefix="wsdp" name="Device" />
        <type ns="http://..." prefix="bs"
                                    name="CounterDeviceType" />
      </types>
      <scopes>
        <scope uri="http://.../adhoc" />
        <scope uri="socom:non-public-area2" />
      </scopes>
      <UDPTransportSenders>
        <sender name="soapudp" />
      </UDPTransportSenders>
    </parameter>

  <!-- -->
  <!-- Metadata definitions -->
  <!-- -->
    <parameter name="DPWSDevice">
      <ThisDevice …>
        …
      </ThisDevice>
      <ThisModel …>
        …
      </ThisModel>
      <Relationship>
        <Host …>
             …
        </Host>
        <Hosted name="Version" …>
          <EndpointReference>
          http://localhost:8080/axis2/services/Version
          </EndpointReference>
          …
        </Hosted>
      </Relationship>
   </parameter>
```

```
  </service>
</serviceGroup>
```

Figure 60: DPWS configuration and device description with DPWS-Axis2

Advanced features

- **Security:** The security aspects are insured by Rampart, an Axis 2 module.
- **MTOM:** the MTOM features are insured by MTOM/Swa a module for Axis 2.

Target platform

WS4D-Axis2 works on any platform that supports Java SE and Axis 2. Since Axis 2 requires an important amount of memory, WS4D-Axis 2 targets principally workstations and server instead of resource constrained devices.

Licence

As this project is closed to Axis2, from the Apache foundation, WS4D-Axis2 is released under the same licence: the Apache Licence 2.0 [61]. This licence authorizes to modify, use in software and redistribute possibly with another licence. However the Apache licence requires that each modification on a file must be specified and all provided NOTICE file must be present in the new distribution.

Documentation

The documentation about this project is almost nonexistent: there are just the trac wiki [71] to present very quickly the project and a documentation page [74] describing how to install the modules and proposes a sample to test the DPWS stack.

Advantages

Finally Axis 2 has a good tooling and is well integrated in the development cycle: eclipse-plug-ins are available to deal with Axis2 and provide graphical editor for WSDL, wizards to perform both top-down and bottom-up approaches using the Axis2 command-line tools, and support for integrating the Axis2 server into the IDE. In addition, development steps for Axis2 Web Services are integrated in Maven, a software project management and comprehension tool. Once deployed, the new created Web Service can use the features of the installed modules such as eventing, security,...

- Many tools exist to develop with Axis2.
- The modularity of Axis2. Many project exists to add new features and we can easily develop new modules to complete them according to our needs.

Limits

- The WS4D-Axis2 stack provides only supports for DPWS server (e.g. receiving probe and resolve request and answer with matches, sending metadata). The discovery and metadata exchange modules does not enable to send discovery requests.

- Axis2 is not designed to work on resource-constrained devices and has a too big memory footprint, although we can lighten it by removing some unused modules and features.
- This project seems to be abandoned since the end of 2008, so we cannot hope a new release that adds new features.
- Although there exists a project to support WS-Eventing on java-based Axis2, SAVAN/Java, eventing features are not provided by this implementation. Effectively, the SAVAN/Java project download is unavailable.
- The Rampart module is not integrated to the discovery module. So this implementation of DPWS does not provide WS-Discovery digital signatures.

4.6. WS4D-gSOAP

WS4D-gSOAP [75] is a C implementation developed by the WS4D initiative, and is based on another well known Web Service Toolkit: gSOAP [76]. gSOAP is an interesting framework as it provides good performances [77][78] and is very lightweight on top of an OS. So, this implementation can work on resource-constrained devices such as development boards such as FoxBoard [81] or NetBurner [82].

Principle

The principle of use of this DPWS implementation is a top-down approach where the device and the service are respectively described in DPWS XML format and in WSDL. Code is then generated from these descriptions thanks to command line tools:

- The **device code generator** generates, from the DPWS device description, the code that sets the device properties as C constants and setup the device and the hosted service at the server start.
- The interfaces and data binding for input and output parameters are generated from the WSDL. This generation uses an intermediary format: the gSOAP file format. It is generated by the **WSDL to gSOAP wsdl2h** tool and then used by the **service code generator soap2cpp** tool to obtain the C code. Unfortunately, since these tools are originally provided by the gSOAP distribution, they are not WS4D-gSOAP aware and do not add the required *include* in the gSOAP file. In addition, wsdl2h does not support the message exchange pattern for eventing (e.g. notification and solicit-response). Thus the developer have to edit manually the gSOAP file to add the required rectifications. So the code generation process cannot be automatized.

Finally the development of the callbacks and main programs for the server and the client remains the responsibility of the developer by using the WS4D-gSOAP API. This API provides the needed functions to initialize the DPWS stack, intiative the discovery and metadata exchanges, or prepare SOAP requests and responses. The figure 56 illustrates the development process with WS4D-gSOAP.

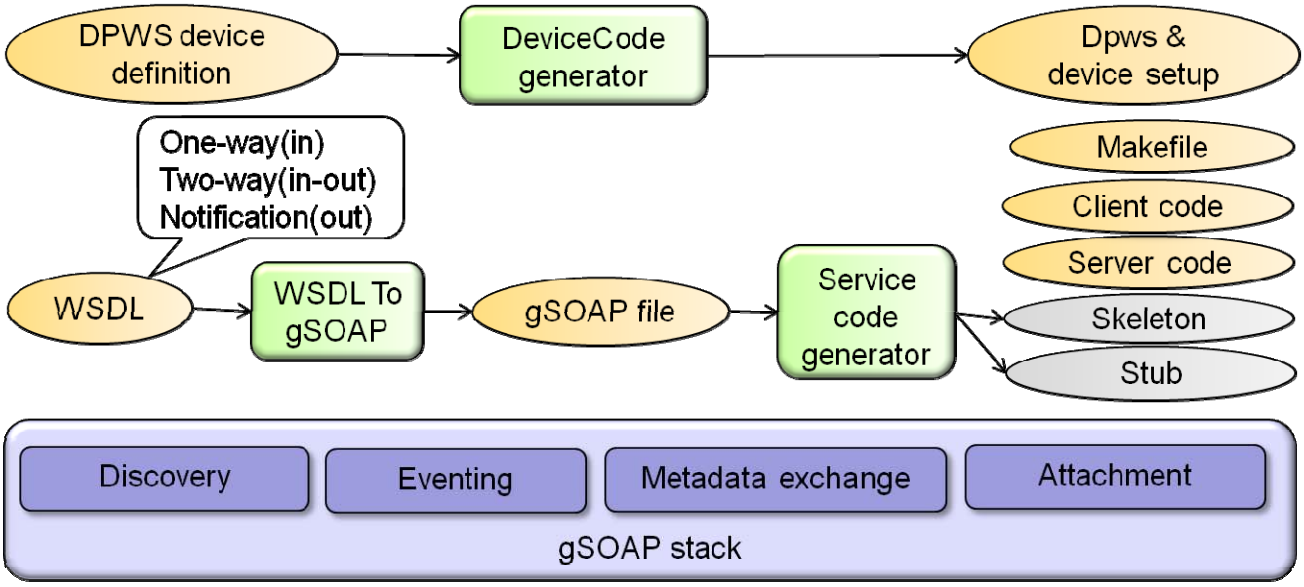

Figure 61: DPWS with WS4D-gSOAP principle

After the generation of the stubs and skeleton, we can find the declaration of the callback in the header file that corresponds to the gSOAP file.The callback takes in input the structures generated for the data-binding that represent the input values and output values. Simple types are mapped with C primitives types while complex types are mapped with C structures and XML enumeration are mapped with C enum. While the structures represent the root element, they allow a direct access to the represented sub-element. After getting the inputs, perform the business process, the callback consists of fill the structure that correspond to the output message, and generate the response header. The figure 62 illustrate that with the implementation of the *Increment* operation.

```
int current = 23;

int __cnt1__Increment(struct soap *soap,
                      struct cnt1__TwoWayReqType* cnt1__TwoWayReq,
                      struct cnt1__TwoWayRespType* cnt1__TwoWayResp)
{
   /* process request message */
   int step = cnt1__TwoWayReq->value;

   current += step;

   /* fill response message */
   cnt1__TwoWayResp->value = current;

   /* create response header */
   return dpws_header_gen_response (…);
}
```

Figure 62: Operation implementation with WS4D-gSOAP

The server main program code follows the pattern introduced in the general principle subsection. The figure 63 is an example of device that provides the Counter service. The DPWS stack is initialized with the functions soap_init, soap_set_namespace and dpws_init in

order to initialize the structures service and device with a specified interface and namespace. The DPWS definition of the device, specified above with a DPWS XML file, is used to generate C source code that embeds the DPWS definition. So this generated code provides the functions *cnt_setup_Hosting*, *cnt_setupCounter*, *cnt_setMetadata* that allows setting the metadata for the device and the hosted services. The WSDL description is added with the same principle: in addition of the invocation and process purposes, the stubs and skeletons embeds the WSDL description in the *cnt_set_wsdl* function.

Then the device is started with the dpws_activate_hosting_service function. The WS4D-gSOAP implementation does not manage automatically the server loop, but provides only function to accept messages (*dpws_maccept* function) and dispatch to the appropriate callback (*dpws_mserve* function). So the developer has to explicitly program the server loop. The code in the figure 58 is a simplified version of main server program: error during the dpws initialization have to be checked with the result of the different called function, and the termination of the program to quit the server loop is the responsibility of the developer by using signals.

```
struct soap service;
struct dpws_s device;

int main(int argc, char * argv){
   const char * interface = “192.168.2.100”;
   const char * uuid = “urn:uuid:123456”;

   /* prepares scope */
   const char * scope = “http://ws4d.gsoap.example/”;
   struct ws4d_list_node scopelist;
   ws4d_alloc_list alist;
   WS4D_ALLOCLIST_INIT(&alist);
   ws4d_sl_init(&scopelist);
   ws4d_sl_add(&scopelist, scope, &alist);

   /* initializes dpws, device and services */
   soap_init(&service);
   soap_set_namespace(&service, cnt1_namespaces);
   dpws_init(&device, interface);
   cnt_setup_HostingService(&device, &service, uuid, 100);
   cnt_setup_Counter(&device, &service, COUNTER_WSDL, 100);

   /* set metadata */
   cnt_set_Metadata(&device);
   cnt_set_wsdl(&device);
   dpws_add_scope(&device, ws4d_sl_tostr(&scopelist, &alist));
   dpws_update_Metadata(&device);
   dpws_activate_hosting_service(&device);
   ws4d_sl_done(&scopelist);
   ws4d_alloclist_done(&alist);

   /* server loop */
   for(;;){
        struct soap * handle = NULL;
        struct soap * soap_set[] = SOAP_HANDLE_SET(&service);
        int (*serve_request[])(struct soap * soap) =
             SOAP_HANDLE_SET(&service);
```

```
        /* waiting for new messages */
        handle = dpws_maccept(&device, 100000, 1, soap_set);

        if(handle){
             printf("processing request from %s:%d",
                   inet_ntoa(handle->peer.sin_addr),
                   ntohs(handle->peer.sin_port));

             /* dispatch message */
             if(dpws_mserve(handle,1, serve_requests)){
                  soap_print_fault(handle, stderr);
             }

             /* clean up soap */
             soap_end(handle);
        }
   }
}
```

Figure 63: Server main program with WS4D-gSOAP

The client is implemented following the common pattern introduced in the general principles section, by using the functions provided by the DPWS API. The code is organized with a C-style, where the different DPWS elements such as the device, services or EPR are maintained in C structures. These structures are then manipulated with functions, where the result is a modifiable input argument, while the function returns an integer to indicate if the operation succeeded. As a consequence, the structure that will contain the result has to be declared before and is passed by reference. Thus the code style is very heavy and verbose. The figure 64 shows a client implementation with WS4D-gSOAP. The initialization of the stack uses the same functions than for the server. Then we can see, in the code that sends a probe request, the devices structure is declared and initialized with the function *ws4d_eprlist_init*. When sending a probe request, the result is put in this structure, while the *dpws_probe* returns the constant *DPWS_OK* if the probe was successful. Finally the hosted service is invoked. To do that, the developer has to set the header fields such as destination and action in the structure representing the client. Then the program can call the stub generated from the WSDL and gSOAP file.

```
struct soap client;
struct dpws_s dpws;

int main(int argc, char * argv){
   const char * interface = "192.168.2.100";
   const char * uuid = "urn:uuid:123456";

   ws4d_alloc_list alist;
   WS4D_ALLOCLIST_INIT(&alist);

   /* initializes dpws, device and services */
   soap_init(&client);
   dpws_init(&client, interface);
   soap_set_namespace(&client, cnt1_namespaces);
```

```
    /* send probe request */
    struct ws4d_abs_eprlist devices;
    ws4d_eprlist_init(&devices, ws4d_eprlist_init, NULL);

    struct ws4d_qnamelist type_list;
    ws4d_qnamelist_init(&type_list);
    ws4d_qnamelist_addstring("http://...", &type_list, &alist);

    int ret = dpws_probe(&dpws, &type_list, NULL, 3000, 1000,
                         NULL, NULL, &devices);

    ws4d_epr * device = NULL;
    ws4d_eprlist_foreach(device, iter, &devices){
        if(ws4d_epr_isvalid(device)){

            /* searches services */
            struct ws4d_abs_eprlist services;
            ws4d_eprlist_init(&services, ws4d_eprlist_init, NULL)

            ws4d_qnamelist service_type_list;
            ws4d_qnamelist_init(&service_type_list);
            ws4d_qnamelist_addstring("http://.../Counter",
                                     &service_type_list, &alist);

            dpws_find_services(&dpws, device, &service_type_list,
                               10000, &services);
            service = ws4d_eprlist_get_first(&services);

            /* invoke service */
            dpws_header_gen_request(&client, NULL,
                               ws4d_epr_get_Addrs(service),
                               "http://.../Increment",
                               NULL, NULL,
                               sizeof(struct SOAP_ENV_Header));

            struct TwoWayReq req;
            req.value = 18;
            struct TwoWayResp resp;
            soap_call__cnt1_Increment(&client,
                               ws4d_epr_get_Addrs(service),
                               NULL, &req, &resp);
        }
    }
}
```

Figure 64: client with WS4D-gSOAP

Advanced features

- **Eventing:** WS-Eventing is not natively supported by WS4D-gSOAP since the WSDL to gSOAP tool does not support the input-output pattern specification for notification and solicit-responses. WS4D-gSOAP proposes a method to bypass this limit: the develop rewrite a reverse version of the WSDL by inverse the input and the output message. Then he generates an inverse gSOAP file, and then the stub and skeleton

code. In this case the goals of the stub and skeleton are inversed: the stub process the request and the skeleton send request that correspond to notification for example. At the server side, eventing is managed with p-threads and routines provided by WS4D-gSOAP to process subscription and check their validity. At the client side, the developer implements the handlers in the same way than for callbacks. He can then use the WS4D-gSOAP routine in the client main program to subscribe, renew, getStatus.

- **Security:** The security support in WS4D-gSOAP is based on openSSL [89].

Target platform

The WS4D-gSOAP works on Windows, Linux and MacOS, with the corresponding toolchains (i.e. gcc, MinGW, Visual studio).

- WS4D-gSOAP is based on the gSOAP framework which is compiled both for Windows, Linux and MacOS.
- WS4D-gSOAP is distributed only as a source tarball. The developer has preliminary to compile the framework for his platform, thanks to CMake, a cross-platform build system.
- The samples and tutorials are based on CMake as build system. So the developer is invited to use CMake to implement DPWS devices and clients.

Documentation

The project is distributed with its sources and a set of samples that allows testing the DPWS features. In the other side, a complete user and developer manual [79] is available and explain how to use the samples. The explanations are organized as a pedagogic tutorial that shows all the steps from the XML file description to the launching of the devices and clients. In addition, the last part of this document is dedicated to a Developer documentation generated thanks to Doxygen.

To complement the user manual, the WS4D-gSOAP wiki provides the installation instructions on Linux with GCC or Windows with Visual Studio.

Licence

WS4D-gSOAP is free software available under GPL/LGPL licence [80]. This licence allows the use, the study of how it works, the modification and the duplication in order to distribute it. This licence is similar to the GPL licence, but in the contrary of it, the LGPL allows redistributing with another licence than LGPL. However this licence does not forbid the software to be paid, if the source code is distributed with a price that is not higher to the cost of distribution.

Advantages

- Small memory footprint and good performance.

Limits

- The code generation steps are not automatized. There is still a lot of manual manipulation to be done. As a consequence the usability is bad.
- Only the very basic features are well supported. In addition, the eventing does not work properly.
- The device does not send hello and bye messages. Thus we cannot do the scenario where the device starts after the client and announce its arrival.
- The API is not intuitive and easy-to-use and the C code –style is verbose and well readable.

4.7. DPWS-Core

DPWS-Core [83] is a C implementation of DPWS implemented by the SOA4D initiative and is based also on the gSOAP framework [76]. This implementation differs from the other C implementation by the number of DPWS supported features and its higher usability compared to the similar framework, WS4D-gSOAP. Effectively, in the contrary of WS4D-gSOAP, the command-line tools of gSOAP have been modified to deal with DPWS and eventing. Thus the code generation is fully automated in this stack.

Principle

The principle of use of this DPWS implementation is a top-down approach where the stubs and skeleton for the hosted services are generated from their WSDL. As in the gSOAP framework, the wsdl2h command-line tool generates a gSOAP file from a WSDL and this output is used by the soap2cpp command-line tool to generate the C data binding and client stubs. Then the developer has just to implement the callbacks and the server and client main program by using the the DPWS-core API providing routine for building messages or initiate discovery and metadata exchanges conversation for example. The device description and DPWS setup can be integrated into the main program in the server code or can be specified in a separate XML file that is interpreted at runtime. The figure 65 summarizes the principle of use of DPWS-Core.

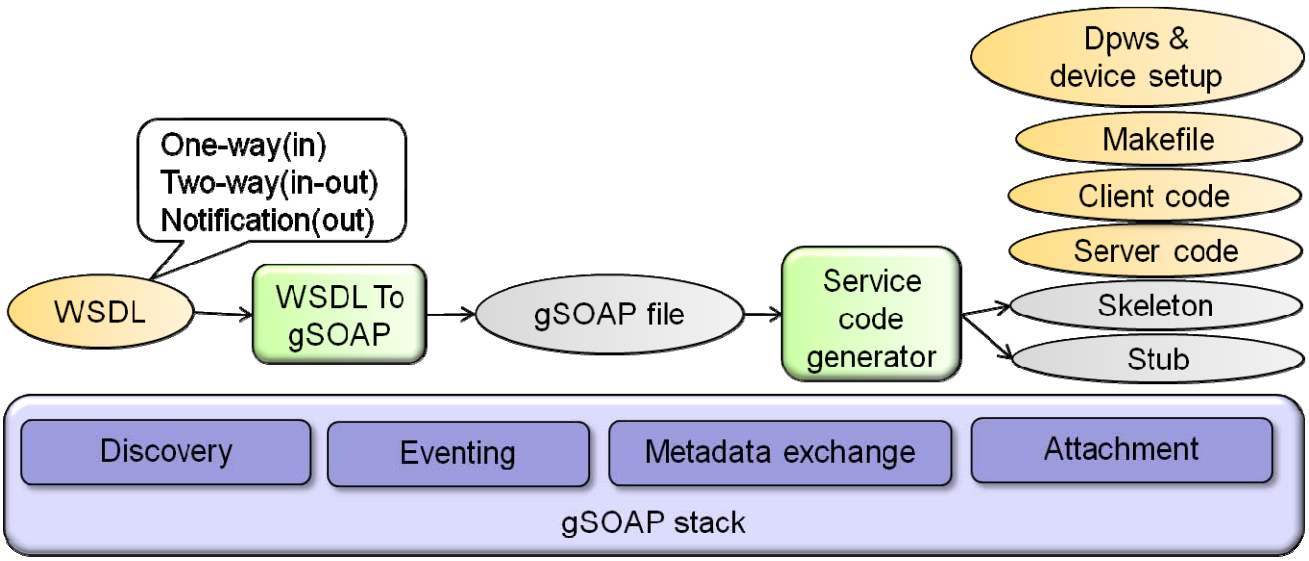

Figure 65: DPWS with DPWS-Core principle

The callback is named with their namespace prefix and operation name. We can find the declaration of the callback in a generated header file in the stub and skeleton. The callback takes in argument a dpws structure that enables to retrieve data that are internal to the device. The next arguments correspond to the inputs and outputs arguments specified in the WSDL. Simple types are mapped with C primitives types, while complex type such as XML schema enumerations are mapped with a C enumerations and XML Schema sequences are mapped as C structures. The figure 66 is an example implementing the *Increment* operation in the Counter device.

```
int __cnt_increment(struct dpws* dpws,
                    struct cnt1__TwoWayReqType* cnt1__TwoWayReq,
                    struct cnt1__TwoWayRespType *cnt1__TwoWayResp)
{
   /* process request message */
   int step = cnt1__TwoWayReq->value;

   int * current = (int *) dpws_get_device_user_data(dpws);
   *current += step;

   /* fill response message */
   cnt1__TwoWayResp->value = *current;

   return DPWS_OK;
}
```

Figure 66: Operation implementation with DPWS-Core

The main program for the server follows the pattern introduced above: it consists of initializing the DPWS stack, configuring the device and the hosted service by giving their description, and performs a loop to wait for requests. The device can be configured by informing a registry that structures the required information about devices and hosted services and is in charge of dispatching messages. The registry is based on an object model similar to the model introduced in the general principles. The registry can be informed by 2 ways:

- With C code, by using the functions dpws_create_* to create instances from the registry object model, and the functions DPWS_SET_Type_ATT on the concerned object with the appropriate constants representing the attributes. The figure 67 shows code that configures the registry object. The figure 68 shows code that describes a counter with its DPWS metadata and its links to the WSDL of its hosted service.
- With a XML configuration file that is read at the runtime. The registry XML element contains all the information described in the registry object model. Then the objects are mapped to XML elements and the attributes are mapped to XML attributes or elements. The figure 69 is a XML configuration file equivalent to the code in the figure 67 and 68.

```
int bootSeq = readintfromfile(BOOT_SEQ_FILE);
writeinttofile(bootSeq + 1, BOOT_SEQ_FILE);
```

```
DPWS_SET_INT_ATT(DC_REGISTRY_HANDLE, DPWS_INT_HTTP_PORT, 4320);
DPWS_SET_INT_ATT(DC_REGISTRY_HANDLE, DPWS_INT_BOOT_SEQ, bootSeq);
DPWS_SET_STR_ATT(DC_REGISTRY_HANDLE, DPWS_STR_BASE_DIR, base_dir);
DPWS_SET_STR_ATT(DC_REGISTRY_HANDLE, DPWS_PTR_HTTP_GET_CALLBACK,
http_get_handler);
```

Figure 67: DPWS configuration with DPWS-Core

```
#define SAMPLE_NS = “http://www.soa4d.org/”

/* service class */
short hServClass = dpws_create_service_class();
struct prefixed_qname PorType = {{SAMPLE_NS, “Counter”},“cnt”}
DPWS_ADD_PTR_ATT(hServClass, DPWS_PTR_PREFIXED_TYPE, &PortType);
struct wsdl_info LightWsdl = {SAMPLE_NS, “/Counter.wsdl”};
DPWS_ADD_PTR_ATT(hServClass, DPWS_PTR_WSDL, &CounterWsdl);
DPWS_ADD_PTR_ATT(hServClass, DPWS_PTR_HANDLING_FUNCTION, &f_req);
DPWS_SET_STR_ATT(hServClass, DPWS_STR_ID, "http://.../Counter1");

/* device */
short hCounter = dpws_create_custom_device(0, -1);
struct qname CounterType = {SAMPLE_NS, “Counter”};
DPWS_ADD_PTR_ATT(hCounter, DPWS_PTR_TYPE, &CounterType);
DPWS_SET_INT_ATT(hCounter, DPWS_INT_METADATA_VERSION, 1);
ls.s = "Counter Device";
DPWS_SET_STR_ATT(hCounter, DPWS_PTR_FRIENDLY_NAME, &ls);
ls.s = "BrightBulb";
DPWS_SET_STR_ATT(hCounter, DPWS_PTR_MODEL_NAME, &ls);
ls.s = "SOA4D.org";
DPWS_SET_STR_ATT(hCounter, DPWS_PTR_MANUFACTURER, &ls);

static char scope[MAXSCOPE_LENGTH];
DPWS_ADD_STR_ATT(hCounter, DPWS_STR_SCOPE, scope);
DPWS_SET_STR_ATT(hCounter, DPWS_STR_FIRMWARE_VERSION, "1.0");
DPWS_SET_STR_ATT(hCounter, DPWS_STR_SERIAL_NUMBER, "1234");
DPWS_SET_STR_ATT(hCounter, DPWS_STR_MODEL_NUMBER, "1.0");
DPWS_SET_STR_ATT(hCounter, DPWS_STR_MODEL_URL, "http://...");
DPWS_SET_STR_ATT(hCounter, DPWS_STR_PRESENTATION_URL, "…");
DPWS_SET_STR_ATT(hCounter, DPWS_STR_MANUFACTURER_URL, "…");

int current = 0;
DPWS_SET_PTR_ATT(hCounter, DPWS_PTR_USER_DATA, &current);

/* hosted service */
short hSPServPort = dpws_create_service_port();
DPWS_SET_STR_ATT(hSPServPort, DPWS_STR_ADDRESS, “123456”);
short hSPServ = dpws_create_hostedservice(hCounter, hServClass);
```

Figure 68: Device and hosted service description with DPWS-Core

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<dcc:Config>
  <dcc:BootSequence>20</dcc:BootSequence>
  <dcc:BaseDir>data/www</dcc:BaseDir>
  <dcc:Listener id="HTTPListener" transport="http" port="4320" />
  <dcc:Registry>
    <dd:ServiceClass classId="http://.../Counter">
      <dd:Interface type="cnt:Counter" />
      <dd:WSDLInfo location="/Counter.wsdl"/>
      <trn:Implementation.static.lit />
    </dd:ServiceClass>
    <dcc:Device dcc:listener="HTTPListener" metadataVersion="1">
      <dd:Address>urn:uuid:00002694</dd:Address>
      <dd:Types>cnt:CounterDeviceType</dd:Types>
      <dd:Scopes>http://.../Counter</dd:Scopes>
      <dd:ThisModel>
        …
      </dd:ThisModel>
      <dd:ThisDevice>
        …
      </dd:ThisDevice>
      <dd:Service serviceId="http://.../Counter">
        <dd:ServiceClass classId="http://.../Counter" />
        <dd:ServicePort dcc:listener="HTTPListener">
          <dd:Address>123456</dd:Address>
          <dd:BindingInfo soapVersion="1.2" name="cnt:Counter"/>
        </dd:ServicePort>
      </dd:Service>
    </dcc:Device>
  </dcc:Registry>
</dcc:Config>
```

Figure 69: DPWS-Core XML file configuration

One advantage of this C implementation of DPWS is the implicit management of the server loop by the DPWS stack. This is illustrated in the figure 70 that represents the code for the server main program. The function *bootServer* and *stopServer* enables respectively to launch the server in another thread, and stop it. Between these 2 calls, the developer can monitoring for a command to quit the program for example.

```
static struct dpws master_dpws;

dc_ip_filter_t ip_selector = {NULL,        //all interfaces
                              DC_FALSE,    //not loopback
                              DC_PROTO_ANY, //dual IPv4/IPv6
                              0,           //no address filtering
                              NULL         //no address filtering
};

/* DPWS initialization */
dpws_init_ex(&ip_selector, NULL, DPWS11_VERSION);

/* DPWS configuration */
```

```
…

/* DPWS description */
…

/* device starting */
dpws_bind_service(hSPServ, hSPServPort);
dpws_enable_device(hLight);
dpws_server_init(&master_dpws, NULL);
bootServer(&master_dpws);

…

/* device ending */
stopServer();
```

Figure 70: Main server program with DPWS-Core

The client implementation follows the pattern introduced in the general principles subsection. DPWS-core API provides a well designed API with routines to perform the discovery with a probe request or get the metadata. Metadata are exposed to the client as a structure that contains the different dialect *ThisModel* and *ThisDevice*, that are also structures. So the navigation to retrieve each metadata fields is transparent by manipulating directly the structure. Hosted services are retrieved as service proxies from the device proxy with the routine *dpws_get_services*. To use this service proxy to invoke the service, the developer has to retrieve the corresponding endpoint reference with the routine *dpws_get_default_endpoint_ref*. Then the stub generated from the WSDL is used with the dpws stack, the obtained service endpoint reference and the structures that encapsulates the input and output arguments. The figure 71 illustrates the implementation of a DPWS client with DPWS-core.

```
/* initializes the stack */
dpws_init();

struct dpws dpws;
dpws_client_init(&dpws);

/* searches device */
int nb_device = 1;
char * namespace = “http://www.soa4d.org/”
char * type = “CounterDeviceType”;
char * scope = “http://...”;
short dev_proxies = dpws_lookup(&dpws, namaspace, type,
                                scope, &nb_device);

if(nb_devices > 0){
   short device_proxy = dev_proxies[0];

   /* gets metadata and displays friendlyName */
   struct device_metadata * metadata = dpws_get_metadata(&dpws,
```

```
                                                    device_proxy);

    printf("friendlyName: %s",
                  metadata->device_info.friendly_name);

    /* invoke service */
    int nb_services = 1;
    namespace = "http://.../Counter";
    type = "http://.../Counter";
    short * serv_proxies = dpws_get_services(&dpws, device_proxy,
                                             namespace, type,
                                             &nb_services);

    short service_proxy = service_proxies[0];

    wsa_endpoint_epr * inv_epr = dpws_get_default_endpoint_ref(
                                        &dpws, service_proxy);
    struct cnt1__TwoWayReqType in;
    in.value = 1;
    struct cnt1__TwoWayRespType out;
    dpws_call__cnt_Increment(&dpws, inv_epr, &in, &out);
}
```

Figure 71: DPWS-Core client

Advanced features

- **Eventing:** WS-Eventing is fully supported by this DPWS implementation based on gSOAP. The mechanism for eventing is similar to the mechanisms for invocation: handler is declared in the same header file than for invocation. Then the handler is implemented at the client side instead of the server side. A method is available at the server side to trigger the event, in the same way than for invoking an operation. In addition, DPWS-core API provides routine to build event sink, ordering a subscription, getting its status or renew it.
- **MTOM:** MTOM features are enabled by customizing generation tools by editing a type mapping configuration file, and then by setting a flag in the dpws structure of the stack.
- **Security:** The security features are based on the OpenSSL framework. This framework provides tools to generate certificates, and an API to manipulate them in order to verify them or perform encryption procedures.
- **Managed discovery:** DPWS-Core allows to use managed discovery from a discovery proxy. To do that, the developer can set 2 properties in the DPWS stack, when developer the client main program, for suppressing multicast mechanism, and specifying the EPR of a discovery proxy. This EPR property for the discovery proxy can be set manually of automatically when receiving Hello messages from the discovery proxy. Secondly, the DPWS-Core API provides routines to explicitly sending a lookup to the discovery proxy.
- **Discovery Proxy:** the DPWS-Core framework enables to develop Discovery proxies. A discovery proxy is considered by DPWS-Core as a special type of device which implements the DiscoveryProxy port type. This port type has operations to receive

hello and bye messages. In addition the DPWS-Core API provides new specialised operations to initialize, start and stop the discovery proxy.

Target platforms

DPWS-Core was tested on Windows with the Visual Studio Express toolchain. Samples are released with Visual Studio solution file. But they may be adapted to be compiled with a GNU toolchain and makefiles, since the generator tools are not specific to Visual Studio. In addition the generators are available for linux too.

Documentation

The DPWS-Core project is well documented and is distributed with:

- A user guide that explains how to use the DPWS features for server and clients. This user guide is slightly presented a s a tutorial that exposes each steps for the development of DPWS services.
- An HTML developer documentation that exposes all the API of DPWS-Core.
- A set of samples that demonstrates the most of the supported DPWS features.

In addition, a developer guide [84] is available on the SOA4D forge to explain the architecture the code organization.

Licence

DPWS Core is distributed under the gSOAP licence version 1.3a and GPL version 2. These licences. These two licences are open source and freely available. The developer can modify, redistribute, but there are restrictions

- The gSOAP licence implies that all modifications made on gSOAP must be provided to the initial developer and these modifications have to be described.
- The GPL licence implies that all application that use DPWS-Core must be redistributed with the GPL licence.

Advantages

- Many DPWS supported features.
- Enables to implement a Discovery Proxy.
- Well documented.

Limits

- Not very intuitive
- Low level code
- Does not support WS-Discovery digital signatures.
- DPWS 1.1 does not works properly.

4.8. DPWS4J

DPWS4J [85] is another DPWS implementation from the SOA4D initiative that aims to target Java SE or J2ME CDC platforms.

Principle

The DPWS4J stack is based on a top-down approach where client stubs and server skeletons are generated from the WSDL. Then the developer implements the server loop, callbacks, client main program and the needed handlers by using the DPWS4J library. The DPWS4J API provides all the high-level methods needed to encapsulate the management of the different DPWS conversations and description of the device and hosted services. The figure 72 illustrates the development process with DPWS4J.

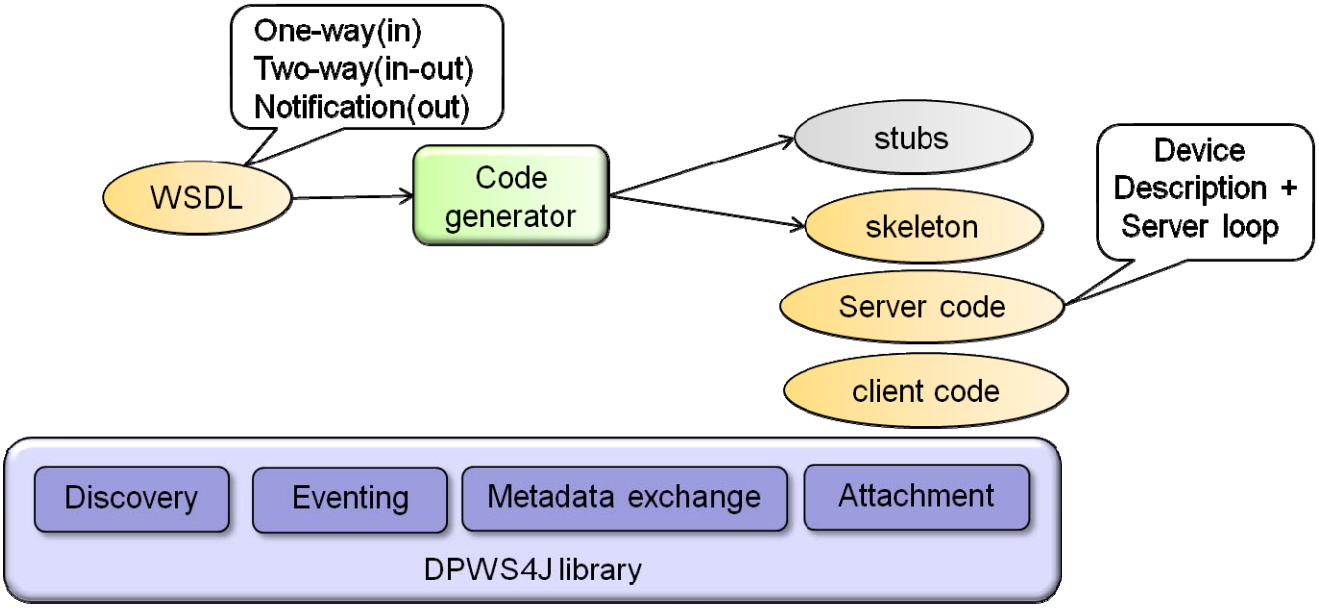

Figure 72: DPWS with DPWS4J principle

The DPWS generator, a command line tool, generates the stubs and skeletons and integrates the WSDL content into the Java code. Interfaces are generated for the service for the one-way and two-ways message exchange pattern, while interfaces are generated for the client in order to handle the notifications. Concerning the proxy to facilitate the operation invocation, client proxies are generated to deal with one-way and two-way operation, while server proxies are generated to trigger notifications. The DPWS4J stack allows generating stubs and skeleton, both one-way and two-way invocation and notification, with 2 well-known data-binding for the input and output arguments

- JAXB [87] provides a Bean-like data-binding. The XML Schema of the types is encapsulated into an object model. In addition of the skeleton interfaces, the classes representing the types are generated. The figure 73 shows an implementation in JAXB of the *Increment* operation.
- JDOM [86] provides a document-like data-binding. The XML document and constituting elements are encapsulated into an object model. Thus the JDOM objects are generic and the generator writes only the signature of the skeleton. Each part of the input message is a XMLStreamReader in the input arguments of the skeleton

while parts of an output message are contained sequentially in a XMLStreamReader. JDOM elements are then extracted from these readers. The figure 74 illustrates the implementation of the *Increment* operation.

```
public class CounterService implements Counter{

   protected int current = 0;

   public TwoWayResp Increment(DPWSContext context,
                                                    TwoWayReq req){
        int step = req.getValue();

        current += step;

        TwoWayRespImpl result = new TwoWayRespImpl(current);
        return result;
   }
}
```

Figure 73: Service implementation with JAXB

```
public Object Increment(DPWSContext context,
                        XMLStreamReader body){
   try{
       FragmentStreamReader fsr = new FragmentStreamReader(body);
       StaxBuilder builder = new StaxBuilder;
       Document doc = builder.build(fsr);
       Element valueElt = doc.getRootElement().getChild("Value");
       int step =  Integer.parseInt(valueElt.getText());

       current += step;
       Namespace ns = Namespace.getNamespace("cnt", Counter.CNT);
       Element result = new Element("TwoWayResp", ns);
       valueElt = newElement("Value", ns);
       valueElt.addContent(String.valueOf(current));
       result.addContent(valueElt);
       return result;
   }
   catch(Exception ex){
       ex.printStackTrace();
   }
}
```

Figure 74: Service implementation with JDOM

DPWS4J proposes an object model, similar to the model presented in the general principles subsection, in order to describe transparently the device and service metadata. In the server

main program, the metadata model must be registered in a registry that is in charge of activating and starting the devices. The figure 75 presents a code that uses the metadata object model to describe a device and its hosted services. The figure 76 shows the server loop that registers the device declared in the figure 75 and starts it.

```
ServiceClass counterServClass = new ServiceClass(Counter.class);
WSDLInfo wsdlInfo = new LWSDLInfoFactory.getWSDLInfo("http://..");
counterServClass.addWebService(wsdlInfo);

String manufacturerURL = "http://...";
String modelNumber = "1.0";
String presentationURL = "Counter.html";
String modelURL = "http://...";
Map manufacturers = new HashMap();
Manufacturers.put(Locale.FRANCE, "Schneider Electric");
Map modelNames = new HashMap();
modelNames.put(Locale.FRANCE, "BrightBulb");
DeviceModel deviceModel = new DeviceModel(manufacturerURL,
                                          modelNumber,
                                          presentationURL,
                                          modelURL,
                                          manufacturers,
                                          modelNames);
deviceModel.addType(new QName(…));
deviceModel.addServiceClass(lightServClass,…);

String firmwareVersion = "1.0";
String serialNumber = "123456";
Device device = deviceModel.createDevice(1,
                                         firmwareVersion,
                                         serialNumber);
device.addFriendlyName(Locale.FRANCE, "Counter");
device.addScope("http://...");
```

Figure 75: DPWS4J device and service creation

```
int port = 4320;
Registry registry = new Registry(…);
HttpPortListener http = new HttpPortListener(
                                    new JettyServletContainer(),
                                    registry, port);
registry.registerDevice(device);
registry.activateDevices();
http.createServlet(…);
http.start();

while(true){
   /* processing to monitoring a command that quit the program */
   …
}

registry.stop();
```

Figure 76: DPWS4J server loop

The client implementation follows the pattern described in the general principle subsection. DPWS4J provides through the *DeviceExplorer* class, the needed methods to perform the discovery. Then the other DPWS features are provided by the different available proxys. For exemple the proxy that corresponds to a discovered device allows to retrieve the metadata, while the service proxys that corresponds to its hosted service allows to retrieve service metadata such as WSDL.

To invoke a service, the service proxy is used in conjunction with an invoker, that was generated from the WSDL. The invoker represents a stub to invoke any service that satisfies a given WSDL specification, while the service proxy represents a parameter to this stub to indicate the endpoint reference to be called. In the contrary of the other DPWS stacks, the response of the invocation is not obtained as a return value of the stub: the developer must implements a response handler to process the output message. This response handler is similar to the callback that processes a input request message. The figure 77 shows an example of client developed with DPWS4J to invoke the increment operation of the Counter service.

```
DeviceExplorer = new DeviceExplorer();

/* Probe */
List types = new ArrayList();
types.add(new QName(…));
List scopes = new ArrayList();
scopes.add("http://...");
List devices = deviceExplorer.lookup(types, scopes);

/* Get device metadata */
DeviceProxy device (DeviceProxy) devices.get(0);
DeviceMetadata dMeta = device.getMetadata();
DeviceInfo info = dMeta.getDeviceInfo();
ModelInfo model = dMeta.getModelInfo();
System.out.println("Friendly name : " + info.getFriendlyName());

Iterator it = device.getHostedServices().iterator();
while(it.hasNext()){
   /* invoke Increment operation */
   ServiceProxy hosted (ServiceProxy) it.next();
   LInvoker invoker = new LInvoker(hosted);
   Namespace ns = Namespace.getNamespace("tns", LInvoker.CNT);
   Element result = new Element("TwoWayReq", ns);
   valueElt = newElement("Value", ns);
   valueElt.addContent(String.valueOf(current));
   result.addContent(valueElt);

   invoker.invokeIncrement(jDom, new TwoRespCallback(){
        protected void TwoWayResp(DPWSContext context,
                                  XMLStreamReader b){
             FragmentStreamReader f = new FragmentStreamReader(b)
             StaxBuilder builder = new StaxBuilder();
             Document doc = builder.build(f);
             Element e = doc.getRootElement().getChildren().get(0)
```

```
                double val = Double.parseDouble(e.getText());
            }
        }
    );
}
```

Figure 77: DPWS4J client

Advanced features

- **Eventing:** DPWS4J supports notifications only, with the same mechanisms than one-way operations for implementing handler or triggering events. Instead of creating a registry that makes work devices, the client instantiate a EventHandlerListener that is linked to the implemented handler. This EventHandlerListener is finally started before receiving notifications. To perform a subscription, the generated service proxy provides a subscribe methods that returns a subscription object. This object allows to manage the subscription for getting the status, renewing or cancelling. In addition, the developer can add to it a end subscription handler.
- **MTOM:** The MTOM features are directly available in DPWS4J. Attachments are considered as different parts of the input message than the regular arguments. In the server side, attachments are accessible from the context given to the callback. This context contains the input and output messages. At the client side, attachments are added to the invoker before the invocation.
- **Security:** The security features are not available directly in the DPWS4J archive. Indeed, SOA4D provides another project, WS-Security for DPWS4J, which represents an add-on to manage the security.

Target platforms

DPWS4J works on Java SE and Java CDC and was tested with Java SE. However DPWS4J does not work on Android because the generated code is based on the javax.xml.stream API, which is not part of the Android SDK.

Documentation

DPWS4J provides a poor documentation which is not up-to-date:

- The user manual gives a quick overview of the DPWS4J principle of use and presents a tutorial to develop DPWS servers and clients with JDOM and JAXB models. This manual corresponds to a very old version of DPWS4J. As a consequence, the features presentation is incomplete.
- The tutorials of the user manual are accompanied with samples to demonstrate the main DPWS features.
- The provided Javadoc is very incomplete.

Licence

The DPWS4J stack is distributed under the LGPL licence.

Advantages

- Uses JAXB and JDOM that are well known frameworks. So XML Schema are well supported.
- Quite easy to use.

Limits

- DPWS4J does not support directly the security features. We can see on the SOA4D forge, the project WS-Security for DPWS4J which represents an add-on to DPWS4J in order to bring support for WS-Security. However this add-on cannot be downloaded because it is in development. So we cannot have security features with DPWS4J.
- Poor documentation
- Does not work on Android.
- The API is less intuitive than other API like JMEDS.

4.9. Comparative summary

In this section, we show a comparative summary of the different tested DPWS implementations. We divided this comparison into 3 parts:

- The DPWS features in order to compare the DPWS support for each implementation.
- The development facilities to compare how the framework makes it easy or not to implementation a DPWS service.
- Other comparing criteria's that are often taken into account when we choose to use a technology in a project.

4.9.1. DPWS Features

In this sub-section of this summary, we compare how the different parts of the DPWS specification and other interesting features are supported by the different DPWS implementations. We compare the implementations according to the following features:

- The supported DPWS version. We pay attention if the implementation is able to choose dynamically between DPWS 1.0 and DPWS 1.1 despite of the uncompatibility of these versions.
- The support for the server-side and client-side.
- The support for the discovery part of DPWS. In particularly, we verify if the implementation is able to perform filtering according to device type or device scope specified in probe messages. We compare if the implementation support the discovery with the managed mode and if it allows to implements the implied Discovery proxy. Finally we verify the use of two possible scenarios for the discovery:
    - Active discovery: the service is started and then the client sends a probe message to discover the available devices.
    - Passive discovery: the server is started after the client and sends a Hello message.

- The support for the metadata exchange part of the DPWS specification
- The support of the eventing part of the DPWS speicification
- The support for the MTOM features to exchange multimedia content. One of the implementation proposes the streaming: a new feature that is based on MTOM to facilitate the transfer of big multimedia contents. So we compare the other implementations in this new feature.
- The support for the security part of the DPWS specification, namely WS-Discovery compact signature and SSL.
- The interpretation on-the fly of the WSDL after the discovery of the devices and their hosted services.
- The invocation pattern supported by the DPWS implementation. We verify also that fault raising is supported.
- The types supported in argument for invocation and eventing.
- The IP version supported.

For each of these criteria, we put the following signs:

- ✓ means that the feature is fully supported
- ~ means that the feature is not fully supported. The code may be incomplete or there is bugs
- - means that the feature is not supported at all.

		.NET	µDPWS	JMEDS	Axis2	gSOAP	DPWS-Core	DPWS4J
DPWS version	1.0	✓	-	✓	✓	✓	✓	✓
	1.1	~	✓	~	-	~	~	-
	Dynamic	-	-	-	-	-	~	-
Server		✓	✓	✓	✓	✓	✓	✓
Client		✓	-	✓	-	✓	✓	✓
discovery	Device filtering	✓	-	✓	-	✓	✓	✓
	Scope filtering	-	-	✓	-	✓	✓	✓
	Managed discovery	~	-	~	-	-	✓	~
	Discovery proxy	-	-	-	-	-	✓	-
	Active discovery	✓	✓	✓	✓	✓	✓	✓

		.NET	µDPWS	JMEDS	Axis2	gSOAP	DPWS-Core	DPWS4J
	Passive discovery	✓	-	✓	-	~	✓	✓
Metadata exchange		✓	✓	✓	✓	✓	✓	✓
Eventing	Notification	✓	-	✓	-	~	✓	✓
	Sollicit-Response	-	-	✓	-	~	✓	-
MTOM	Attachment	✓	-	~		-	✓	✓
	Streaming	-	-	~	-	-	-	-
security	WS-Discovery Compact signature	-	-	~	-	-	-	~
	SSL	-	-	~	✓		✓	-
WSDL Interpretation		-	-	✓	-	-	-	-
Invocation	One-Way	✓	✓	✓	✓	✓	✓	✓
	Two-Way	✓	✓	✓	✓	✓	✓	✓
	Faults	~	-	~	✓	-	✓	~
Types	Primitives	✓	✓	✓	✓	✓	✓	✓
	Sequences	✓	✓	✓	✓	✓	✓	✓
	Choice	✓	✓	✓	✓	✓	✓	✓
	Restrictions	✓	✓	✓	✓	✓	✓	✓
IP	IPv4	✓	✓	✓	✓	✓	✓	✓
	IPv6	-	✓	✓	-	~	✓	-

Figure 78: DPWS features comparison

4.9.2. Development facilities

In this subsection, we compared the development methods that are used to develop DPWS devices:

- Top-down and bottom-up approaches are well-known development methods in the Web Service domain. Top-down approach allows to generate code from a contract specification while the bottom-up approach allows to generate the contract from an implementation in order to be used by a client.
- How the device metadata (e.g. ThisModel and ThisDevice section, UUID, etc) are described.

- How the service metadata are described concerning the EPR for example.
- How the configuration of the DPWS stack is described, concerning the ports to be used for example.
- The data binding used to implement invocation and events.
- The facility to use the DPWS API. 3 stars means that the API is intuitive, pleasant and make a nice code. Only one star means in the contrary that the API is difficult to understand, and the written code is unreadable, too verbose.
- The usability of the whole DPWS framework. We take into account the number of steps required to develop a server and its client, the simplicity and automation of each step.
- The development platform that can be used to develop DPWS server and client with this implementation.

		.NET	µDPWS	JMEDS	Axis2	gSOAP	DPWS-Core	DPWS4J
Development method	Top-down	✓	-	-	✓	✓	✓	✓
	Bottom-up	-	-	✓	✓	-	-	-
Device description		Code	DPWS + text files	Plain text	XML file + DPWS	XML DPWS like	Code or XML	Code
Service description		Code	Text files	Plain text	XML file		Code or XML	Code
Configuration		Code	Code	Plain text	XML file	Code	Code or XML	Auto
Data Binding			XML content	Generic encapsulation	Type mapping, ADB, JAXB, AXIOM	Type mapping	Type mapping	DOM or JAXB
API comprehension		***	*	***	-	*	**	***
Automation/ usability		**	*	***	**	*	***	**
Development platform		Microsoft Visual Studio Express	GNU toolchain or MinGW	Eclipse	Eclipse + Ant	Cmake, + C toolchain	Microsoft Visual Studio Express, GNU	Eclipse

Figure 79: DPWS development facilities comparison

4.9.3. Other comparing criterias

In this section we compare the implementation according to criteria that are often used when we decide to integrate a technology in a project:

- The targeted platform: which programming language, architecture, OS. We also measured the RAM and ROM required to respectively execute and store the program.
- The distribution licence. This criteria is important to verify that there is not conflict between the targeted licence and the licence of the integrated framework.
- The availability of the source: this ensures that we can modify and adapt the framework to our needs.
- The documentation of the framework.

		.NET	µDPWS	JMEDS	Axis2	gSOAP	DPWS-Core	DPWS4J
Target platform	Language	C#	C	Java	Java	C	C	Java
	CPU/OS	ARM9 (Tahoe, FEZ)	PC, TelosB, AVR Raven	Java SE, CLDC, Android,	Java SE	Linux, Windows, ARM9 (FoxBoard, NetBurner)	Linux Windows	Java SE, Java CDC
	RAM	~300 Kb	~40 - 500 Kb	3 – 30 Mb	~50 Mb	~250Kb	~3 Mb	2 – 30 Mb
	ROM	~30 Kb	~7 - 250 Kb	~1 Mb	~20 Mb	~1.5 Mb	~800 Kb	~2 Mb
Licence		Apache 2.0	FreeBSD	EPL	Apache 2.0	GPL/ LGPL	LGPL	LGPL
Sources available		-	✓	✓	✓	✓	✓	✓
Documentation	Tutorials	✓	-	-	-	✓	✓	✓
	Samples	✓	✓	✓	✓	✓	✓	✓
	User documentation	✓	-	-	-	✓	✓	✓
	Programmer documentation	✓	-	~	-	~	✓	~

Figure 80: DPWS comparison

4.9.4. Conclusion

The most common DPWS features are globally well supported by the different implementation of DPWS. However, some implementation distinguishes itself by interesting extra features in addition of a full support of DPWS, ease-of-use or a good documentation:

- JMEDS is an interesting implementation to support fully DPWS and interpreting WSDL on the fly. This feature could, with an adaptation work, to add semantic for example and discovering the signature at runtime to invoke the operation. This dynamicity is not possible with another implementation where stubs and skeletons are generated at design time. In addition JMEDS works on Android and fixes could be made to work on SunSPOT.
- DPWS-Core is an interesting implementation to support fully DPWS and to develop discovery proxies, an important feature to build large scale systems based on DPWS.
- µDPWS is a promising implementation to target very constrained devices, but it lacks of maturity to be more automated and support important feature like eventing.

5. Projects and research works based on DPWS

5.1. Windows Rally

Windows Rally is an architecture and a toolset for improving security, reliability and usability of network-connected devices. It enables an easier integration of devices with the end-user's digital environment, while advancing control of network Quality of Service (QoS) and diagnostics for data sharing, communications and entertainment. This architecture is natively integrated in the last Windows versions: Windows Vista and Windows 7.

As illustrated in the figure 81, the Windows Rally architecture is constituted of the following components:

- **Device Profile for Web Services (DPWS)** is used in the Windows Rally architecture in order to provide discovery and control on the DPWS compatible devices. The Web Services on devices are presented as utilities to configure and monitor them. The DPWS is based on the WSDAPI for .NET. The support of DPWS devices is transparent in Windows Vista/7: the figure 82 is a screenshot of the network explorer view under Windows Vista. The projector, which is DPWS compatible, is showed in the same manner than the other networked machines.
- **Universal Plug 'n Play (UPnP)** and DPWS are similar and complementary protocols except that UPnP is not based on Web Service standards and is more used for multimedia sharing between devices and computers. For example, Windows Media Player includes a UPnP server and client, allowing it to share media with other UPnP devices, and to view media from other UPnP devices, like the Freebox, the last game consoles as the Xbox 360 and the PlayStation. In the figure 83, the "Lire jusqu'à" option of Windows Media Player proposes to read a playlist on the Pinnacle Systems ShwoCenter device.
- The **LLTD topology map**: this component aims to discovering dynamically devices at the data layer in the OSI model. The data layer is the first layer on top of the physical layer and is in charge of routing packets from a machine to another through routers. The LLTD Topology Map enables to discover the topology the network and present it with a schematic view representing the overall network and devices health in a user-friendly GUI. In addition, each node of the obtained graph can show local information like the IP address or the connection properties, signal strength and so on. These informations are interesting to identify and locate more easily the network issues, especially for streaming content over a wireless connection. The figure 84 is a screenshot under Windows 7, which shows the topology of a network with a hub, a switch and a modem to connect the local network to internet. This window is obtained by clicking on *start->Control Panel->Network and Internet ->Network and Sharing Center->View full map*. If, for example, the modem fails to connect to internet the red link between the modem and the internet picture becomes black with a red cross.
- The **LLTD QoS Extensions** works in collaboration with the LLTD Topology map component to ensure the QoS between connected devices. This QoS extension targets mainly the support for Audio and Video (A/V) streaming where the traffic competes with other data and best-effort traffic. The LLT QoS Extensions component contains a API, qWave, that allows application to dynamically adapt to changing network conditions in real time. This technology enables A/V applications to provide a quality user experience, especially on wireless home network. qWave enables the

auto-discovery of end-to-end QoS compatibility, the bandwidth estimation, the intelligent packet prioritization, the congestion notification and the managing of multisource stream coexistence. The devices have to use the qWave interface to be compatible with this QoS extension.

- **Windows Connect Now (WCN)** technologies include specifications and tools to enable simple and secure configuration of Wi-Fi networks and for provisioning of wireless devices such as wireless access points, PCs, servers, printers cameras and game consoles. WCN provides an assistant to configure efficiently and easily a new network from only one machine and configure the others by distributing the settings through a USB key or an Ethernet connection. The settings may contain the network login and passwords for example. The figure 85 is a screenshot of Windows Connect Now wizard to configure a new wireless network with an access point, while the figure 86 shows the wizard to add a new device to the network thanks an USB key.
- **Plug and Play Extensions (PnP-X)** component allows network-connected devices to appear as devices inside Windows and provides an installation experience that is similar to attaching a bus-connected device. The PnP-X component is based on the DPWS component to use the appropriate drivers and metadata device information needed to perform the installation. Thus the drivers are specified thanks to XML format that are transmitted by the DPWS conversation.

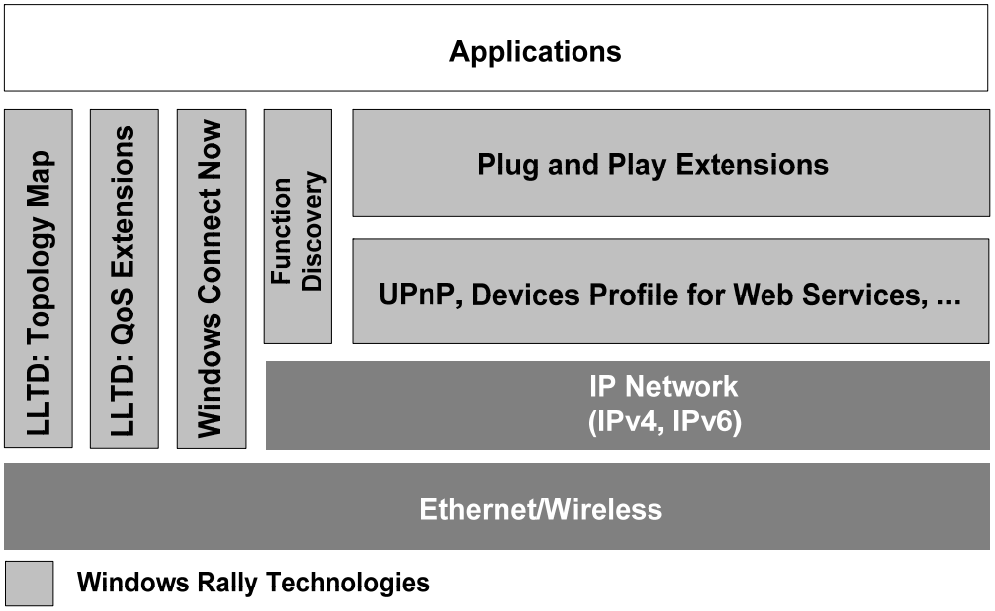

Figure 81: Windows Rally architecture

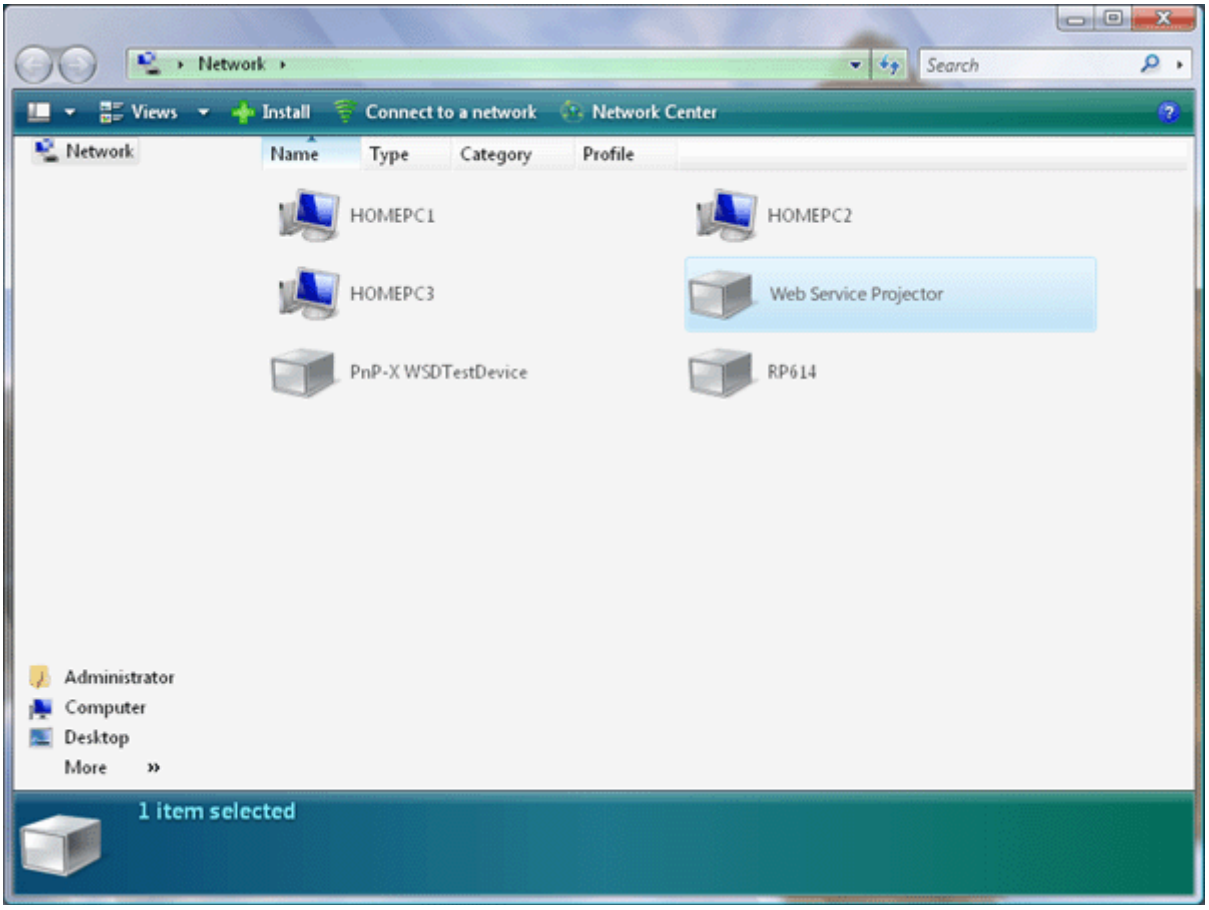

Figure 82: DPWS managing on Windows Vista

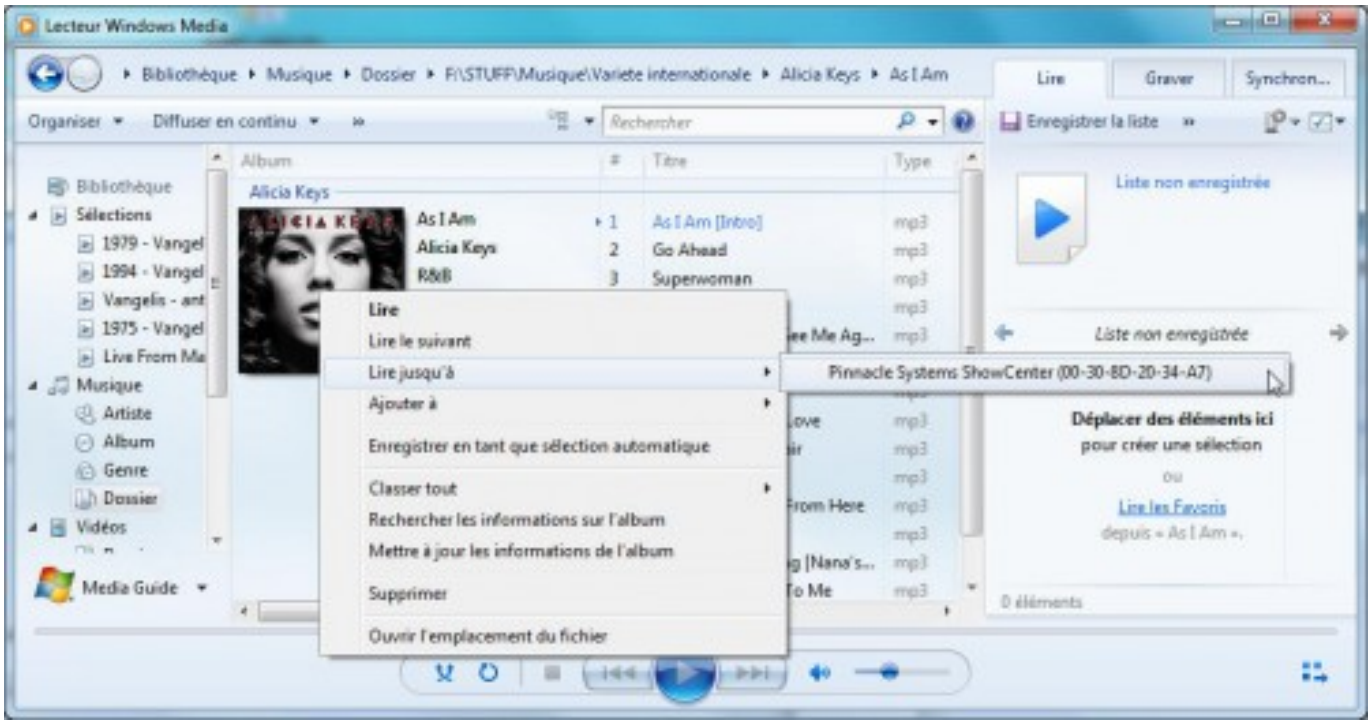

Figure 83: UPnP support with Windows Media Player under Windows 7

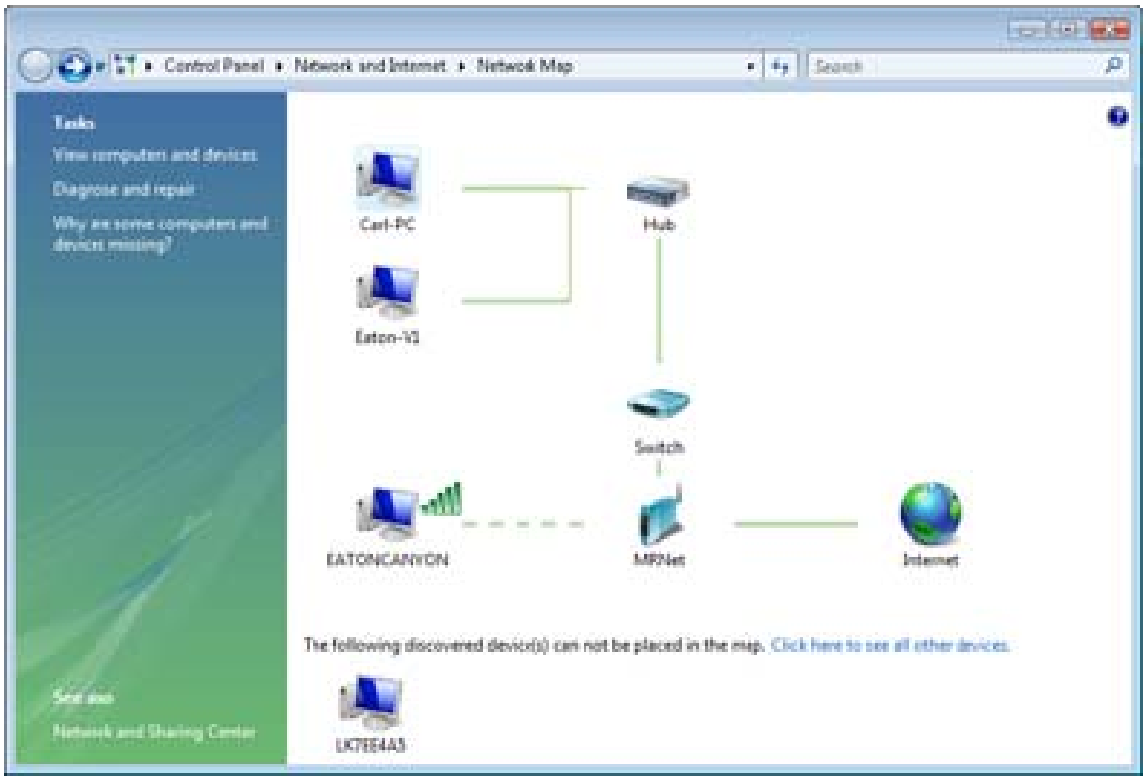

Figure 84: Network mapping with the LLTD protocol

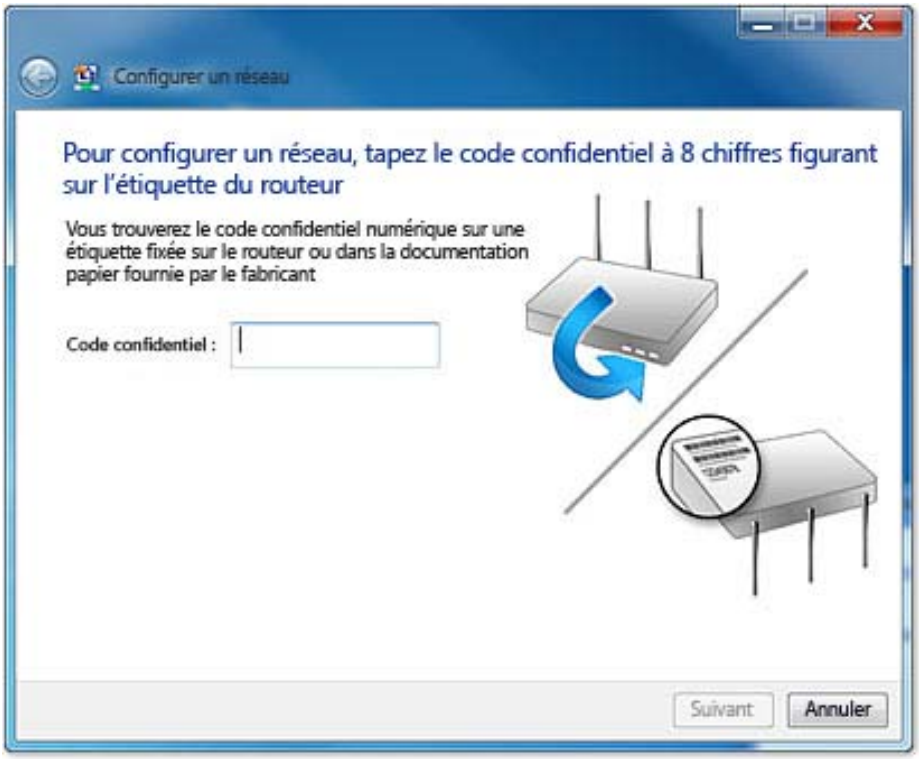

Figure 85: Windows Connect Now wizard to configure an access point

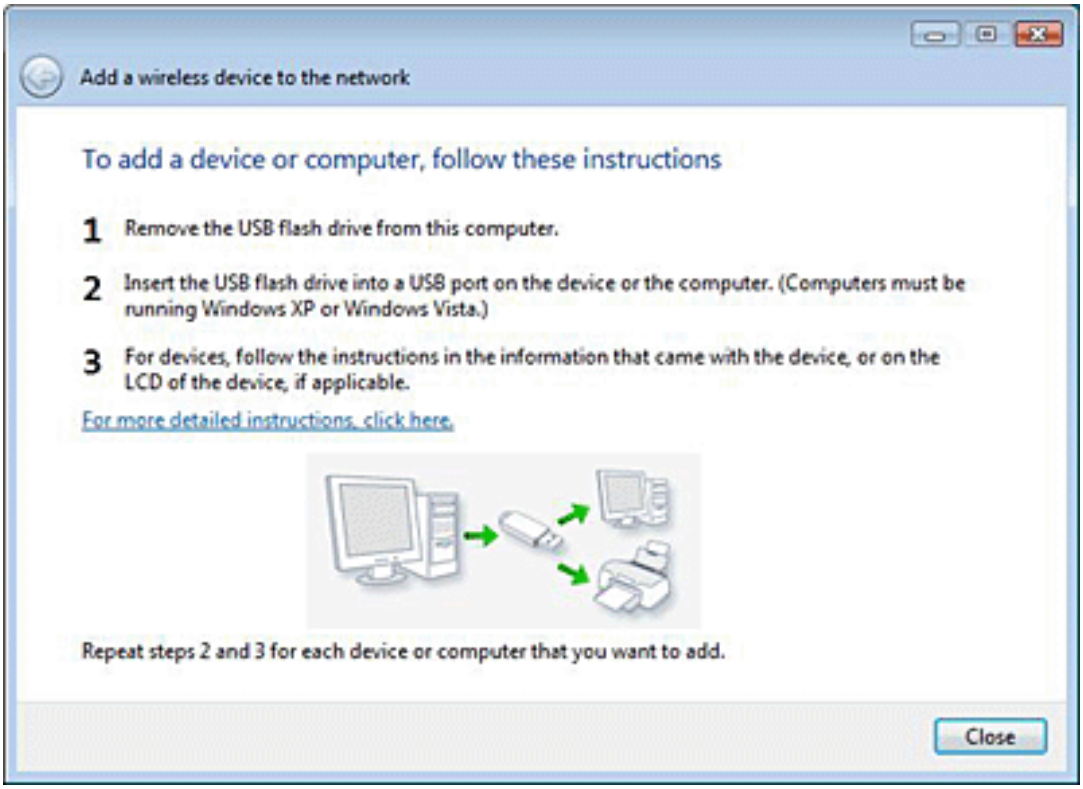

Figure 86: Windows Connect Now wizard to add a device to the network

To conclude on the Windows Rally technologies, DPWS is a mainstay in this architecture to enable the discovery and the installation/configuration of a networked device in the same manner of a Plug-&-Play device.

5.2. DPWS & OSGi

DPWS and OSGi are known as accomplished and complementary approaches to deal with SOA architecture and devices in the context of pervasive systems and Ambient Intelligence. In one hand, OSGi is broadly used in applications such as mobile phones, industrial automation, grid computing, as we can see in the OSGi market and solutions [6]. An example of project based on OSGi is IST Amigo (Ambient Intelligence for the networked home environment) [90][91]. The goal of IST Amigo project is the promotion of domestic IP networks to improve everyday life, addressing all vital user aspects: home care and safety, home information and entertainment, and extension of the home environment by means of ambience sharing for advanced personal communication.

OSGi is a framework which provides to developers an integrated development environment based on bundles. This environment offers several useful APIs for the governance of a system composed by services. Thus OSGi offers a way to compose exposed services in order to build a complex system by “plugging” bundles with their required or provided interfaces. The IST Amigo project used OSGi as the main architecture element to manage the integration of the networked component into the system. But OSGi was not self-sufficient to deal with all the challenges: the project turns to the Web Services standard such as WS-Discovery, and WS-Eventing to deal with the discovery of new services and the sending or receiving of event through the architecture.

On the other side DPWS allows to integrate a wide variety of devices in a SOA, from the tiny device to the workstation and servers. DPWS performs the discovery and gets at runtime the metadata of devices. This specification is principally based on the Web Service standard to

deal with changing environment populated with devices. But DPWS represents only a protocol to address device into a SOA architecture, and does not provides tools for composing them into a high-level feature.

So many researchers and projects were interested in integrating DPWS and OSGi. A. Bottaro in his thesis [93] studied how to integrate protocols dedicated to devices in an OSGi architecture to realize the Home SOA, a framework that makes benefit of recent components models and the service platform concept. In particular, he proposes an integration of DPWS as a new driver in OSGi. The idea was based on the Device Access OSGi service introduced in the section 2, in order to provide a base driver for DPWS that is usable in an OSGi platform. This base driver should have a generic API for handling all the DPWS devices. This integration allows 2 main scenarios defined by The OSGi Alliance in his DPWS Discovery Base Driver specification [92] :

- The OSGi platform accesses a DPWS service. The platform can react when the DPWS device joins the network or leave it. The DPWS Base driver is responsible for managing the discovery and metadata exchange conversation, and then reify the device into the OSGi registry.
- The features deployed in the OSGi platform are exposed as DPWS Web services on the network. The DPWS base driver is responsible to advertise the other device with Hello and Bye messages, and managing the DPWS conversation at the server-side.

The figure 87 shows how this DPWS Base driver is integrated into the OSGi architecture to import DPWS services or export the platform as DPWS services. This driver is composed of two bundles:

- The DPWS interface API that corresponds to the interface to interact with a DPWS service. For exemple this interface specifies the operation to launch the device discovery or exchanging metadata. The goal is the presentation of a DPWS device as a Java object.
- The DPWS Base Driver is an implementation of this interface to reify the remote services into proxy services in the service registry of the OSGi platform. This implementation is based on the DPWS4J stack.

The DPWS Base driver interacts with other bundles that mutualise the features for other OSGi applications:

- The HTTP Service bundle is used to manage the DPWS communications on top of the HTTP protocol.
- The log service bundle is used to perform logs.
- The service Compendium bundle is used to configure the DPWS Base driver according two properties for saving bandwidth and storage capacity: the late requesting and the late loading. It consists of performing the requesting for device information or the loading only when it is explicitly needed by the application based on DPWS, instead of at the beginning.
- The Pseudo Device bundle and the Control Point bundle represent respectively the device and its hosted services.
- The WSDL Description Factory bundle allows representing the services of the devices as Java Objects from their WSDL. While by default a Control Point bundle represents a generic interface for a service, the WSDL Description bundle can parameterize a Control Point bundle to match with a specific WSDL specification.

- The DPWS Service Factory bundle is used for the exportation of the OSGi services as DPWS services. This module manages the DPWS metadata and WSDL description of the exported services.

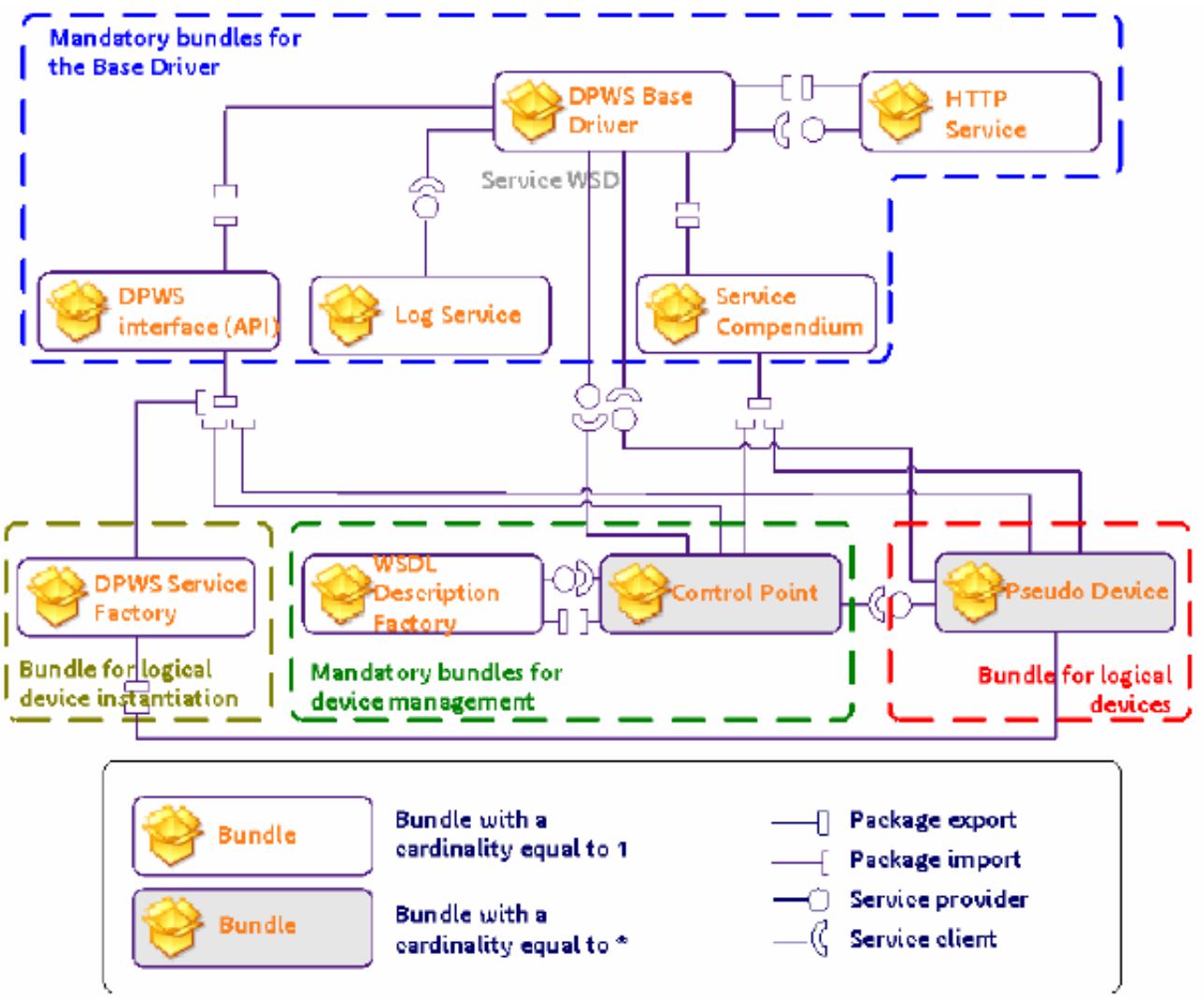

Figure 87: DPWS Base Driver

This DPWS Base Driver can be then embedded for example in smart application (e.g an application that runs on a smart device like a smartphone) that is based on OSGi. A. Bottaro explained in his thesis a way to integrate the DPWS device in such an application and interoperate with other type of device like UPnP devices. This is done by using the existing concept of refined driver. While the base drivers for DPWS, UPnP and others hide the remote aspects of them with driver that acts as proxys, the refined drivers hide the heterogeneity between the different device protocols to match with an application domain. The figure 88 is an illustration of the integration of the DPWS Base driver and other like UPnP Base Driver, Bonjour Base Driver and x10 Base Driver into a smart application. Each protocol is firstly refined to match with smart application domain. They are then abstracted with a general Driver, the Smart Application driver, which can be used by the application without worrying about the underlying protocols.

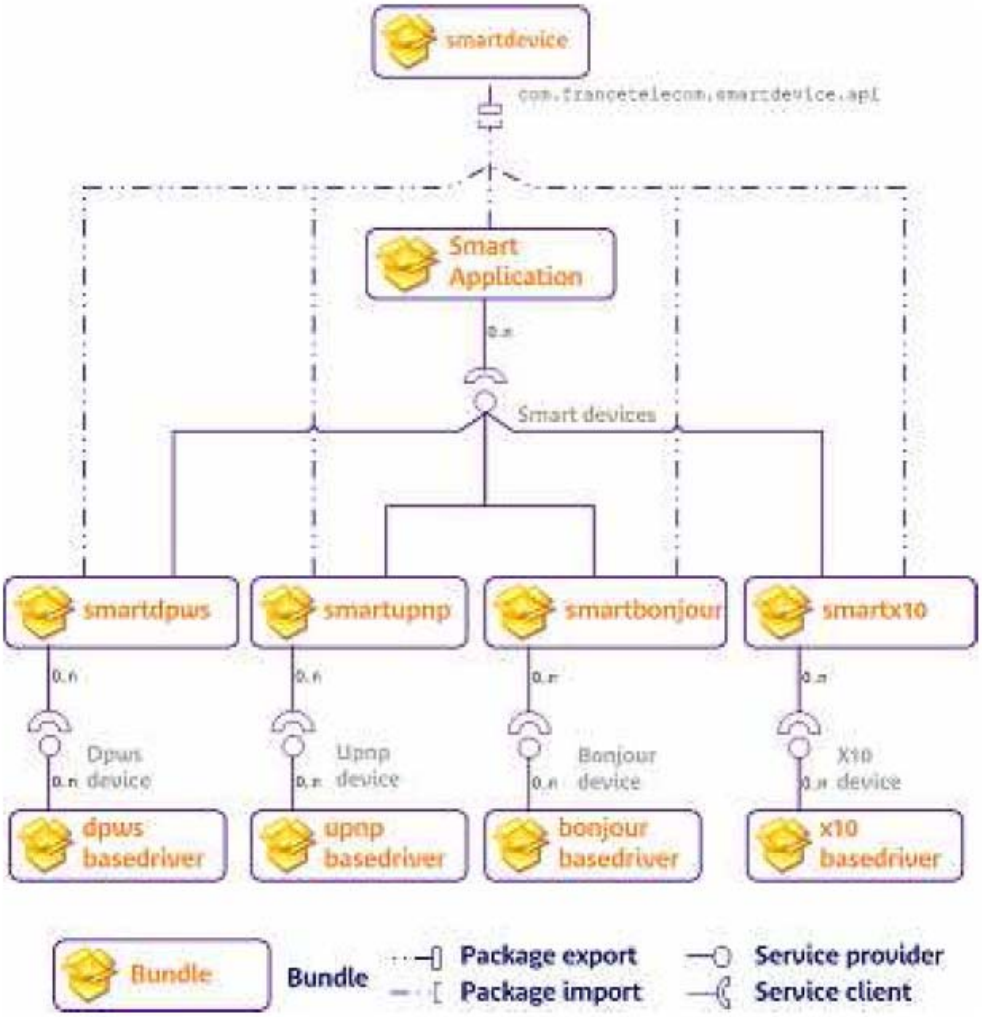

Figure 88: DPWS Base Driver in a smart application

The DPWS Base driver was added to the IST Amigo project after its end in 2006, in order to complete the first tentative to discover services, and adding support for DPWS devices.

This DPWS Base Driver was also implemented in the context of the ANSO project (Autonomic Networks for SOHO user) [94]. The ANSO project addresses the challenges for enhancing the quality of life for people living in context-aware digital homes, namely dependability issues like reliability, availability, security and manageability in small enterprise and home networks. The aim of this project is a system that enables homogeneous access to heterogeneous multimedia services, supports the full variety of automation, computing and entertainment devices.

While the previous project and research works are principally in the context of the SOA4D initiative, the concurrent initiative for DPWS, WS4D, leads also works on the integration of DPWS and OSGi. They adopted an approach a little different, since they do not use the OSGi Device Access Service features to integrate DPWS as a driver. Nevertheless they address the same scenarios (e.g. importing DPWS devices and exporting services of the platform as DPWS services) [97]. This architecture is based on JMEDS, a DPWS implementation provided by the WS4D initiative. JMEDS is wrapped as a regular bundle.

This implementation has remarkable features such as the interpretation on the fly of WSDL specifications. Thus the dynamicity to providing new adapted proxy into the OSGi platform is already performed by the JMEDS stack and is completed by a Proxy generator bundle for the client side. The proxy is packaged into new bundle, while other complementary bundles are deployed for marshalling or eventing managing. The figure 89 depicts the architecture for consuming native DPWS devices, export a service from OSGi to a DPWS service or bridge two OSGi platforms with services.

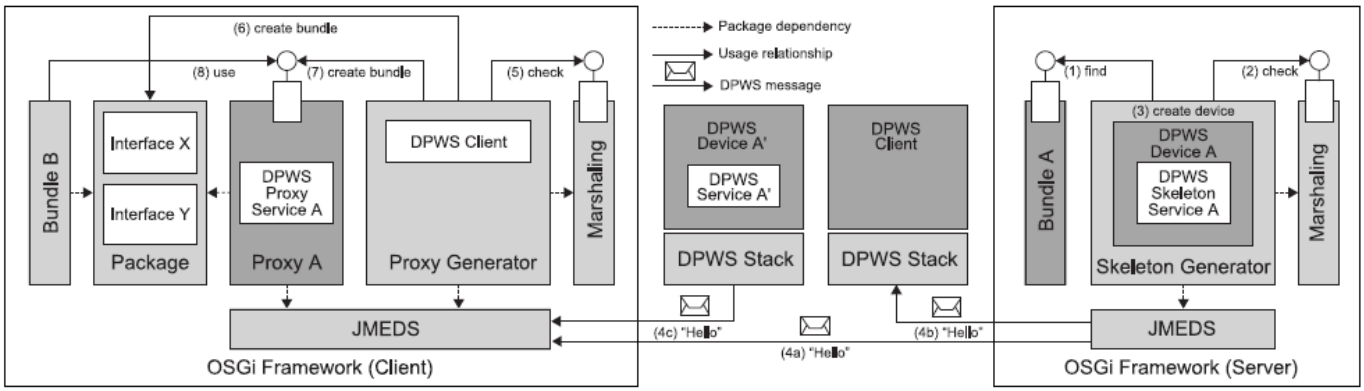

Figure 89: DPWS and OSGi integration in the OSAmI project

One of the advantages for this DPWS integration is a better transparency than with the DPWS Base Driver. Effectively, the interaction is performed in the same manner with local DPWS devices in a OSGi platform than with native DPWS devices. In the contrary, the DPWS Base driver addressed only external devices, or export externally.

This work is a part of the OSAmI (open Source Ambient Intelligence) [98] project that starts in 2008 and ends in 2011. This project targets open source common foundation for a dynamic service-oriented platform which is able to personalize itself in large diversity of cooperating Software Intensive Systems. Software Intensive Systems include large-scale heterogeneous systems, embedded systems for automotive applications, telecommunications, wireless ad-hoc systems, business applications with an emphasis on web services. OSAmI consists of a number of networked national sub-projects focussing on different areas as an approach for converging towards common technology foundations. The OSAmI Deutsch sub-project aims to deliver an open source platform build on top of OSGi specification. This platform enables easy and efficient interoperability of different devices with the service oriented architecture. One of its research topics was allowing non-expert users to deal with highly-complex device and sensor environment and a novel approach for implementing OSGi Remote Services with DPWS. As a result, this project contains 3 DPWS implementations of the WS4D initiative in order to cover the larger variety of devices: JMEDS, µDPWS and WS4D-gSOAP. These implementations are used in use cases that focus on the health care sector, in particular cardiologic rehabilitation of patients at home after surgery in the clinic. The patient performs training sessions that are supervised by a doctor in the clinic. The interactions between the patient and the supervisor are supported by two gateways, and the doctor is informed about the state of patient thanks to various devices including the ergometer and other sensors. Usage and operation of the system overall is facilitated by an easy-to-use user interface. The figure 90 illustrates the training session use case.

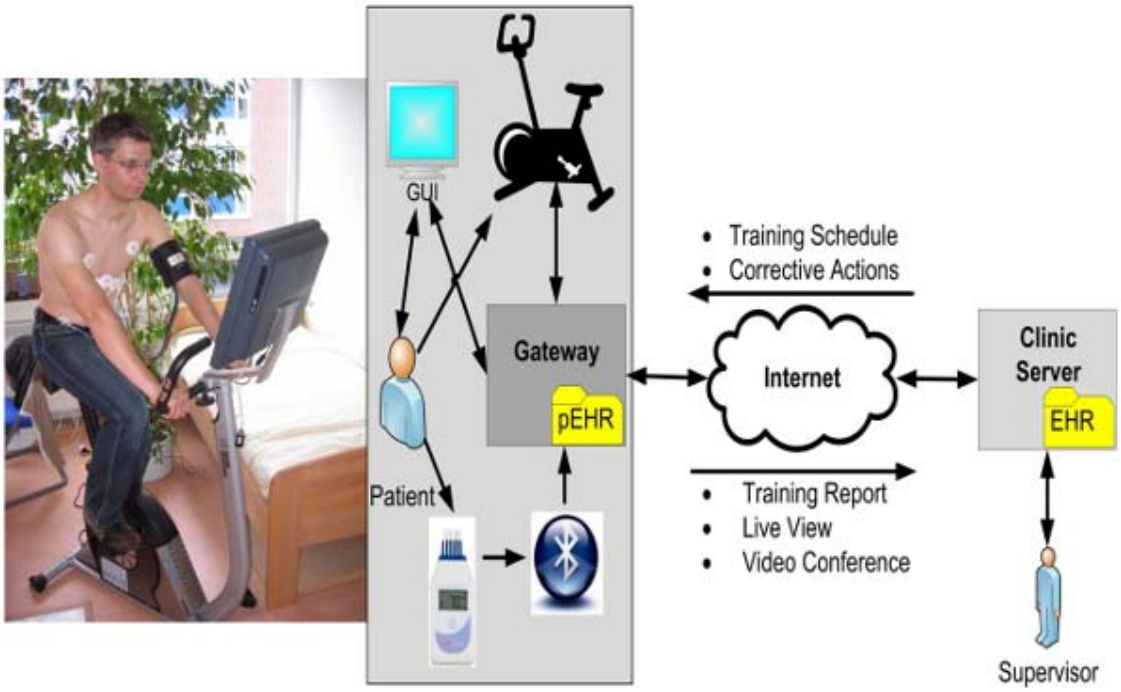

Figure 90: Training session for cardiologic rehabilitation use case.

The WS4D initiative focuses also on the usability of a system based on OSGi to combine DPWS or UPnP devices. They evaluated that most of the composition tools are dependant to technology and it results that although the definition of high-level language, they still require technical knowledge and programming experience. The WS4D initiative provides a new tool, WS4D-PipesBox, which allows combining devices and Internet services in an easy way, without programming knowledge. It can integrate any technology like DPWS, UPnP or Bluetooth by wrapping their features into modules that can be combined in a graphical way. WS4D-PipesBox provides principally a web-based frontend to a system based on OSGi and technology to integrate devices. The user can combine the services, represented by boxes, by building pipes between the output and the inputs of the services. The figure 91 is a screenshot of the WS4D-PipesBox tool. Black points and white points represent respectively outputs and inputs. This example sends a SOAP request to get the value measured by a sensor. The response is processed by an XSL transformation to extract the obtained value. If the value is higher than 500, it sends an SMS.

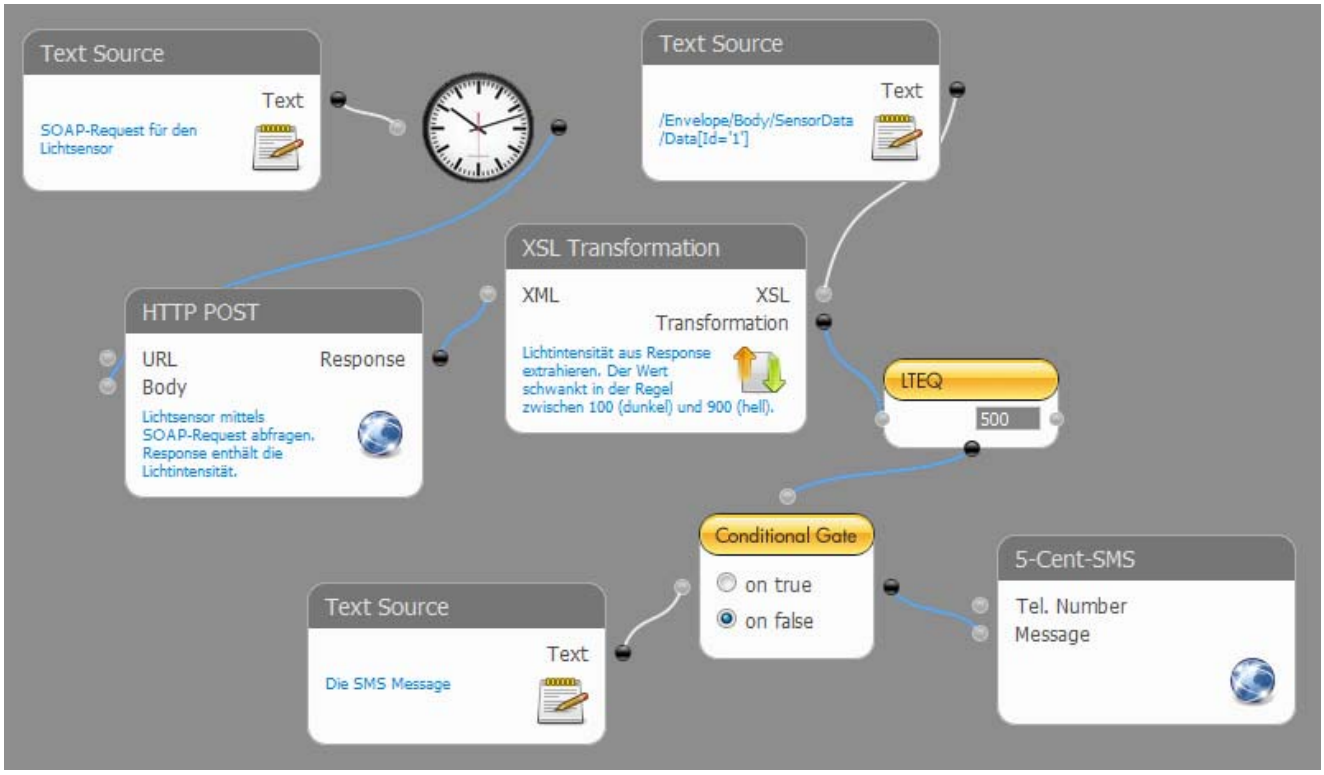

Figure 91: The WS4D-PipesBox frontend screenshot

5.3. DPWS ecosystems

Service Oriented Device and Delivery Architecture (ITEA/SODA) project started after the end of the ITEA/SIRENA project and is presented as a continuation by extending the framework implemented in SIRENA. It ends in 2008 and is a large contribution to the OASIS standardization of DPWS in 2009.

The objectives of SODA were to create a comprehensive, scalable and easy-to-deploy service-oriented ecosystem for embedded devices. Such an ecosystem is indispensable for bringing the device-level SOA approach closer to widespread industrial use. The main contributions of SODA are

- Providing a complete set of tools for design, development, deployment and run-time support that can be used throughout an application's lifecycle:
    - A **SOA4D Maven repository** in order to centralize the SODA contribution and help to organize the dependencies for projects based on SODA.
    - **DPWSBuilder** is a tool that facilitates the creation of devices and hosted services by providing a GUI for editing the device description and importing the WSDL of the hosted services. Then this tool enables to generate the source files and the programmer has just in charge of implementing the operations provided by the hosted services. DPWSBuilder is based on the DPWSCore 2.1.0 stack implementation of DPWS. The figure 92 is a screenshot of DPWSBuilder where the device SimpleLight is created with a hosted service Lighting that provides the operations Switch and GetStatus.
    - **LittleExplorer** is a GUI tool to explore the available devices on the network. It enables to show, for each devices, their properties like the manufacturer name

or the model name. The figure 93 is a screenshot of Little explorer, where it recognizes the compatibles DPWS printers available at the INRIA network.

  - **C++ funtionnal test suite** that corresponds to low-level tests on the different ports or isolated functionalities. These tests are usable as projects in Microsoft Visual Studio.
  - **JMeter functional test suite** allows to perform high-level unit tests on the DPWS features. JMeter is an open source application aiming at testing functionally an application and measuring its performances. This software allows to design scenarios where the tester indicates how send inputs and how to interpret the outputs. The outputs can be compared to assertions to verify the validity. The results (e.g. verifications and measures) can be organized in a summary report or a view result tree. The DPWScore functional test suite proposes a set of test plan usable with JMeter to test the DPWSCore examples.

- Improving and extending the run-time infrastructure:
  - In term of **performances**: the DPWS stack has been optimized to target low-cost devices with very limited resources.
  - By integrating the **security**
- Seamlessly integrating device-provided services with high level business processes.
- Developing elaborated experimental applications in several application domains
- Conducting feasibility studies on application of the device-level SOA approach in application areas.

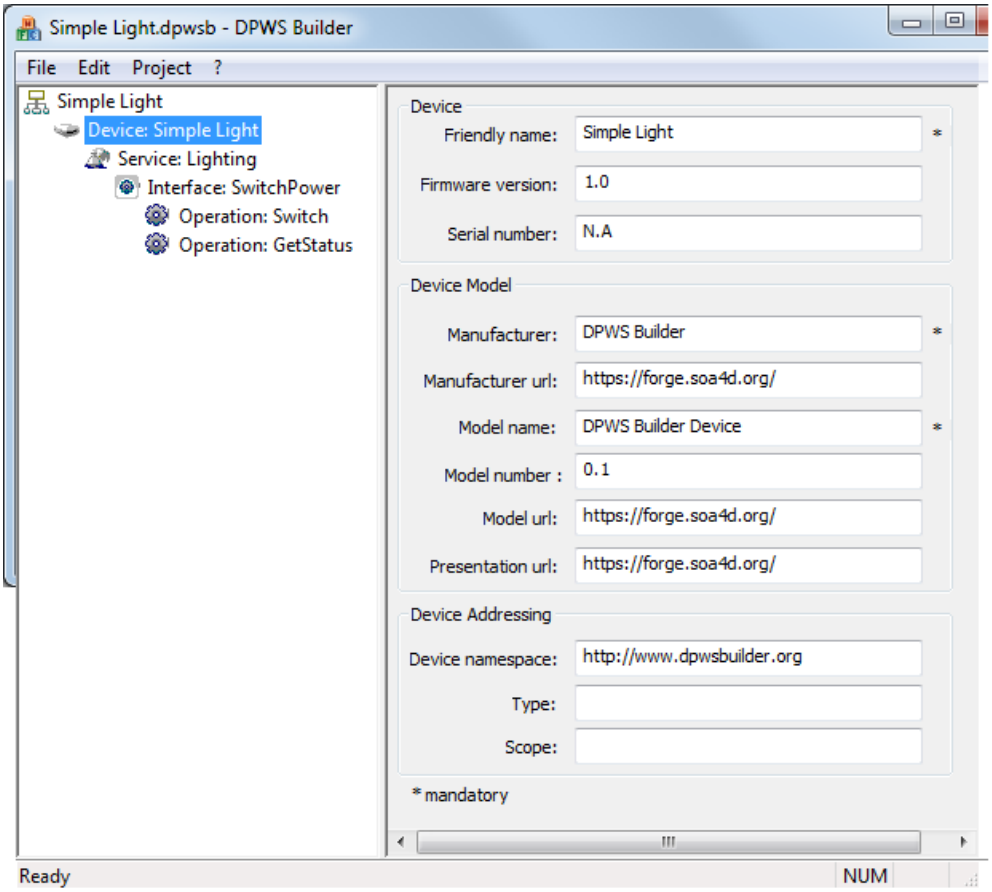

Figure 92: DPWSBuilder screenshot

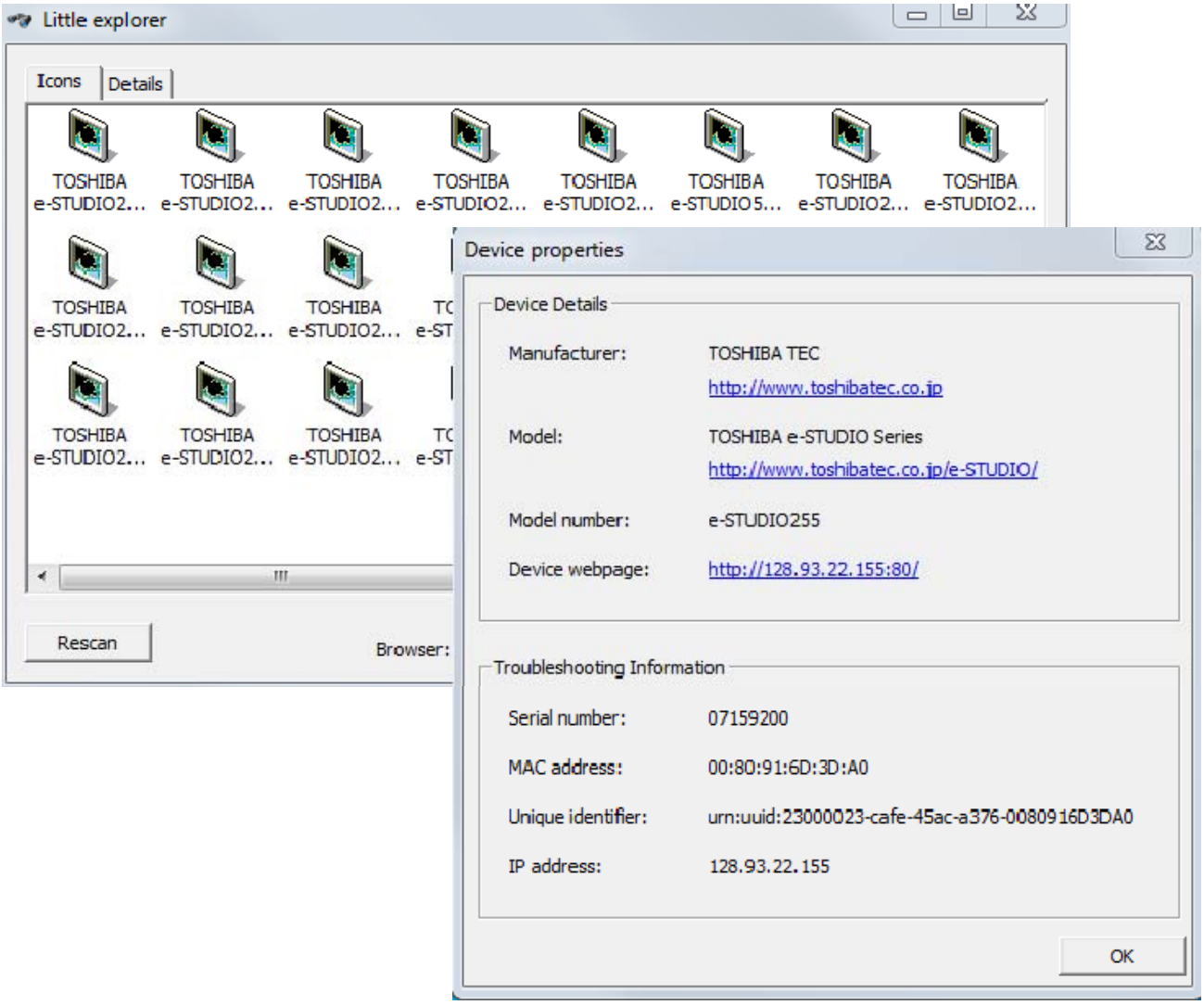

Figure 93: LittleExplorer screenshot

5.4. DPWS & industrial e-Management

Industrial maintenance is gaining significance both within the academic and industrial community, as it develops from being considered a minor activity, towards a strategic task in operation management, thus being called asset lifecycle management. Production facilities still have to become more efficient and more effective. The automation and total integration of software in the operation management is necessary to be able to fully accommodate the customer's demand for flexibility. The system management of a factory can be divided into three layers:

- **The Enterprise Resource Planning (ERP)** that consists of a system that helps taking the decisions on the working of the factory in order to meet the customer's demand. ERP integrates internal and external information across the entire organization about the finance, sales, decisions for maintenance... The plant manager has to be informed about what happens in the factory in order to take the right decisions and then propagate his decision to the factory.

- **The Supply Chain Management (SCM),** is “the management of a networked interconnected business involved in the ultimate provision of product and service packages required by end customers” (Harland [100]). Supply management spans all movement and storage of raw materials, work-in process inventory, and finished goods from point of origin to point of consumption. Applications at this level command globally the factory and monitor it.
- **The Distributed Control System (DCS)** is the system composed of devices that performs the supply chain. This level corresponds to an integration of the devices and the software to control the factory. It allow also to exploit for example sensors to monitor the factory.

The challenge is to establish an efficient communication between these layers, in order to make the factory agile. However, in order to implement effectively e-Maintenance application for asset lifecycle management at each of these levels, several requirement need to be met. One major need is a platform that can fully support e-maintenance practices, taking also into consideration the latest concepts and technology trends such as the SOA approach at device level, including DPWS technologies for example.

The future factory will be dominated by SOA, as a new architecture which empowers new capabilities and enables the realization of sophisticated approaches based on the collaboration of devices, network services within a single enterprise and among several enterprises. Often integrated devices can have embedded intelligence and sensing/actuating capabilities to facilitate the maintenance. The use of SOA paradigm at the device level enables the adoption of a unifying technology for all level of the enterprise, from sensors and actuators up to enterprise business process. This will lead to information being available “on demand” and allow business-level applications to use high-level for such purposes as diagnostics, traceability and performance indicators – resulting in increased overall equipment effectiveness and business agility.

The SOCRADES project [101] aimed at creating and integrating new methodologies, technologies and tools for the modelling, design, implementation and operation of networked systems made of smart embedded devices. The SOCRADES project is a platform for next-generation industrial automation systems that exploits SOA paradigm. It has developed a Service-Oriented Cross-Layer infrastructure for Distributed smart Embedded devices. The figure 94 illustrates the goals expected in the SOCRADES project. SOCRADES provides mainly the following modules:

- The **SOCRADES Integration Architecture (SIA)** that realizes an effective integration of the capabilities offered by the Internet of Things by exposing real-world devices with embedded software to standard IT systems by making them accessible in a service-oriented way. The implemented architecture hides the heterogeneity of hardware, software, data formats and communication protocols that is present in today’s embedded systems. SIA allows seamless interaction among devices and services hosted at different layers (e.g. at device, network or enterprise system) to perform monitoring and management function at the device level.
- The **SOA-based E-Maintenance platform** [102] that coordinates timely maintenance information shared among different actors (devices, plant managers, external partners, business managers, decision support systems, etc) and provides the basic tools for decision to be made. As depicted in the figure 39, the e-maintenance platform provides easily its functionalities thanks to the usage of the SOCRADES Integration Architecture. The monitoring and management functionalities provided by the SIA are used to be combined in more sophisticated service behaviour. As an example, device status can be monitored and the e-maintenance platform has a direct mapping between the business logic hosted in the device and the whole

business process. So the platform knows which part of the process is affected, and automated ticket can be issued (e.g. remote evaluation of the health status, exchange of the device, or even an order to the ERP system for a repair task to the nearest worker in the field). In addition this module exploits the dynamic discovery feature enabled by DPWS devices and services, in order to help prevent or timely identify conflicts that otherwise would be discovered only after a problem arises and halts the production. This module focuses also on P2P communications among devices. This allows decentralization of management, and enhances collaborative scenarios where several devices cooperate for decision making for example.

- Tools and frameworks for the modelling and simulation of industrial systems and their monitoring. These elements can be integrated in the factory environment according to the needs, as they participate to provide an ecosystem for the management of the factory.
    - **Modelling frameworks**:
        - A framework for modelling consistently embedded software and systems with a component-based approach [105]. This framework addresses two issues: (i) the tight integration between hardware and software, typical for embedded systems, make it virtually impossible to model and implement software separately from hardware (ii) it is difficult to express timing requirements, an intrinsic part of functionality of many embedded systems, in dataflow abstractions traditionally used in component-based design. This framework provides firstly a software design methodology for the conception of the embedded system from the product specification to the final product. The system is specified step by step, where a step refined the previous at a lower level. Each step integrates the timing requirements that are then verified. Secondly they provides the Timber language as an implementation approach. A service
        - Modelling language and monitoring engine [106]. This language addresses the increasing need to predict, monitor and manage the quality of service that is delivered to users. It aims at leading with large number of object instances with calculated key performances indicators, including indicators which are calculated over time intervals and late arrival of data. It aims also to present result in an intelligent way that allows for example looking backwards and forwards to understand the evolution of the key performances, and personalize them according to the viewer role. This language is an object-oriented functional language tailored to the domain-specific requirements. Finally they provides a runtime environment for calculating the service status.
    - **Simulation tool**: A simulator for a system based on DPWS devices [104]. It aims to simulating the discovery and the cooperation of large number of distributed networked embedded devices in a peer-to-peer way. This simulation tool is based on multi-agents to represents the devices. In particular, the tool considers two types of agents: DC-Agents which corresponds to DPWS clients, and DS-Agents which corresponds to DPWS server. This tool is integrated between the business applications that are in the ERP layer, and the device layer of the enterprise.
    - **Decision system** for automation component based on Petri Net [107]. At design time, the system is modelled by specifying, thanks to Petri Nets, in terms of how the system should be controlled and how processes are

executed. But complex systems are dynamic and stochastic, and not all parameters can be considered during this phase. Additionally, a system may be defined with different branches and choices possibilities. Processes must becomes more flexible in order to cope with constant product changes and increasingly volatile demand. In another hand, Petri Nets have a good matching to modelling the targeted systems characterized by adoption of decentralized distributed automation, with manufacturing resources composed of intelligent modules. This collective functionality replaces the logical programming of manufacturing sequences and supervisory functions in traditional production systems. All these complex situations at runtime require a complement mechanism to decide over the application, through the Petri Net model of the system. This decision support addresses the following needs:

- Resolution of conflicts by selecting the firing transition between several ones.
- Behavioural and structural analysis
- Selection of the best service corresponding to a transition
- Management of Petri Net to start and stop it.
- Automatic composition and aggregation of sub-Petri Nets that represent the behaviour of the constituting devices.

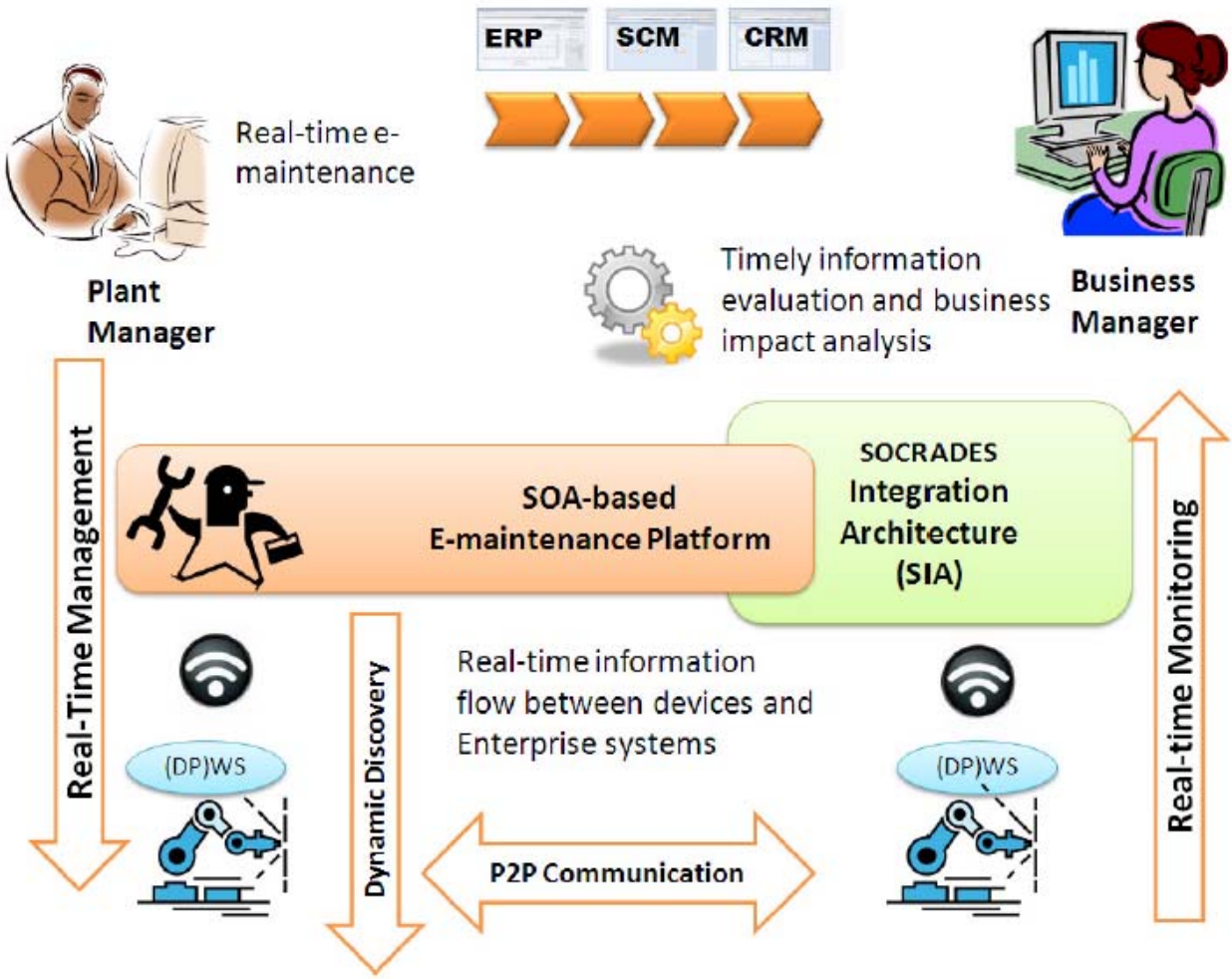

Figure 94: SOCRADES Integration Architecture overview [102]

The direct support for device with the DPWS technologies are principally addressed in the SOCRADES Integration Architecture. Thus we are going to detail a bit this part of the SOCRADES contribution. The figure 95 shows the overall architecture of the SIA module, presented in [103].

The SIA is composed of the following layers:

- The **Devices layer** contains the devices. They can be of different types, from DPWS devices to RFID for example.
- The **Platform Abstraction layer** hides the heterogeneity of the devices to control and deploy services on them. The Device Service Proxy integrates the protocols that correspond to each type of devices. The Device Service Injector is responsible for deploying or un-deploying a given service on a device. The Enterprise Service Proxy Factory allows creating virtual devices that acts as proxies between the device layer and upper layers.
- The **Security layer** verifies that devices and beck-end services are accessed only by clients that have a certain role. This layer support credentials and authentication between the device layers and platform layers and the upper layers.
- The **Device Management layer** is responsible for the management of the devices and the hosted service by maintaining an inventory on them and manages their lifecycle. Monitoring and Inventory listens events and performs active and passive search with the discovery sub-component. Thanks to these inputs, this component maintains a database of the devices and their metadata, QoS data. In another hand, the Service Lifecycle management component manages the services that are hosted on the devices and can for example ordering the deployment of a service on a device. A database is maintained to know the mapping between devices and hosted services.
- The **Service Management layer** aims to support the composition of business process by running processes that are expressed for example in underspecified BPEL for more flexibility. It also manages, through the service monitor component, a repository of all the currently connected hosted service instances by storing their fast-changing, live and up-to-date data.
- The **Application Interface layer** enables enterprise-level applications to interact with and consume data from a wide range of networked devices using a high-level, abstract interface that features Web Services standards. This level features a messaging system for eventing and invocation handling. The eventing support is based on WS-Notification and can act as a broker to intercept subscriptions and propagate notifications. In the same manner, the invocation handler is an intermediary for service invocations and is capable of buffering an invocation in case an endpoint is not available. Additionally, it provides a service catalogue that allows to querying for services. It finds and delivers services description and EPR to running service instances.

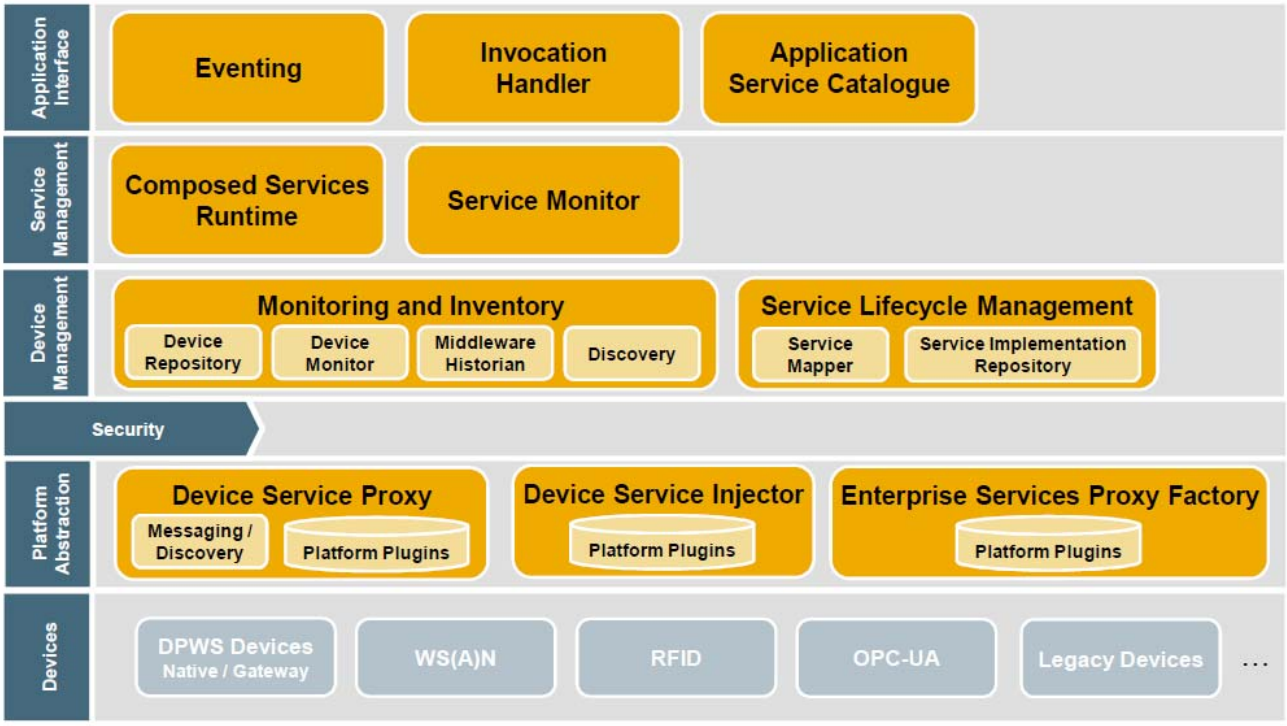

Figure 95: SOCRADES Integration Architecture overview

In the SOCRADES project, several prototypes have been implemented as a proof of concept:

- A robotic gripper is coupled with a wireless temperature sensor (e.g. a SunSPOT). This installation is linked to a workstation that allows monitoring the robotic arm temperature. Firstly the GUI shows the actual temperature and the history as a chart. Secondly the stopping of the gripper is automated when the gripper temperature is higher than a threshold. The gripper installation is connected to a Programmable Logic Controller that stops the gripper. Then the GUI allows to edit graphically some rules to make the controller stop the gripper when the SunSPOT detects that the temperature is too high. Additionnaly, this case raises an event that is captured by the GUI. Then it can present a report of all the event that occurred. More globally, considering that this installation represents a factory, the GUI is able to present a report that shows the status of all the factories in a region. This is done by placing icons with a color code on a google map. For example a green factory icon means that the factory is OK, and a red factory icon means that this factory has a failure. This demonstrator was presented at ITEA2 Symposium in 2007. Another prototype of this type was presented at ITEA2 Symposium 2008.
- A prototype that integrates the decision system for Petri Nets was realized to schedule the routing of plastic plates by elevators and conveyor belts. In this example, an elevator can receive a plate and push up or down according to criteria. Additionally, conveyor belts can form a crossroad. The decision system has to solve possible conflicts at this crossroads and take decision for elevators. Conveyor belts and elevator are modelled as Petri Nets that are assembled to form the Petri Net of the overall system.

5.5. DPWS and Reliability

WS4Dsec [109][110] is a new 2-years project that started on January 2011. Its goal is to address the security and reliability issues raised by the conception of complex systems with the Internet of Things:

- Central authority may not be available
- The encryption and certificate cause message and computing overhead while devices are very restricted resources in term of memory and power.
- The security mechanisms face the dynamicity and interoperability of the devices and systems.

In this project, they want to provide tools for modelling devices and their constraints (e.g. security, power, memory,...) and the requirements. These models may allowing to perform formal verifications for the DPWS Security framework.

6. Conclusion

In this document we studied the DPWS specification and its related works.

We firstly showed that DPWS is a new specification that solves issues when integrating devices and software with a Plug-&-play manner in a large scale system, thanks to managed discovery mechanisms. In addition, contrary of most of the approaches to integrate devices, it presents the advantage to be independent of the language or network media. DPWS mixes the advantages of UPnP for the device integration and Web Services for the independence and the scalability.

Secondly, we presented DPWS as a set of the following elements:

- A specification format to describe devices metadata
- A subset of Web Service standard to support the discovery, the exchange of metadata, the eventing and the security.
- A set of implementation constraints to deal with the resources constrained devices and their implications

Thirdly we presented 7 implementations that allow developing DPWS services. We concluded that JMEDS, DPWS-Core and µDPWS provides interesting and complementary features for working on a large set of common devices, and supporting more dynamicity and scalability.

Fourthly we presented research work and project that are based on DPWS. Today 3 communities are particularly actives with DPWS: Microsoft works to enhance the Plug and Play in Windows Vista/7 for networked devices. In another side, the WS4D and SOAD initiatives, born from SIRENA, the first DPWS project, continue similar goals: the development of an ecosystem to enhance the use of the DPWS implementations, the integration of DPWS with OSGi and the e-Management of DPWS devices.

Bibliography

[1] Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisionning of Web Services. D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, D. Savio. IEEE Transactions on Services Computing. Vol 3 – No 3. July-September 2010.

[2] Tahoe II development platform. http://devicesolutions.net/Products/TahoeII.aspx

[3] SunSpot. http://www.sunspotworld.com/

[4] OSGi Alliance. http://www.osgi.org/About/Technology.

[5] The Network on Wheels – Java Developers, Start your Programming Engines!. Steven Meloan. August 4, 2005. Available at http://java.sun.com/developer/technicalArticles/J2ME/TLA.html

[6] OSGi Markets and Solutions. OSGi Alliance. http://www.osgi.org/Markets/HomePage.

[7] How OSGi Changed My Life. Peter Kriens. March 4, 2008. Queue vol 6, No 1. ACM Digital Library.

[8] Apache Felix. http://felix.apache.org/site/

[9] Apache CXF – Distributed OSGi. http://cxf.apache.org/distributed-osgi.html

[10] HAVi: Home Audio Video Interoperability. Jussi Teirikangas. Technical report, Helsinki University of technology, 2001.

[11] Jini Architecture Specification. Sun Microsystems. 2001.

[12] Device Architecture. http://www.jini.org/wiki/Device_Architecture.

[13] UPnP Device Architecture 1.1. UPnP Forum. 15 October 2008. http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf

[14] UPnP in Integrated Home and Building Networks. R. Kistler, S. Knauth, A. Klapproth.

[15] Web Service Architecture. W3C. 2004. http://www.w3.org/TR/ws-arch

[16] Apache Axis. http://ws.apache.org/axis/

[17] Apache CXF. http://cxf.apache.org

[18] .NET Framework Developer Center. Microsoft. http://msdn.microsoft.com/en-us/netframework/aa663324

[19] IBM Web Services. IBM. http://ibm.com/fr/web-service.

[20] Hypertext Transfer Protocol – HTTP/1.1. R. Fielding and al. June 1999. http://www.ietf.org/rfc/rfc2616.html

[21] SOAP Version 1.2 Part 1: Messaging Framework. M. Gudgin and al. April 2007. http://www.w3.org/TR/soap12

[22] Web Service Description Language (WSDL) 1.1. E. Christensen and al. March 2001. http://www.w3.org/TR/wsdl

[23] Device Profile for Web Service. Version 1.0. February 2006. http://schemas.xmlsoap.org/ws/2006/02/devprof

[24] Device Profile for Web Service. Version 1.1. OASIS standard. July 2009. http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.pdf

[25] SIRENA. 2005. http://www.sirena-itea.org.

[26] SIRENA – Service Infrastructure for Real-time Embedded Network Devices: A service oriented framework for different domains. H. Bohn, A. Bobek, F. Golatowski. In International Conference on Networking (ICN). 2006.

[27] Web Services for Devices (WS4D) initiative. http://www.ws4d.org

[28] SOA4D. http://cms.soa4d.org

[29] Materna Information & Communications. http://www.materna.com

[30] M. Gudgin and al. SOAP Version 1.2 Part 1: Messaging framework. http://www.w3.org/TR/soap12. June 2003

[31] H. Combs and al. SOAP-over-UDP. September 2004. http://schemas.xmlsoap.org/ws/2004/09/soap-over-udp

[32] T. Nixon and al. SOAP-over-UDP version 1.1 – OASIS Standard. July 2009. http://docs.oasis-open.org/ws-dd/soapoverudp/1.1/wsdd-soapoverudp-1.1-spec.html

[33] E. Christensen and al. Web Service Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl

[34] D. Box and al. Web-Service Addressing (WS-Addressing). August 2004. http://www.w3.org/Submission/ws-addressing/

[35] M. Gudgin and al. Web Services Addressing 1.0 – Core. May 2006. http://www.w3.org/TR/ws-addr-core/

[36] J. Beatty and al. Web Service Dynamic Discovery (WS-Discovery). April 2005. http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf

[37] Oasis standard. Web Service Dynamic Discovery (WS-Discovery). July 2009. http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01

[38] J. Alexander and al. Web Service Transfert (WS-transfert). September 2004. http://specs.xmlsoap.org/ws/2004/09/transfer/WS-Transfer0904.pdf

[39] J. Alexander and al. Web Service Transfert (WS-Transfert). http://www.w3.org/Submission/WS-Transfer/

[40] K. Ballinger and al. Web Service Metadata Exchange (WSMetadataExchange) September 2004. http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange0904.pdf

[41] K. Ballinger and al. Web Service Metadata Exchange 1.1 (WSMetadataExchange). August 2008. http://www.w3.org/Submission/WS-MetadataExchange/

[42] D. Box and al. Web Service Eventing (WS-Eventing). August 2004. http://specs.xmlsoap.org/ws/2004/08/eventing/WS-Eventing.pdf

[43] D. Box and al. Web Service Eventing (WS-Eventing). March 2006. http://www.w3.org/Submission/WS-Eventing/

[44] A. Nadalin and al. Web Service Security: SOAP Message Security 1.0 (WS-Security 2004). March 2004. http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[45] A. Nadalin and al. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). February 2006. http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

[46] S. Bajaj and al. Web Service Policy Framework (WS-Policy). September 2004. http://specs.xmlsoap.org/ws/2004/09/policy/ws-policy0904.pdf

[47] A.S. Vedamuthu and al. W3C Recommendation. Web Service Policy 1.5 – Framework. September 2007. http://www.w3.org/TR/ws-policy/

[48] WSDAPI for .NET. Microsoft msdn. http://msdn.microsoft.com/en-us/library/ff793745(v=WinEmbedded.1001).aspx

[49] The .NET Micro Framework Plateform SDK. Microsoft msdn. http://msdn.microsoft.com/en-us/library/cc533015.aspx

[50] Web Service on Devices Interfaces. Msdn library. http://msdn.microsoft.com/en-us/library/bb881989.aspx

[51] C. Miller, L. Tessiore. Introducing: NET Micro Framework. Embedded Systems Conference. Boston. September 2006. http://www.emacinc.com/Public/info_sheets/MicroFramework_ESC%20Presentation.pdf

[52] Web Service on Devices Reference. Msdn library. http://msdn.microsoft.com/en-us/library/aa386288(v=vs.85).aspx

[53] Web Service on Device. Msdn library. http://msdn.microsoft.com/en-us/library/aa826001(v=VS.85).aspx

[54] WSDAPI – Class library. Msdn library. http://msdn.microsoft.com/en-us/library/ff793811(v=WinEmbedded.1001).aspx

[55] Use WSDAPI for .NET. msdn library. http://msdn.microsoft.com/en-US/library/ff793722(v=WinEmbedded.1001).aspx

[56] API Reference for .NET Micro Framework. Msdn library. http://msdn.microsoft.com/en-us/library/ee435793.aspx

[57] .NET Micro Framework Platform SDK. Msdn Library. http://msdn.microsoft.com/en-us/library/ee436350.aspx

[58] Quick Start Guide. Msdn Library. http://msdn.microsoft.com/en-us/library/ee435412.aspx

[59] Configuring WSDAPI Applications to Use a Secure Channel. Msdn library. http://msdn.microsoft.com/en-us/library/aa823078(v=VS.85).aspx

[60] Securing Communications with SSL and the .NET Compact Framework. Msdn library. http://msdn.microsoft.com/en-us/library/bb738067.aspx

[61] Apache Licence, Version 2.0. The Apache Software Fondation. Januray 2004. http://www.apache.org/licenses/LICENSE-2.0.html

[62] Contiki – The Operating System for Connecting the Next Billion Devices – the Internet of Things. http://www.sics.se/contiki/

[63] udpws – DPWS for highly resource-constrained devices. http://code.google.com/p/udpws/

[64] The BSD Licence. Open Source Initiative. http://www.opensource.org/licenses/bsd-license.php

[65] Basic Profile Version 1.0. WS-I. April 2004. http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html

[66] Web Service for Devices – WS4D. http://ws4d.e-technik.uni-rostock.de/

[67] JMEDS. Web Service for Devices. http://ws4d.e-technik.uni-rostock.de/jmeds/

[68] JMEDS. Sourceforge. http://sourceforge.net/projects/ws4d-javame/

[69] Eclipse Public Licence – v 1.0. http://www.eclipse.org/legal/epl-v10.html

[70] Apache Savan. Apache Software Foundation. http://ws.apache.org/savan/

[71] WS4D-Axis2 – trac. http://trac.e-technik.uni-rostock.de/projects/ws4d-axis2

[72] Apache Rampart – Axis 2 Security Module. The Apache Software Foundation. http://axis.apache.org/axis2/java/rampart/

[73] Handling Binary Data with Axis2 (MTOM/SwA). The Apache Software Foundation. http://axis.apache.org/axis2/java/core/docs/mtom-guide.html

[74] User Documentation. WS4D-Axis2. http://trac.e-technik.uni-rostock.de/projects/ws4d-axis2/wiki/userdoc

[75] WS4D-gSOAP. Trac. http://trac.e-technik.uni-rostock.de/projects/ws4d-gsoap

[76] The gSOAP Toolkit for SOAP Web Services and XML-Based Applications. http://gsoap2.sourceforge.net/

[77] SOAP/XML Web Service Performance. http://www.cs.fsu.edu/~engelen/soap.html

[78] SOAP Benchmarks. http://grid.cs.binghamton.edu/projects/soap_bench/

[79] WS4D-gSOAP 0.8. http://trac.e-technik.uni-rostock.de/projects/ws4d-gsoap/chrome/site/ws4d-gsoap-refman-0.8.pdf

[80] GNU Lesser General Public Licence. GNU. Version 3, June 2007. http://www.gnu.org/copyleft/lesser.html

[81] FOX Board G20. ACME Systems. http://www.acmesystems.it/

[82] NetBurner. http://www.netburner.com/

[83] DPWS Core project. SOA4D forge. https://forge.soa4d.org/projects/dpwscore/

[84] DPWS Core – Abstract layer reference guide. Version 2.3. https://forge.soa4d.org/docman/view.php/8/117/DPWSCore+AL+guide+2.3.pdf

[85] DPWS4J. SOA4D forge. https://forge.soa4d.org/projects/dpws4j/

[86] JDOM. http://www.jdom.org/

[87] Java Architecture for XML Binding (JAXB). http://jaxb.java.net/

[88] SOAP Message Transmission Optimization Mechanism (MTOM). W3C Recommandation January 2005.

[89] OpenSSL. http://www.openssl.org/

[90] Amigo homepage. http://www.hitech-projects.com/euprojects/amigo/index.htm

[91] OSGi et le projet IST Amigo. A. Gérodolle, A.Bottaro. Atelier de travail OSGi 2006. Colocated with UbiMob. 2006. http://andre.bottaro.pagesperso-orange.fr/papers/Gerodolle-OSGiAmigo-Ubimob06.pdf

[92] RFP 86 – DPWS Discovery Base Driver. OSGi Alliance. Draft. http://andre.bottaro.pagesperso-orange.fr/papers/rfp-86-DPWSDiscoveryBaseDriver.pdf

[93] Home SOA: Composition contextuelle de services dans les réseaux d'équipements pervasifs. A. Bottaro. Orange Labs & Laboratoire d'informatique de Grenoble. Décembre 2008. http://andre.bottaro.pagesperso-orange.fr/papers/Bottaro-thesis-vfinale.pdf

[94] ITEA ANSO – Autonomic Networks for SOHO users. http://anso.vtt.fi/results.html

[95] ITEA SODA – Service Oriented & Delivery Architecture. http://www.soda-itea.org/

[96] Service Oriented Architecture for Heterogeneous and Dynamic Sensor Networks. J. Leguay, M. Lopez-Ramos, K. Jean-Marie and V. Conan. In proceedings of the second international conference on Distributed event-based systems (DPFS08). ACM. New-York. 2008.

[97] Towards the Web of Things: Using DPWS to Brige Isolated OSGi Platforms. O. Dohndorf, J. Krüger, C. Fiele, A. Litvina, I. Lück and F. Stewing. First International Workshop on the Web of Things. 2010. http://www.webofthings.com/wot/2010/pdfs/151.pdf

[98] OSAmI – Open Source Ambient Intelligence. http://thewiki4opentech.org/index.php/Portal:OSAmI#Consortium

[99] WS4D-PipesBox. WS4D initiative. http://ws4d.e-technik.uni-rostock.de/pipesbox/

[100] Supply Chain Management, Purchasinf and Supply Management, Logistics, Vertical Integration, Materials Management and Supply Chains Dynamics. C.M Harland. Slack, N (ed.) Blackwell Encyclopedic Dictionary of Operation Management. UK.

[101] SOCRADES -- Service-Oriented Cross-layer infRAstructure for Distributed smart Embedded devices. 2006-2009. http://www.socrades.eu

[102] Dynamic E-Maintenance in the era of SOA-Ready device dominated industrial environments. A. Cannata, S. Karnouskos and M. Taish. IMS – Manufacturing Technology Platform (M4SM) workshop, World congress on Engineering Asset Management (WCEAM). Athens. Greece. Spetmber, 2009. http://www.socrades.eu/Documents/objects/file1259605112.65

[103] SOA-based Integration of the Internet of Things in Enterprise Services. P. Spiess, S. Karnousklos and al. In proceeding of the 2009 IEEE International Conference on Web Services. IEEE Computer society Washington, DC, USA. 2009. http://www.socrades.eu/Documents/objects/file1259604734.51

[104] An agent-based simulation of SOA-ready devices. S. Karnousklos, M.M.J Tariq. In Proceedings of the Tenth International Conference on Computer Modelling and Simulation. IEEE Computer Society. 2008. http://www.socrades.eu/Documents/objects/file1202983364.62

[105] Enabling Component-Based Design for Embedded Real-Time Software. J. Wiklander and al. Journal of Computers, vol 4, No 12, pp 1309-1321. December 2009. http://www.socrades.eu/Documents/objects/file1259601630.86

[106] SALmon – A service Modeling Language and Monitoring Engine. V. Leijon, S. Wallin and J. Ehnmark. In SOSE'08 Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering. IEEE Computer Society. 2008. http://www.socrades.eu/Documents/objects/file1259601146.76

[107] Decision Support System for Petri Nets Enabled Automation Components. J. Pinto and al. 7th IEEE International Conference on Industrial Informatics. August 2009. Pp 289-294. http://www.socrades.eu/Documents/objects/file1259604353.99

[108] Generic Management Services for DPWS-enabled devices. G. Cândido, J. Barata and A.W. Colombo. 35th Annual Conference of the IEEE Industrial Electronics Society (IECON 2009), Porto, Portugal, 3-5. Novmeber 2009. http://www.socrades.eu/Documents/objects/file1259605647.07

[109] WS4Dsec – Reliably Secure Web Services for Devices. Universität Rostock. 2011-2012. http://www.imd.uni-rostock.de/index.php?id=ws4dsec

[110] WS4Dsec talk Kickoff. S. Pfeiffer and A. Lehmann and al. http://www.slideshare.net/correctsystems/ws4-dsec-talk-kickoff-rs3