FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**Departamento de Engenharia Electrotécnica e de Computadores**

# Bio-Inspired Anatomy for Autonomous DPWS-Compliant Automation Components

José Eduardo Bruno de Sousa

Dissertação apresentada na Faculdade de Ciências e Engenharia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Engenharia Electrotécnica e de Computadores

Orientador: Prof. José Barata

Lisboa

Fevereiro de 2009

# Acknowledgements

This work is the last movement of one more step of my long journey on studies.

Among all my university colleagues, there were some who I had the pleasure of work with, and within these, there were some who really helped me on this journey. I would like to thank here all of them, with no special order: David Rodrigues, Rodrigo Guerreiro, Nuno Garrido, Paulo Bonifácio, João Oliveira, Carlos Barros, David Valério, Filipe Lopes, José Serra and Ricardo Mendes.

However, one special colleague was Alexandre Rodrigues, who, besides helping me at university, came with me to Schneider-Electric GmbH, and helped me with not only my work, but also programming a higher-level application that uses my work demonstrating it.

Also, I would like to thank all my colleagues at Schneider-Electric GmbH, as they were my family there, again with no special order: Daniel Cachapa, J. Marco Mendes, Nuno Barral, Axel Bepperling, Natalyia Popova, Viktor, Thanh Pham, Dimitri and Martin.

I must also send a very special thanks to all my teachers, who, some more, others less, cared about my work as a student. A particular word of gratitude goes to my supervisor Prof. José Barata, whose support and motivation were fundamental, as was the opportunity to realize my training/dissertation at Schneider-Electric GmbH.

My experience at Schneider-Electric GmbH was not possible, however, without the authorization and support of Armando Colombo and Ralf Neubert, who had always time to oversee the progress of my work despite having very busy work schedules.

There is also a group of friends who I must show my gratitude, as they always cared about my studies even being nothing to do with it: David Freitas, Tânia Bragança, Ana Lucia, Diogo Santos, Pedro Dias, Pedro Leiria, Janete Faustino, Pedro Tomaz e Ana Maria.

However, the biggest debt I have goes to my family. They always supported me, principally in the most difficult times, and I know I can always count on them.

# Resumo

Esta dissertação aborda o uso da tecnologia DPWS para implementação de web-services em dispositivos, explica as suas limitações e apresenta uma arquitectura para ultrapassá-las.

Foi estudada uma arquitectura baseada em DPWS com o objectivo de tornar um dispositivo autónomo, chamada de DPWS Simples. Esta arquitectura é baseada na programação por módulos, sendo o esqueleto suportado por dois módulos obrigatórios, o módulo de comunicação e o Event Router-Scheduler.

O módulo de comunicação controla a comunicação para o exterior, enquanto o ERS é responsável pela comunicação interna, em tempo real, entre os módulos.

O toolkit do DPWS não oferece possibilidade de interagir com serviços que emergem em run-time. Foi necessário implementar algumas melhorias de modo a permitir ao DPWS ser mais dinâmico. Este novo serviço foi chamado Serviço Dinâmico.

Foi realizada uma experiência fazendo ligação entre um serviço do DPWS e o mesmo serviço mas criado dinamicamente. Foi usado o exemplo das lâmpadas, que consiste em ligar e desligar uma lâmpada, e obter o seu estado. Uma interface gráfica foi implementada para a aplicação ter fácil utilização. Os resultados foram satisfatórios, pois a experiência funcionou plenamente.

# Abstract

This thesis approaches the use of the DPWS technology to implement web-services on small devices, addresses its limitations, and explains an architecture to solve it.

An approach to an autonomous device’s simple architecture was realized, using DPWS, and was called Simple DPWS. The objective was to implement/simplify some features in a device in a way that the device can work on its own. The designed architecture is based on that each component has its framework of modules, having always at least the skeleton modules communication and Event Router-Scheduler.

The communication module controls all the communication between the devices and the ERS is the responsible for the other modules’ real-time communication.

The DPWS toolkit offers no capability of interacting with run-time-appearing services. Thus there was a necessity to do enhancements over the DPWS toolkit to have a dynamic stub and skeleton. This service was called the dynamic service.

An experience was done connecting a DPWS toolkit sample service with the corresponding hand-created dynamic service. It was used the lighting service that consists on turning a lamp ON or OFF and getting its status. A GUI was done for the application to be more user-friendly. The results were satisfactory, as the connection worked.

# Glossary of Abbreviations

DPWS - Devices Profile for Web Services

ERS - Event Router-Scheduler

GUI – Graphical User Interface

HAVi - Home Audio/Video Interoperability

HTTP - Hypertext Transfer Protocol

I/O - Input/Output

ICT - Information and Communication Technologies

IMS - Intelligent Manufacturing Systems

IP - Internet Protocol

ISC - International Security Controls

JINI - Java Intelligent Network Infrastructure

K/BIS - Kitchen and Bath Show

LAN -Local Area Network

MAS - Multi-Agent Systems

MTOM - Message Transmission Optimization Mechanism

OASIS - Organization for the Advancement of Structured Information Standards

OSGi - Open Service Gateway Initiative

PC - Personal Computer

PLC - Programmable Logic Controller

QoS - Quality of Service

RMI – Remote Method Invocation

SoA - Service-oriented Architecture

SOA4D - SoA for Devices

SOAP - Simple Object Access Protocol

TCP - Transmission Control Protocol

UDP - User Datagram Protocol

UPnP - Universal Plug And Play

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

WS4D - Web Services for Devices

WSDL - Web Services Description Language

XML - eXtended Markup Language

# Table of Contents

# Table of Figures

# Chapter 1

# Introduction

## Contents

## *1.1 – Research Problem*

Modern manufacturing processes are focusing on flexibility, agility and re-configurability. Production focus is shifting from mass production to mass customization. New revolutionary manufacturing concepts are, thus, emerging. Centralized architectures aren’t capable of dealing with the new reality of decentralized agile systems. As so, next generation manufacturing must support strong market responsiveness but low costs and high quality remain vital concerns. So, new technology is required to be considerably more flexible and adaptable.

Nowadays, the communication between manufacturing components is facilitated by a central system. This structure design approach fails when trying to use an intelligent control structure. Recent technology is capable of much more of what is being used. Such technology should be integrated with new engineering solutions that will support the research and implementation of new paradigms in automation and control.

Besides this, a new approach to the enterprises is required, like intra-enterprise dynamic integration of modules and inter-enterprise dynamic cooperation.

Some different approaches had been studied, like Multi-Agent Systems and Service-Oriented Architectures. These technologies have been the subject of great attention, as they implement effectively two principles that may sound contradictory: autonomy and interoperability.

On top of Service-oriented Architectures, a new technology called Devices Profile for Web Services (DPWS) is becoming steady in some areas. DPWS is a plug-n-play protocol middleware built on top of a set of Web Services specifications. Consequently, it is a distributed architecture. It leverages TCP/IP and the Web technologies to enable seamless proximity networking in addition to control and data transfer among networked devices in the home, office, and public spaces.

DPWS objectives are similar to those of Universal Plug And Play (UPnP) but, in addition, DPWS is fully aligned with Web Services technology and includes numerous extension points allowing for seamless integration of device-provided services in enterprise-wide application scenarios.

In distributed automation and especially in the branch of production systems, the set of equipment and other components in the system may be comparable under some circumstances to a society of living beings. Taken a closer look into a component itself, its internal mechatronical organization may correspond to functional organs that are responsible for specific tasks, providing the "vital" properties to be able to fulfill its requirements. A central question is how these functional modules or “organs” may be integrated, controlled and able to pass impulses between them and therefore to form complex and operational structures.

Taking this in mind, an approach to an autonomous device was realized, using DPWS. It was called Simple DPWS (SDPWS), and is represented in figure 1.1. The main objective is to implement some features in a device in a way that the device can work on its own, owning the necessary features. This simplification and features were studied and engineered. The designed simplification is based on that each component has its framework of modules (figure 1.1). There are two kernel (obligatory) modules, the communication module and the Event Router-Scheduler module (figure 1.1), and optional modules, and the work was based on the kernel ones, as they were studied and developed.

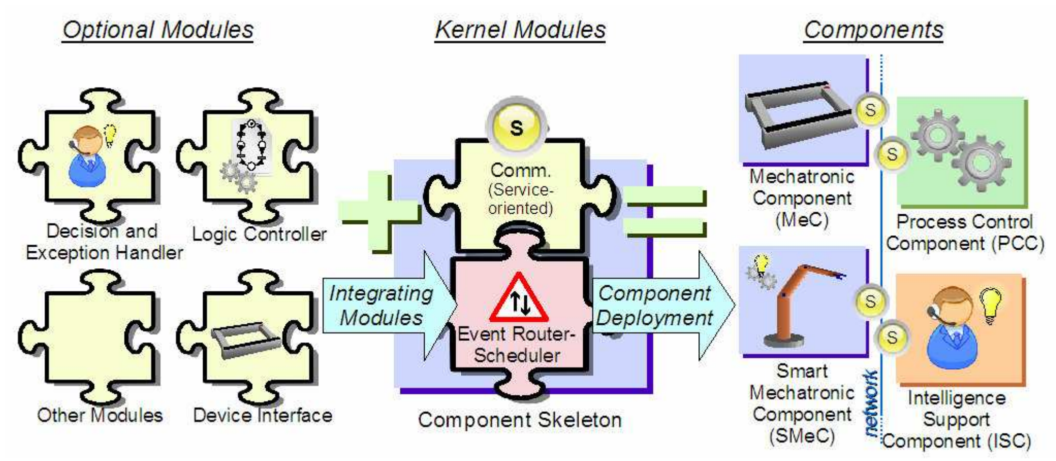


**Figure 1.1 – The SDPWS with the kernel modules**

The communication module controls all the communication between devices and, therefore, is responsible for using DPWS. The Event Router-Scheduler (ERS) is the responsible for the other modules' real-time communication. The optional modules are user-defined and adapted to the respective component's function.

The DPWS toolkit generates compact code, creating a light weight resource-constraint program structure with a high performance predictive parser for each service. However, it offers no capability of interacting with new services that appear at run-time, and new code has to be generated and compiled on the client stub to integrate the unprecedented service. Thus there was a necessity to do some enhancements over the DPWS toolkit to have a dynamic stub and skeleton that can invoke and receive any kind of message, and thus, having the possibility to create or update services in real-time, without machinery shut-down. This would, among other benefits, greatly increase the agility of the shop floor, and consequently reduce maintaining costs.

This new feature on automation components must accomplish some main objectives, like be able to read a new service at run-time and inform the other identical devices that a new service is available, as update a service and automatically send services to "new-born" identical devices. For this purpose, the component must have a built-in service-receiving and service-updating service, as also a service-spreading service, as shown in figure 1.2.

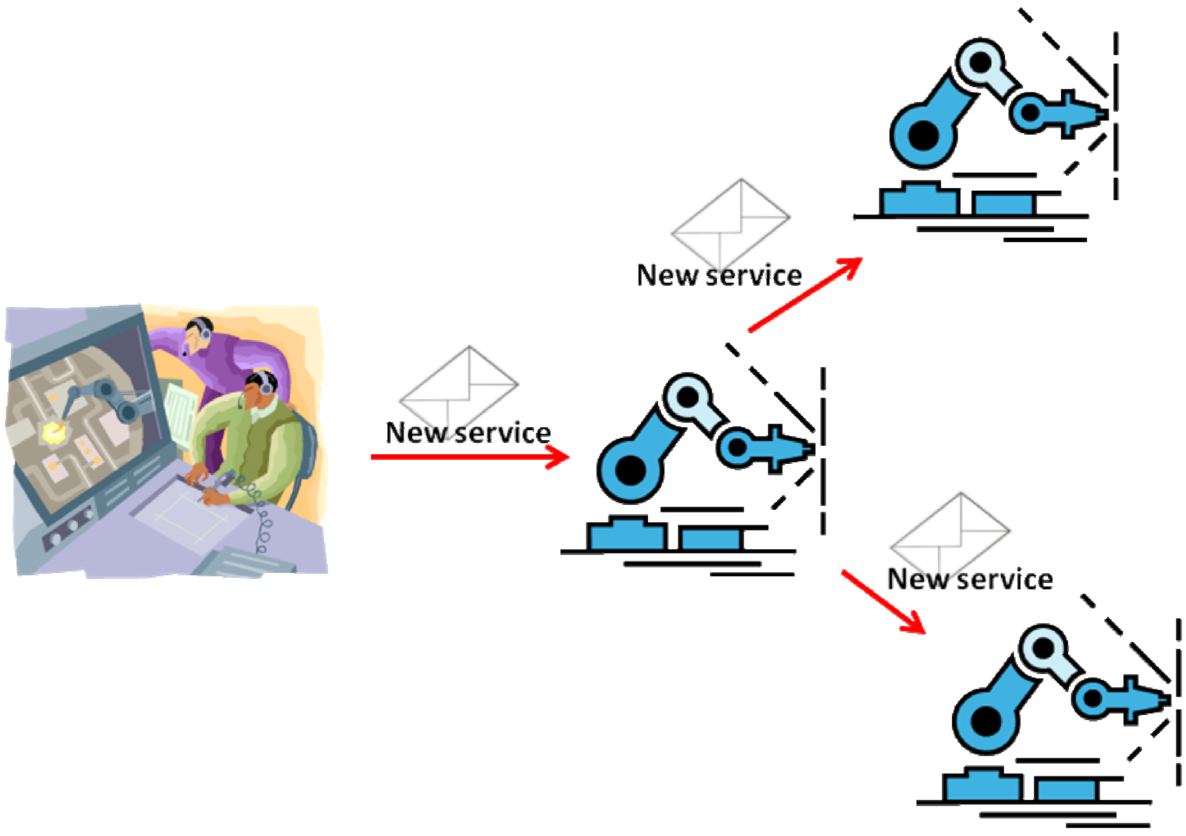


**Figure 1.2 – Real-time Service Loading and Spreading**

The dynamic service cannot, however, throw away the DPWS capacity of maintaining the code compact and light-weight, as embedded memory becomes short rapidly, and costs must be kept low.

Both Schneider-Electric colleagues and supervisors appreciated the concluded work, giving a very positive feedback.

## *1.2 – Thesis outline*

This thesis is divided in eight chapters: “Introduction”, “The State of the Art in Industry”, “Supporting Technologies”, “SDPWS – the Living Components”, “The Event

Router-Scheduler", "The Dynamic Service", "SDPWS Complete Implementation" and "Conclusion".

This introductory chapter briefly explains the problem under research and presents the outline of this work.

The second chapter, "The State of the Art in Industry", explains the aging in currently used technologies in industry, comparing it to what is possible to do, and gives an example of emerging technologies that can be implemented in this sector, Service-oriented Architecture and Multi-Agent Systems.

The third chapter, "Supporting Technologies", presents the DPWS technology, a middleware which is used on top of web-services, and allows web-services on small devices.

The fourth chapter, "SDPWS – the Living Components", presents an enhanced and simplified architecture using DPWS, called Simple DPWS (SDPWS), which was designed to add the necessary features to a device using DPWS which would turn the device independent.

The fifth chapter, "The Event Router-Scheduler", shows the implementation of the intra-application module-communication-way module of the SDPWS and explains the advantages and importance of such module in real-time multi-tasking devices.

The sixth chapter, "The Dynamic Service", exposes the DPWS' current version technological limitation of being a static, pre-compiled service-implementation, and presents a new architecture of turning those services dynamic, real-time implementable and self-spreading between similar devices.

The seventh chapter, “SDPWS – Case Study”, presents exhaustively an implementation of all of the previously mentioned architectures and technologies. The example is a little Light ON/OFF service.

The concluding chapter discusses the results of the complete implementation, and points future research directions.

# Chapter 2

# The State of the Art in the Industry

Contents

## *2.1 – The industry is changing*

Modern manufacturing processes are focusing on flexibility, agility and re-configurability. Production focus is shifting from mass production to mass customization, and technology must follow new patterns to accommodate all the requirements. As so, new revolutionary manufacturing concepts and emerging technologies are being researched to take advantage of the newest mechatronics, information and communication technologies [11].

Nowadays, any enterprise with stability must be able to change promptly and dynamically its product or service catalogue to react to any unexpected disturbances or market's new directions. Centralized architectures aren't capable of dealing with this new reality, as the emergence of decentralized systems is a very important issue, for the systems to be capable of dealing with the fast changes in the production environment. These new systems must be agile and efficient to compete in a global market. The future of manufacturing will be characterized by rapidly changing markets, pressure from competition and continuously emerging technologies. Therefore, next generation manufacturing must support strong market responsiveness. However, low costs and high quality remain vital concerns, so new technology is required to be considerably more flexible and adaptable to changing than today's technology.

Approximately one third of the cost of a manufacturing plant over its lifetime is spent on installation and setup. Another substantial part of the costs is spent on maintenance. For a plant to be adapted to new products, it must change its process flow and its machines. This situation generates high costs and time spending. Actual components are inflexible, so communication between them is hard to configure, as is porting software applications to new machines. Industry has to develop, deploy and support automated systems on a global basis in ever shorter timeframes. Furthermore, the lifecycle engineering of production machines requires complex, innovative and timely interaction between geographically distributed members of project engineering teams comprising automation suppliers, control system suppliers, machine tool builders and end-user product, process and control engineers. Collectively they have responsibility for the implementation and lifecycle support of the automated system as product and production requirements change [5].

Currently, machines are categorized according to their functionality. As they are independent and can even be brand-mixed, programming is made individually. Thus, the communication between them is facilitated by a central system. This structure design approach fails when trying to use an intelligent control structure.

There are many factors why today’s technology is surpassed:

- Increasing computational power and Ethernet are more and more available on ever smaller devices;
- SOA based on Web service technology is more and more used in the world of automation technology and is already used as a platform for communication and control;
- The entire lifecycle of a product and the equipment is considered at the planning phase;
- Development and linkage of service components, as well as the design and modeling of application and workflows are already supported by engineering tools.
- Simulation and emulation tools are available for control logic entities, but not at the same extent as those for distributed applications.

These advances should be integrated with new engineering solutions that will support the research and implementation of new paradigms in automation and control in order to bring flexibility, agility and robustness to the production lines of the future. Such a tool would have to support the production line throughout the whole production lifecycle [10].

A new technology capable of hold a new generation of industrial components and architectures is desperately needed. A wide variety of open platforms has been proposed for years in the Information and Communication Technologies (ICT), which proposes industry to look at open solutions for manufacturing plants. Although several proposals have been submitted, today’s reality shows the still dominance of old standards that fight against progress.

Besides the requirements already mentioned above, there are some requirements needing to be satisfied in order to truly have a new generation of manufacturing system:

- Intra-enterprise dynamic integration of modules, turning the whole system completely robust;
- Inter-enterprise dynamic cooperation, opening a new vision to the actual enterprises' services requested and offered
- Non-disruptive scalability through addition of resources, either hardware or software;
- Fault-tolerant, self-configuration and self-monitoring modules capable of automatic recovery.

These issues have more importance than ever to maintain productivity and competitiveness.

## *2.2 – Emerging technologies: SoA and MAS*

Some different approaches had been studied, developed and analyzed to cover the new requirements. Multi-Agent Systems and Service-Oriented Architectures are maybe the best implementation of such technology. These technologies implement effectively two principles that may sound contradictory: autonomy and interoperability.

These two technologies particularly have been the subject of great attention. However, despite their promise, they have not made significant inroads in manufacturing plants yet. The lack of widely accepted standards is resulting in variety of islands with poor scalability.

## *2.3 – The SoA technology and SoA for devices*

A Service-oriented Architecture (SoA) is as a group of services that communicate with each other, trading data and/or coordinating some activity together. This intercommunication implies the need for some means of connecting two or more services to each other.

In theory, Service-Oriented Architectures offer the potential to provide the necessary system-wide visibility and device interoperability in complex collaborative automation systems subject to frequent changes. They are proven in a business system context, and initial analysis suggests that SoA could meet the technical and business level requirements for future automation systems. SoA is basically an architectural paradigm that defines mechanisms to publish, find and bind services. Message-based communication, loose coupling and open standards characterise SoA. Those features make it particularly applicable for a global multi-vendor environment where interoperability is essential [5].

In practice, SoAs build applications out of spread software services. They typically implement functionality most humans would recognize as a service, such as filling out an online application for an account, viewing an online bank-statement, or placing an online booking or airline ticket order. Instead of services embedding calls to each other in their source code, they use defined protocols which describe how one or more services can communicate with each other. This architecture then relies on a business process to link and sequence services. This process is known as orchestration, and it allows meeting a new or existing business system requirement.

The big breakthrough areas of SoA for devices are in particular the industrial sector and home automation.

In home automation, SoA helps to bridge the heterogeneity of products and brings new opportunities for networking and interaction of diverse devices. For industry applications, SoA can be used for integration of equipment or even products themselves into the enterprise infrastructures. This allows a higher level of transparency and opens completely new ways for optimization of business processes.

Research activities in the last years have been dedicated to apply and evaluate SoA on medium sized embedded systems like embedded PC, PDAs or network routers. Development frameworks like gSOAP, Intel Authoring Tools or the aveLink suite for UPnP have been created to provide development support for SoA on these devices. For smaller embedded devices likes actuator and sensor nodes this is different. They focus on mechanisms to outsource computational load from the devices. Thus, a proxy approach is favored over direct implementation of SoA to compensate for the computational restrictions of the small devices [12].

## *2.4 – SoA and the automation industry - a society of service-oriented automation components*

Production and automation systems are heterogeneous in nature, made of different components with distinguished roles. It is therefore predictable that the specifications of those systems are moving from the traditional central-controlled manner to the corresponding distributed counterpart, assimilating the natural appearance and layout of the real system.

Thus, one promising guideline in this respect is to have a conglomerate of distributed, autonomous, intelligent, fault-tolerant, and reusable manufacturing units, which operates as a set of co-operating entities. Each entity is capable of dynamically interact with each other to achieve both local and global manufacturing objectives, from the physical/machine control level on the shop floor to the higher levels of the factory management systems [4]. This new generation of systems is referenced as Intelligent Manufacturing Systems (IMS) [17].

One of the rising solutions to adapt the majority of the concepts behind IMS into feasible principles is Service-oriented Architectures device communications using the devices.

The concept SoA has gained significant attraction in just a few years and will undoubtedly have a major impact in many branches of technology. According to [1], "*A service-oriented architecture is a set of architectural tenets for building autonomous yet interoperable systems.*" and this proposal is facing one of the challenges of IMS, namely providing interoperability between autonomous systems.

Adapting the service-orientation concepts to the automation and production "ecosystem" at the shop floor and considering the principles of IMS, a "society" of service-oriented automation components is born. Each participant in the system is referred as Service-oriented Component and in some extends, Service-oriented Automation Component (when it has automatic control duties). Components may have different roles (e.g. production, transportation and monitoring) and operate autonomously. Since services are the main guide, these components should have the need of requesting services and also the desire in providing services to the community. Services itself are a form of providing resources and actions that are shared in some circumstances, much similar to the real-life services.

Fig. 2.1 shows the basic description of a Service-oriented Component and its integration into the environment of automation and production shop floor. The given example is a component that represents a physical conveyor (**Mediator of**: *Conveyor*) and has the transportation role (**Role**: *Transportation*). Implicitly, the communication to the outside world would be via services (**Orientation**: *Services*), being able to provide and request services when needed. The integration into the IT enterprise is also reached by the service-orientation. A component has a set of tasks or activities (**Tasks**: *Transport, Monitoring,* etc.) and those may be used as services provided by the component.

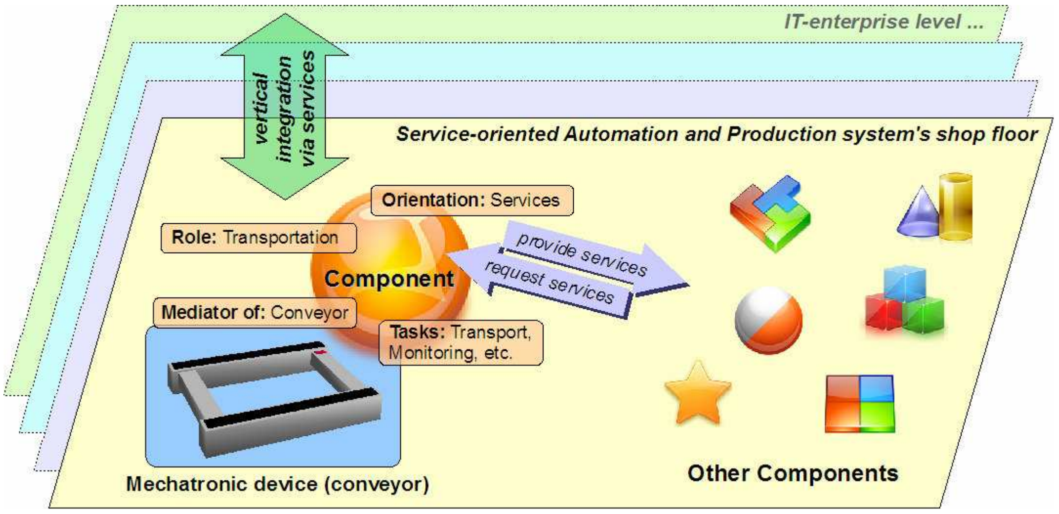

IT-enterprise level ...
vertical integration via services
Service-oriented Automation and Production system's shop floor
Orientation: Services
Role: Transportation
Component
provide services
request services
Mediator of: Conveyor
Tasks: Transport, Monitoring, etc.
Mechatronic device (conveyor)
Other Components

**Figure 2.1 – description scheme of a service-oriented component**

Interaction between components is done by the two-way service orientation, in sense of requesting and providing services. It is expected in production and automation that heterogeneous components work together for mutual benefit and global objectives. This can be distinguished as *symbiosis*, similar to the interactions between different biological species [6]. It is also possible that components may compete with each other for resources (services), but in the end the global goal must be respected.

In simple scenarios, like lighting, the use of SOA has following benefits [12]:

- Homogeneity: Heterogeneous systems become homogeneous using the notion of services. Although different vendors may provide different lighting systems, from the SoA point of view they will provide a service for switching the light on or off. Different vendors may even provide the exact same service for their devices. This allows our SoA-based switches to be used for nearly arbitrary lightings.
- Dynamics: In a device centric SoA network, services are announced when devices become available. In our setup this means that when new lighting is installed, the respective lighting service is advertised. Next time a switch is pressed it can also switch the new lamp, as it is aware of all existing lighting services.

- Self-Description: Devices are enabled to find out what capabilities other devices have using the self descriptive notion of services. This supports dynamic behavior of a system. If a switch was supposed to create a light scene for watching movies, it would dim all lights that are dimmable and switch the rest off. The lighting itself will tell the switch if it is dimmable or not.

In another example, the application case "*information management for tracking & tracing of products for recycling*", which is part of the EU funded research project PROMISE, focuses on optimization of processes in a plastics recycling facility [13]. Wireless SoA-enabled temperature sensors are used to monitor containers carrying milled plastic material that inheres the risk of self ignition. Each sensor provides a service which conveys its ability to monitor temperature to the facility's management system. The management system uses these services to assign monitoring tasks to the sensors with an appropriate threshold with respect to its current load. Respectively, sensors will inform the management system if goods enter a critical state, so that appropriate countermeasures can be initiated.

- Eventing: SOA is used to translate a physical value i.e. temperature into a system parameter. With this, the management system is now able to even react on physical events as they are now system events. In our case the violation of the temperature threshold will lead to an event based notification of the system, which then will start countermeasures.
- Dynamics: Leveraging the dynamic lookup mechanisms of SOA, the management system is aware which containers are in the monitored storage area at any given moment and can automatically react on incoming or outgoing containers.
- Distribution of responsibility: Furthermore, the intrinsic philosophy of SOA dictates that the sensor themselves are responsible to implement the actual monitoring and to

determine when a threshold has been reached. This minimizes management overhead in backend systems and prescribes adequate distribution of responsibilities.

In some situations Service-oriented Components can be seen as software agents according to the definition given by (Schoop et al.) [3], adapted from (Jennings and Wooldridge) [2], to flexible production systems:

*“An agent is considered a software entity situated in a flexible production environment, with enough intelligence that is capable of autonomous control actions in this environment and of co-operation relationships by participating in associations’ agreements with other entities in order to meet its design objectives”.*

Moreover, Multi-agent Systems [7] are of special interest since these systems bring the idea of collaborative agent society, in which each of them can take autonomous actions over their environment or over the system that they represent. On the other hand and differentiating from the agent concepts, the true meaning of service-orientation is centered in the requirement of providing services and in the necessity of requesting services by a component in the system. The real architecture, habitat and objectives of the system are truly open to the developer and thus it may adopt different strategies to cover the requirements.

## *2.5 – Existing SoA realizations for devices*

In reality, the SoA concept is not applied singly but needs a framework which provides means like protocols or data structures to bring SoA down to the implementation level. Over the years, so-called middlewares have been developed to implement different flavors of SoA.

Middlewares that are especially relevant in the area of SoA for devices are Jini, OSGi, UPnP and DPWS as they try to address the dynamic nature of device integration and/or have specifically been designed for application in this area. We will now briefly outline the main concepts of these frameworks:

1. **Jini** (www.jini.org) is a Java based solution which provides mechanisms for distributing and discovering services and supports migration of executable code from one computer to another. The core of a Jini system is a lookup-service which facilitates the search for services throughout the Jini network. Each service has to announce its existence to the lookupservice and to deposit a set of attributes as description of its features. The communication with the lookup-service is based on Java Remote Method Invocation (RMI) where the communication between server and client is defined by the drivers and may therefore use any proprietary protocol and format. Due to the RMI-based communication, Jini requires a participating device to execute a Java Virtual Machine and a respective Java application.

2. **OSGi** (www.osgi.org) is targeted on the connection of various components in home networks. It is a Java centric approach where so-called bundles disclose capabilities of a device and allow interaction via local method invocation. A central device (the gateway) provides the necessary communication platform on which the bundles are executed. Additionally, standardized mechanisms are defined to dynamically install or remove bundles and respectively discover active bundles during runtime. Maintenance/installation of the bundles can be done locally on the gateway or even remotely via the Internet. Remote communication with bundles is not supported natively. For distributed communication a mapping between OSGi bundles and UPnP devices has been defined.

3. **The Universal Plug and Play (UPnP)** (www.upnp.org) Architecture uses open and standardized protocols based on XML to describe and control devices. Information is transferred over TCP/IP and UDP/IP using high-level communication protocols like SOAP. All interaction is done on top of the IP layer and is thus completely hardware-independent. In UPnP, mechanisms for Addressing, Discovery, Description, Control, Eventing and Presentation are defined. A peer-to-peer philosophy is inherited in all these parts, so that no central component is needed to facilitate interaction among the participants of a UPnP network.

4. **Devices Profile for Web Services (DPWS)** is the approach to make the successful 'Web Services' fit for usage on the device level. DPWS combines a set of functionalities taken from the existing WS protocol suite and specifies additional protocols on top of them (WS Eventing, WS Discovery). Like Web-services, DPWS uses SOAP for message transmission and XML as data format. The projects SIRENA [14], SODA [15] and SOCRADES [16] consider the application of DPWS in the industrial sector and until now have created a DPWS stack capable to be executed on embedded devices.

# Chapter 3

# Supporting Technologies

## Contents

## *3.1 – The Device Profile for Web Services*

The Devices Profile for Web Services (DPWS) defines a minimal set of implementation constraints to enable secure Web Service messaging, discovery, description, and eventing on resource-constrained devices.

Its objectives are similar to those of Universal Plug And Play (UPnP) but, in addition, DPWS is fully aligned with Web Services technology and includes numerous extension points allowing for seamless integration of device-provided services in enterprise-wide application scenarios.

## *3.2 – DPWS description and history*

The Devices Profile for Web Services is a plug-n-play protocol middleware built on top of a set of Web Services specifications. This protocol middleware addresses discovery, description, and control of devices and services on local networks. It is a distributed architecture and leverages TCP/IP and the Web technologies to enable seamless proximity networking in addition to control and data transfer among networked devices in the home, office, and public spaces.

DPWS specification began in 2002 under the initiative of Microsoft with the aim to become the second version of the basic protocol layers of UPnP™. However, since UPnP™ devices have emerged on the market and are not interoperable with the new DPWS specification, the UPnP™ Forum does not accept the proposed roadmap.

Therefore, DPWS appears today as a competitor. Some implementations exist, like the one delivered open source by Schneider-Electric in ITEA SODA project. Microsoft well-known Vista OS hosts DPWS tools beside the UPnP™ ones.

DPWS is fully aligned on Web Services specification: WSDL 1.1, XML Schema, SOAP 1.2, WS-Addressing, WS-MetadaExchange, WS-Transfer, WS-Policy, WS-Security, WS-Discovery and WS-Eventing. It leverages lower-level Internet components, including IP, TCP, UDP, HTTP, and XML. This alignment on Web Services technologies is an opportunity to benefit from successful specifications and to simply make a bridge between local networks and the World Wide Web. On local networks, DPWS refines the Web Services specification with a specific devices profile and leverages specific local mechanisms like multicast networking. The latter is used for device and service discovery on the Local Area Network (LAN).

## *3.3 – DPWS overview*

In the last years has been noticed a convergence from user-controlled distributed systems to automatic distributed, autonomous and self-configurable systems.

The emergent technology DPWS offers the possibility to use Web Services in electronic devices, taking in consideration their constraints and implementing the most recent key needs of technology: footprint, security, plug & play, asynchronous data exchange and event-driven data exchange, among others.

DPWS permits many interactions as Discovery, which allows performing search operations and exposing operations, Eventing, which manages subscriptions between devices, Naming, allowing searching and indexing operations over data, and Description, which uses metadata to explain a device’s operations and services to other devices.

The DPWS specification was initially published in May 2004 and was submitted for standardization to the Organization for the Advancement of Structured Information Standards (OASIS) in July 2008.

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

The OSGi specification defines a service platform that relies on Java. An OSGi service is a simple Java interface but the semantics of the service are not clearly specified.

HAVi offers plug-and-play as well as Quality-of-service (QoS) capabilities and is restricted to the home domain.

JINI was developed by Sun Microsystems for spontaneous networking of services and resources based on the Java technology. Services/devices carry the code (proxy) needed to use them.

UPnP supports ad-hoc networking for devices and services and is easy to develop for. It has a very similar functionality in comparison to DPWS but does not address security issues and is only applicable for small networks (no service registry/proxy).

The big advantage of DPWS compared to all other mentioned SoAs is the reliance on Web services which implies high acceptance among developers and platform as well as programming language independence.

This technology allows devices to do a plug & play protocol when connected to the Ethernet, i.e., they know which devices are on the Ethernet, and the other devices know about it. So, any device with this technology can discover, invoke and offer services and functionalities. The concept is identical to the UPnP but it uses web-services to communicate. Support of discovery has led some to dub DPWS as "the USB for Ethernet."

There are two types of services defined by DPWS: hosting services and hosted services. Hosting services are directly associated to a device. They play an important part in the device discovery protocol. Hosted services are mostly functional, and depend on their hosting device for discovery.

## *3.4 – DPWS protocol*

DPWS is partially based on the Web Services Architecture (WSA) and uses further standards from the Web services protocol family, as seen next:

- WS-Addressing - The main objective is to provide an addressing mechanism for Web services as well as messages in a transport-neutral matter. By introducing both concepts endpoint references (EPR) and message information headers (MI) WS-Addressing overcomes the lack of SOAP's independence of underlying protocols and secondly support of asynchronous message exchange. Both limitations are historically caused by the default SOAP to HTTP binding.
- WS-Discovery - is a discovery protocol based on IP multicast for enabling services to be discovered automatically. Discovery introduces three different endpoint types: target service, client and discovery proxy. Target services are Web services offering themselves to the network. Clients may search for target services and discover them dynamically. Discovery proxy is an endpoint enabling discovery in spanned networks since simple discovery is limited to a multicast group and hence to local managed networks only. WS-Discovery defines four operations or messages to discover target services in a network. To explicitly discover target services in a network a client can use the Probe operation, send as multicast message. Matching target services will answer with the Probe Matches operation send as UDP unicast message to the client. To implicitly discover target services a client can listen for Hello and Bye messages. A target service announces its availability with these messages send as UDP multicast. To resolve logical addresses introduced with the endpoint structure in WS-Addressing a client can use the Resolve operation send as UDP multicast message. The corresponding target service responds with the Resolve Matches operation send as UDP unicast to the client. The discovery proxy does not need any additional operations.
- WS-MetadataExchange / WS-Transfer – is a specification that defines data types and operations to retrieve metadata associated with an endpoint. This metadata describes what other endpoints need to know to interact with the described endpoint. WS-

MetadataExchange defines the MetadataSection that divides the metadata into separate units of metadata with a dialect specifying its type. Until the latest version of DPWS only WSMetadataExchange was used for service and device description and retrieval. In the latest DPWS version of February 2006 WS-Transfer is used to retrieve the metadata. The structure of the metadata is still as specified in WSMetadataExchange. The main difference is that WSMetadataExchange defined operations to retrieve all or parts of the metadata of an endpoint, whereas WS-Transfer only can be used to retrieve all metadata of an endpoint. We expect that WS-Transfer and WS-MetadataExchange will be merged closer in future releases.

- WS-Eventing – defines a protocol for managing subscriptions for a Web services based eventing mechanism. This protocol defines three endpoints: subscriber, event source and subscription manager. Subscribers request subscriptions on behalf of event sinks to receive events from event sources. Subscription requests contain an event delivery mode and event filter mechanism to negotiate event delivery mechanisms and event filter mechanism. Subscription managers are responsible of holding subscriptions of event sources.

The DPWS terminology is represented in figure 3.1.

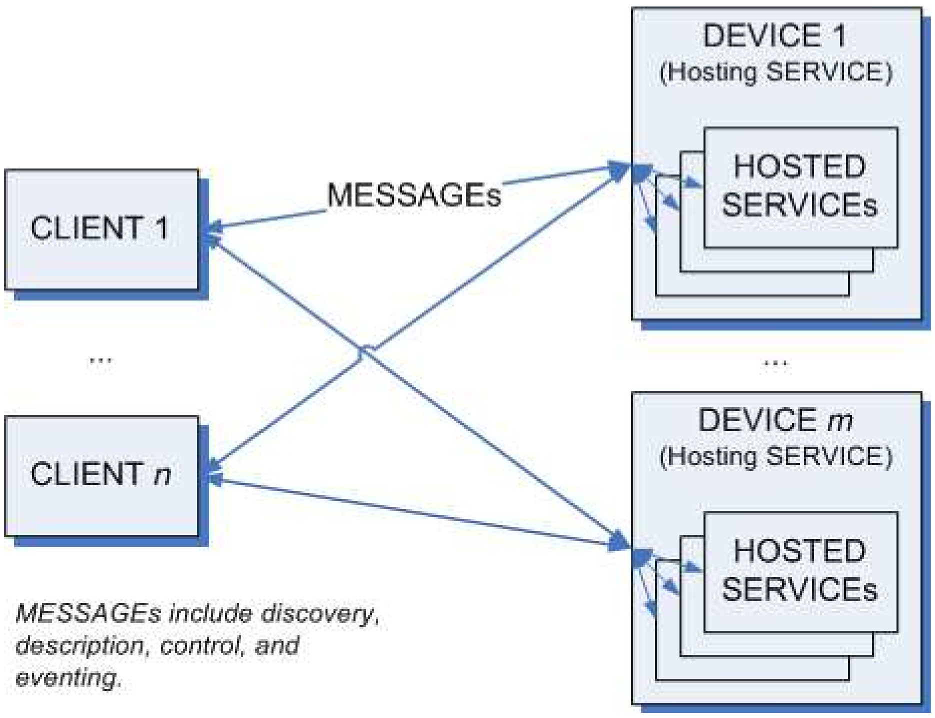


**Figure 3.1 – DPWS terminology**

Other used Web Services standards are XML, WSDL, XML Schema and MTOM. The DPWS protocol stack is shown in figure 3.2.

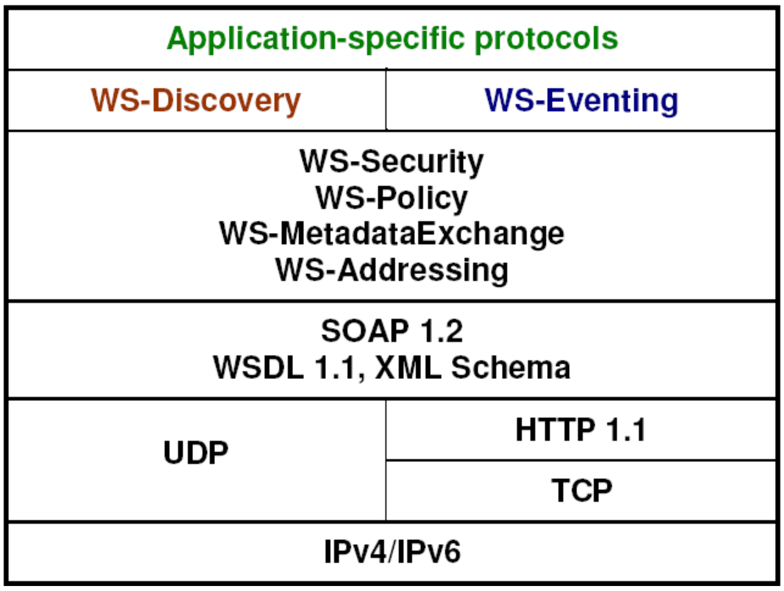


**Figure 3.2 – DPWS protocol stack**

## 3.5 – DPWS architecture principles

As said before, the DPWS specification defines an architecture in which devices can run two different types of services: hosting services and hosted services.

DPWS is built on top of the SOAP 1.2 standard, and relies on additional Web Services specifications to further constrain the SOAP messaging model.

The figure 3.3 taken from DPWS User Guide v2.9 shows the general architecture of a device compliant with DPWS.

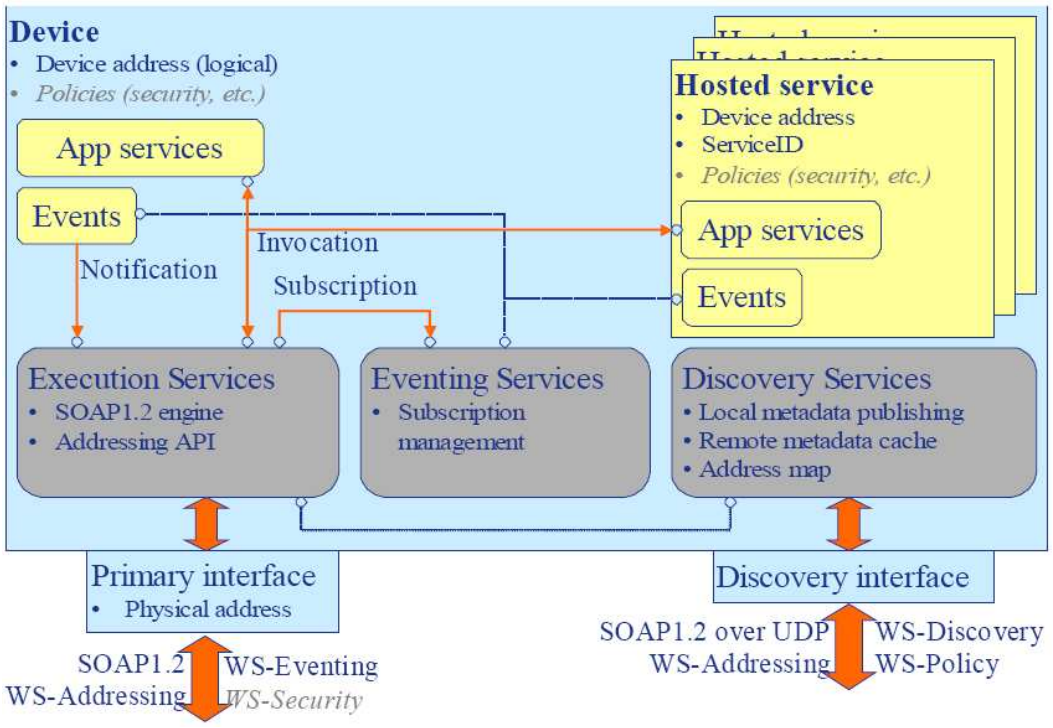


**Figure 3.3 – architecture of a device compliant with DPWS**

In figure 3.3 can be seen:

- User-defined services and events are shown in yellow. They are provided as user-written code and generated code in the DPWS toolkit;
- Predefined services are shown in grey. They are provided as run-time libraries in the DPWS toolkit;

- The two network interfaces are shown: the primary interface uses the standard SOAP 1.2 over HTTP binding to exchange regular SOAP messages, while the discovery interface uses UDP and a multicast address to broadcast and listen to the predefined discovery messages. Both interfaces rely on a standard IP stack.

## *3.6 – DPWS advantages and disadvantages*

DPWS strong points are the reliance on web services which developers highly accept, as well as it is platform and language independent. The number of functionalities that DPWS can offer at the time this thesis was written is vastly superior to any other similar technology. Devices implemented using DPWS can provide services to any application on the network. Thus, developers only must focus on the application itself.

The DPWS technology is a web-service implementation, so it is totally compatible with the web-services architecture. Inheriting the same concepts, it is very easy and fast to add, replace or change components. Web-services provide high-level mechanisms that abstract away the low-level effort to build a distributed architecture. The capability of making a component itself independent, as its dependency from any other is non-existent, makes the DPWS a very propitious technology for implementing in manufacturing industrial devices.

Also, DPWS have many practical advantages for developers, users and sellers:

- Lower production costs;
- Common solution to the industry;
- Web-service extension to devices;
- Strong security mechanisms embedded;
- Known internet patterns;
- Open-platform solution;
- Common development tools;

- Easier configuration
- Easier installation;
- Easier connection;
- Less product return;
- Richer user experiences;
- Better product differentiation;
- New automation opportunities;
- Lower support costs;
- More product confidence;
- Easy product upgrade.

On the other side, embedded systems normally are short on memory, not having processing power to run some technologies used in DPWS as a Hypertext Transfer Protocol (HTTP) web server, Simple Object Access Protocol (SOAP) engine, and a XML parser. All this requires more RAM usage, and furthermore, increases bandwidth and operating costs.

To allow SOAP implementation, the DPWS specifies limited constraints functionalities, allowing it to be implemented on small devices to restrict traffic. The protocol itself allows a large variety of options, but it also brings some complexity in the design of a concrete framework.

The DPWS services' interface description is made by the Web Services Description Language (WSDL). The current version of this technology (WSDL 1.1) is lacking information about more advanced interaction patterns. Therefore, additional methods are required for these purposes. However, the newer version WSDL 2.0 could offer better support and more realistic association with the service concept.

## *3.7 – DPWS – old and current implementations and application areas*

DPWS is an emerging technology used in many implementations and research projects. There are open toolkits enabling the development of service-oriented software components in DPWS as are SoA for Devices (SOA4D) and Web Services for Devices (WS4D). In the manufacturing industry there are many applications using DPWS, such as methods for developing efficient diagnosis mechanisms in devices [18].

In the content of this thesis, the obvious example is industrial automation. In this field, communication between devices is automatic, and the responsible people just either must pay attention to any warning given by some supervisor device, or give new orders to a specific device. This is a complete intelligent supervision and communication system, as it uses informative communication, intelligent control, supervising, etc

There are many examples of DPWS applications, being one of the most noticed the Windows Rally, a set of technologies from Microsoft that integrates DPWS in a stack side-by-side with other new web-based technologies. This turns many personal computers in the world compatible with this technology.

In industry, there are some applications using DPWS, as methods for developing diagnosis mechanisms in devices.

Another big application area is web-services it selves. For example, buying a flight ticket, the technology can be used as a client-to-machine connection, with the client at home buying to the air company server, or as a machine-to-machine connection, as the company server asks a printer to print the new (material) ticket, for example, or, more complex, alerts the police office for a just used stolen credit card.

Companies are also using DPWS to create products that allow a new superior degree in smart homes, controlling the interaction between the human, the house and the compatible devices on it. Infinity of possibilities is present here, as it can be used for controlling the house devices, it can be used for security, or even to control the devices remotely, by cell phone for example. Asking the microwave to start warming the food five minutes before arrive home is a very attractive idea.

In this field, a new product is revealing itself very fast. It is called Life|ware™. This next-generation home technology was highlighted by an automation system that used DPWS to communicate with a lighting system, audio distribution system, security system, motorized shades, security cameras, thermostats, washers and dryers, and a motorized television mount. As it says on its website (www.life-ware.com):

*"Life|ware™ is a simple but sophisticated software program that works with Windows® Media Center to give you one-touch control of your home's climate, lighting, security, audio and entertainment systems. So you can worry less and do more."*

For manufacturers, the first step towards DPWS adoption is the creation of a small device bridge between their native proprietary code and Web Services. At least 117 automation products from 37 different vendors currently support DPWS this way. At the International Security Controls (ISC) trade show, a major security company demonstrated a security system that supported DPWS, while the Kitchen and Bath Show (K/BIS) saw two major appliance manufacturers demonstrating washers and dryers that communicated using DPWS.

# Chapter 4

# SDPWS – to support Autonomous Components

Contents

## *4.1 – SDPWS: the autonomous component community*

An approach to an autonomous device was realized, using a DPWS simplification and enhancement. The temporary name of SDPWS (Simple DPWS) was given at its time, and from now on it will be used in this document. It is represented in figure 4.1.

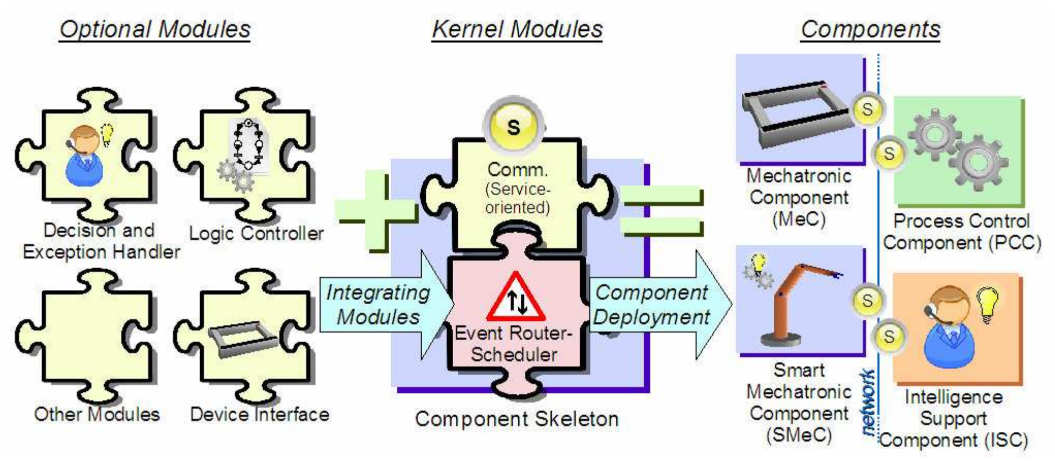


**Figure 4.1 – Simple DPWS architecture**

The main objective of this approach is to implement, with DPWS, a number of features in a device in a way that the device can work on its own, serving the others and using their services, be capable of interacting in as many ways as needed (I/O, Ethernet, User Interface, etc), and be capable of actualize its services in real-time, without shut down.

To reach this goal, some features are needed and were engineered, as seen below:

- The device must be able to run a DPWS client and a DPWS server, in order to offer services and use services;
- The device must be divided in different modules, each one independent from the others, and with its own functionality (for example, a I/O module that cares of information passed to and by an I/O physical port, or a communication module, that uses the DPWS to communicate with the other devices);
- It is needed a special module that will handle the internal communication of the device, i.e., the communication between modules;
- It must be possible to update the services of the device in real-time, in order to add or remove services without recompiling code or shutdown any device, this means, the update itself must work like a service that will be sent by the programmers.

The architecture designed is made of several service-oriented components, with different roles. The interaction is implemented by providing and requesting services. Each component is made of a framework with several functional modules.

Some modules are essential, like the communication module, which controls all the communication between devices and, therefore, is responsible for using DPWS, and the module responsible for the other modules' communication, the Event Router-Scheduler (ERS), which will be explained exhaustively in the coming chapters. The other modules can be user-defined and adapted to the components function. For example, if the component is controlling an automation device, it will require the interface module to access the I/O's, and probably will need some form of control module.

## *4.2 – SDPWS: the communication module*

Modular programming concepts are applied in the communication module providing a higher level of abstraction and making de DPWS program interaction easier to understand. It has many functionalities of the DPWS toolkit and has a list of WSDL structures. Services can be added and removed from the list, and new services are analyzed to determine the presence of existing parameter and consequent reuse, minimizing the required memory.

A dynamic stub and skeleton were implemented. This functionality is called Dynamic DPWS and will be exhaustively explained in chapter 6. This feature allows to dynamically add and remove services to the component, in real-time, and is made with C structures.

This ability promotes the reuse of the DPWS component in other programs and increases greatly the agility and life cycle of the components. Time saving and speeding up development are the main advantages.

The communication module offers a very useful set of technical functionalities. On the server side, it has the ability to support multiple devices (hosting devices), each one with one or more hosted services. The module uses the WS-Addressing protocol for coordination of service operations and device identification. On the client side, it offers an easy way to implement and perform lookups, retrieve metadata from hosted services and devices, and subscribe/unsubscribe services to the user.

As will be explained latter, the Event Router-Scheduler has a fundamental role offering a way of interacting between modules synchronous and asynchronously, making it not only extremely efficient, but also very easier to understand how to solve the problem of shared data between modules.

## *4.3 – SDPWS: enhanced interaction patterns*

The new methodology resolves some of the previously mentioned DPWS limitations in terms of advanced interaction patterns. The devices can be both clients and servers, providing and requesting services, and are implemented as part of the distributed control approach at the shop-floor level in automation and production systems.

All this process is made by the Communication module, generating the necessary events, through the Event Router-Scheduler (ERS) module, reaching the target functional modules. In the communication module, it is possible to create many ports, for the same service, and bound it with a port type. A new received message is automatically de-serialized into the C structures implemented by the ERS, and the entry port allows the developer to know which port type was intended.

Traditionally, in service-oriented systems, a service is a set of ports, being each one an instance of a port type. A port type defines a set of interaction operations and the corresponding message transfers between the service provider and the service requesters. In the proposed methodology, a service involves several phases of interaction with its requesters, which must follow specific protocols associated to the instances of the port types (ports).

The access to a service is done performing the following phases:

- Discovery phase – which is the ability to discover services;

- Negotiation phase – for quality of service (QoS) discussion or different priorities, for example;
- Operational phase – which is the service itself, with both mains ports and logical ports being used here. Before the use of a service, the client must search a one that completely fulfils its needs. After obtaining the interface, it can contact the provider, and then, if agreed, it uses the service as negotiated;
- Termination phase – is the disconnect phase, when the service is not needed anymore. Consists on a *finish* message followed by a *finish* reply.

The figure 4.2 shows a sequence diagram of the process to interact with services using the four phases. Besides that, the proposed methodology has some additional features. First, ports can be created dynamically, and a non-existent operational port can be created for a specific situation. Second, it is possible to provide similar ports for the same functionality, but instantiated from different port types.

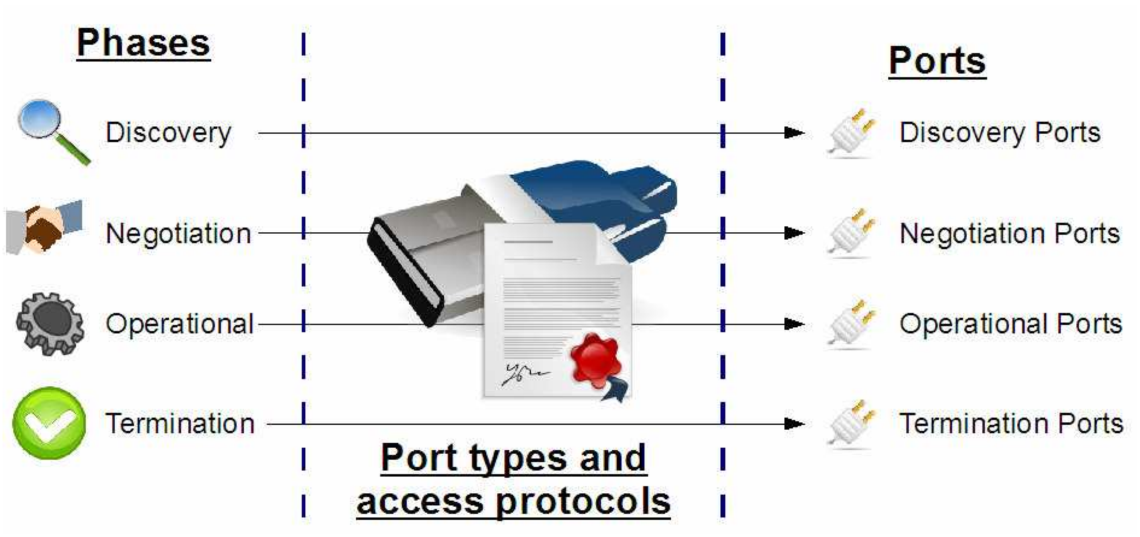


**Figure 4.2 – Service Accessing Interaction Sequence**

The interaction itself is separated into the four phases also. To each phase corresponds one or more port types with the associated access protocol that defines the interaction rules. The access of the requester is done to the instances of the port types. In generally, discovery phase can be associated to the facilities provided of DPWS and thus does not need special treatment for simple discovery processes.

In service-oriented systems the interactions are made of requesting existing services by a client that want to use them and obviously coordinating the process. One approach is to use the WS-Discovery protocol that defines a dynamic multicast discovery mechanism without using any intermediate entity. Before the use of the service, the requester must search for a specific service that fulfills its needs. After obtaining the interface (which describes the service), it can contact the provider.

Obviously it must first make a proposal to use the service. If not accepted, it may proceed to a more complex negotiation with the provider. After the operational phase, the termination phase may setup processes to conclude the usage of a service.

The logical ports of the operational phases can be directly related to the physical ports of the devices.

The interaction method may be complex. Semantic-rich descriptions allow using machine reasoning to perform automatic matchmaking of services using logical inference. This allows the use of services that did not exist or were not known when the client was programmed, as the services are selected dynamically.

# Chapter 5

# The Event Router-Scheduler

Contents

## *5.1 – SDPWS: The Event Router-Scheduler module*

Automation and production systems are evolving in the direction of autonomous and collaborative components, approaching the idea of an ecosystem. Each habitant of this system is responsible for different and concurrent activities and thus requires an adapted anatomy that is balanced for the several requirements.

The Event Router-Scheduler introduces an anatomical-like way for implementation of functional and reusable modules which constitute service-oriented automation components.

Paying attention to the internal software structure of the automation device, the ERS is the mechanism that binds the several modules together. The resulting software automation components are customized for different tasks due to the inclusion and management of the specialized functional modules, and provide the ability to operate in a service-oriented automation and production environment.

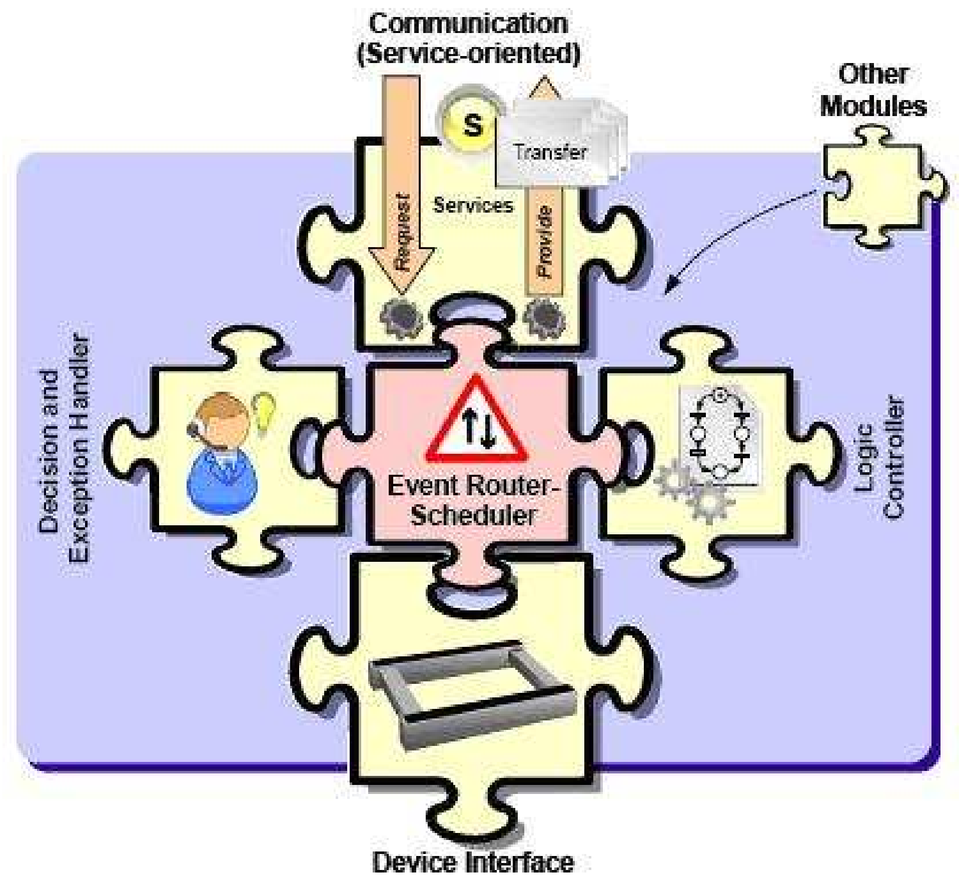


**Figure 5.1 – The Event Router-Scheduler in the application structure**

The ERS can be compared to the nervous system of living beings in sense of carrying impulses from and to different organs, and so, maintaining the dynamic information flow (figure 5.1). Intelligent behavior can be reached when these “nerves” are linked to the “brain”, that provides static control based on workflow processes and also autonomy to respond to unexpected events, undocumented situations and internal objectives. Being inserted in a service-oriented environment, interaction with other components is achieved only by providing and requesting services o reach local and global objectives.

## *5.2 – Event Router-Scheduler: Internal Anatomy of components*

Each Service-oriented Component may be implemented independently and differently. The only requirement is that it should share its functions as services and obey to the protocols of communication and processes. To be able to construct and deploy these components in a simple but functional way, an anatomical-like framework was specified.

A general component is structured in an anatomical form comprising several "organs" (functional modules) that are responsible for individual tasks, as illustrated in Fig. 5.2: Logic Controller, Decision and Exception Handler, Communication, Device Interface and Event Router-Scheduler. These modules are included in the control component according to its needs and possibly implemented using different technologies. It is also possible to develop and integrate other modules for diverse functionalities, if they respect the rules provided by the framework for the integration (task of the Event Router-Scheduler).

The Event-Router-Scheduler and Communication modules are the kernel modules to develop a Service-oriented Component based on the proposed anatomy. They are responsible, respectively, for the main framework of the component (event-based inter-module communication and integration) and external communication with other components (service-oriented inter-component communication). Other modules may be added to the structure according to the component's requirements.

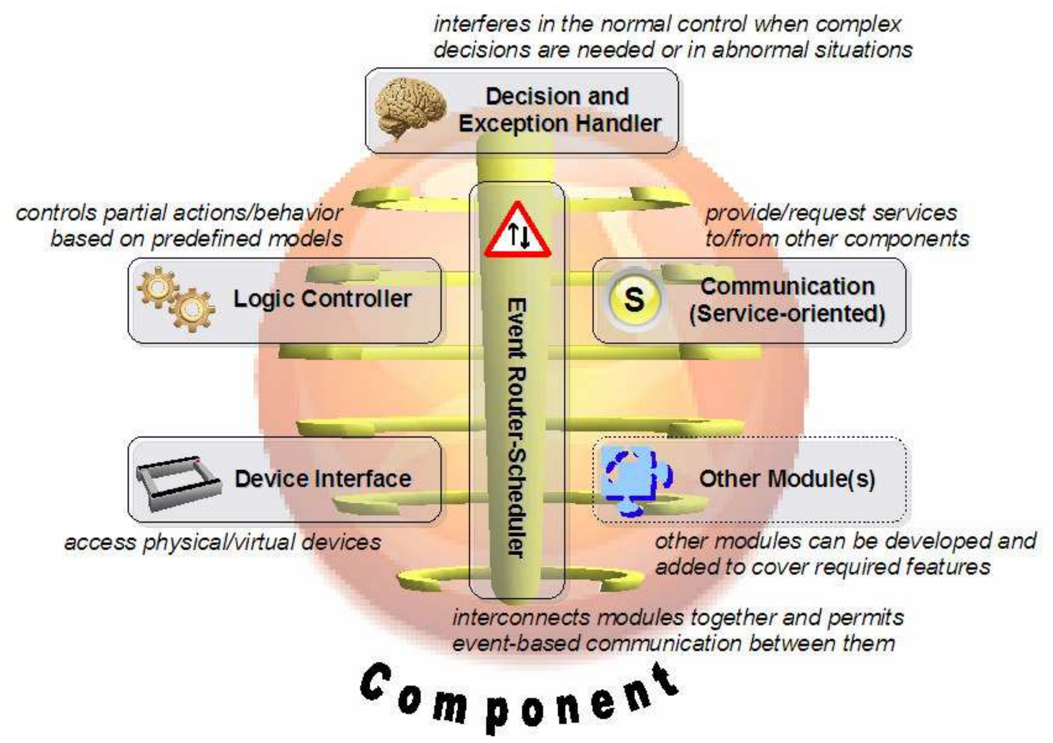

interferes in the normal control when complex decisions are needed or in abnormal situations
Decision and Exception Handler
controls partial actions/behavior based on predefined models
provide/request services to/from other components
Logic Controller
S
Communication (Service-oriented)
Event Router-Scheduler
Device Interface
Other Module(s)
access physical/virtual devices
other modules can be developed and added to cover required features
interconnects modules together and permits event-based communication between them
Component

**Figure 5.2 – Anatomical-like structures component**

In more detail, the Communication module provides the necessary functions to expose the services from the associated component and request services from other components. Other functions include, among others, discovery and negotiation mechanisms. The remaining modules of the component may use the Communication module to access these functions through impulses (events) provided by the Event Router-Scheduler module.

As an example, a conveyor may provide the *Transfer* service to handle the movement of pallets, which is controlled by the Logic Controller module and accessed by the Device Interface module. The *Transfer* service may be used by the other components, but the component itself can also call external services when needed (e.g. to be connected to other conveyor it requests the *Transfer* service of that conveyor) [8]. A suitable technological solution to implement the service-oriented communication module is to use Web technology, and most specifically Web services. At its core, Web services technology is quite simple and it is designed to move XML (eXtended Markup Language) documents between service processes using standard Internet protocols. This simplicity helps Web services to achieve the primary goal of interoperability and also means that it is necessary to add other technologies to build complex distributed applications.

The remaining modules are described briefly in Figure 5.2. The goal to include the other modules is to provide an example of a Service-oriented Automation Component that is mediator of some physical equipment with control capabilities. For example, the resulting component of Figure 5.2 represents a smart controller of a conveyor device, by providing several features such as control and access over the physical device, ability to decide in unexpected and undocumented situations and also the possibility of service-oriented communication to other components. Other example is a service-oriented PLC-like controller, which may interpret control models and give the necessary orders to other components via the invocation of the provided services by them. In this case, it is not necessary to have the Device Interface module, since it does not command directly the devices.

Finally, the “nervous system” of the anatomy represented in Fig. 5.2 is managed by the Event Router-Scheduler.

## *5.3 – Event Router-Scheduler: the module*

Components and devices that implement several of the expressed aspects of service-orientation require a consistent anatomy to deal with the different function modules (“organs”) in order to fulfill the necessary requirement. Other problems may arise from the asynchronously operating modules, possible data inconsistencies and concurrent processes/threads. For this purpose, it is proposed a mechanism to provide an “impulse” (event) passing and scheduling feature to guide the impulses to different modules, thus permitting the synchronized communication between them. The heart of the component is the Event Router-Scheduler (ERS) module.

During the design phase it was clear that the ERS should meet the following objectives:

- Common event routing/scheduling mechanism for the communication and integration of modules;
- Provide some transparent functions for creating and managing modules;
- Suitable for software application that are deployed both in traditional PC and embedded systems;
- High performance, especially in critical situations and targeting real-time applications;
- Use of C language, aiming to balance between performance, portability and features;
- Thread safety and management of data concurrency;
- Easy to use by developers, in sense of building modules and how events are processed.

The function of the ERS is comparable in some parameters to the nervous system of living beings, including humans. H. Gray wrote in his book “Gray's Anatomy of the Human Body” [9]:

“*The Nervous System is the most complicated and highly organized of the various systems which make up the human body. It is the mechanism concerned with the correlation and integration of various bodily processes and the reactions and adjustments of the organism to its environment.”*

In the case of Service-oriented Components, the “environment” is captured and manipulated by specific modules (e.g. Communication and Device Interface), but the natural equilibrium with impulses (events) of the several modules and their integration is reached with the help of the ERS.

Figure 5.3 shows the generic conceptual structure of the Event Router-Scheduler. The feature groups are separated in blocks that correspond to the Scheduling and Routing of Events, Hardware/Software Abstraction, Threading and Data Consistency and Template/Interface for Event-based Modules.

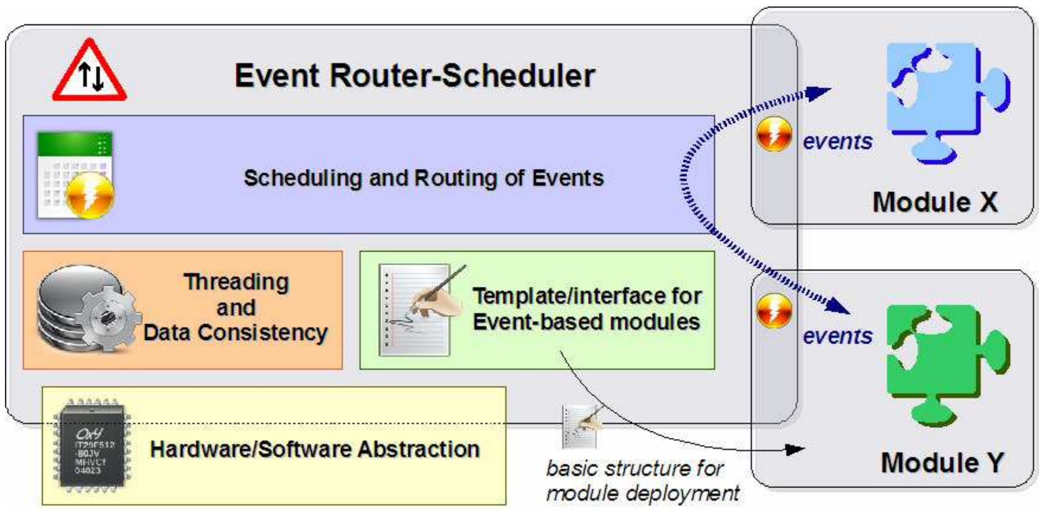

Event Router-Scheduler
Scheduling and Routing of Events
Threading and Data Consistency
Template/interface for Event-based modules
Hardware/Software Abstraction
events
Module X
events
Module Y
basic structure for module deployment

**Figure 5.3 – the Event Router-Scheduler generic concept**

The main feature is to provide event-based communication between functional modules and the corresponding routing and scheduling of events (see Scheduling and Routing of Events block of Fig. 5.3). From the practical point of view, the component’s internal impulses (events) between its functional modules are integrally managed by the ERS. The ERS allows synchronous and asynchronous event calling between any modules (which is critical in real-time applications), and offers several additional procedures to realize more complex operations, like events generated by other events and time-triggered events. In the most basic form, a sender module must only emit an event to a specific destination (other module) and the ERS routes it to the destination. There are also other options for sending and processing events, such as events with reply and multicast events to several destination modules (see Fig. 5.4).

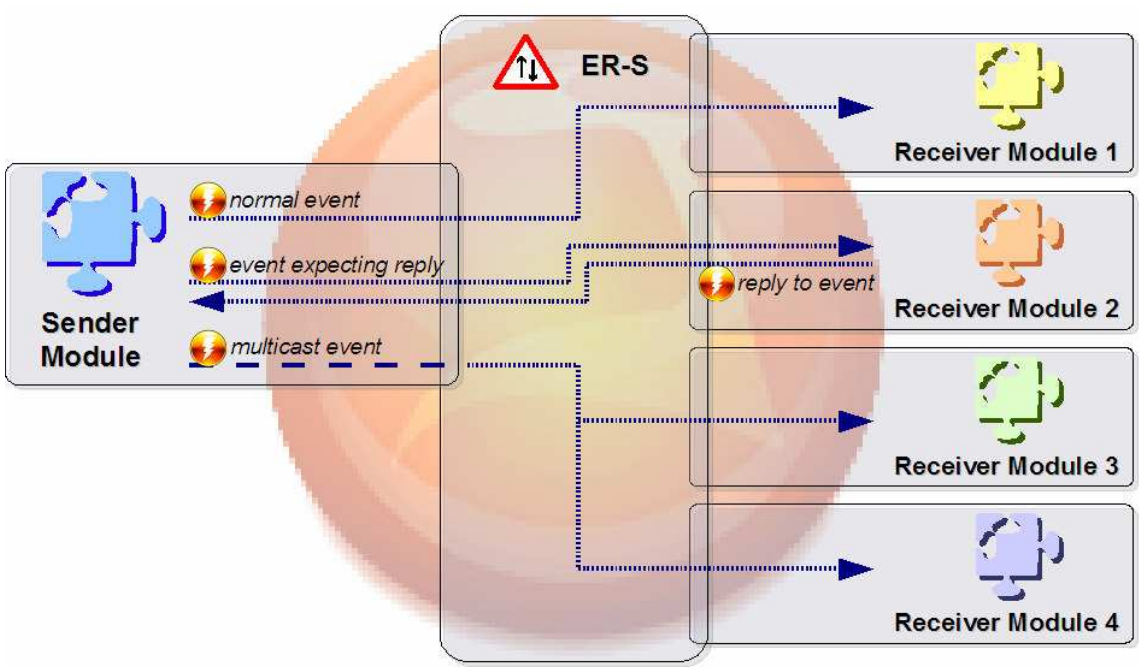


**Figure 5.4 – ERS' many possible operations**

An event is a structure with all the information a module needs to know regarding various possible situations. Besides the standard information as the intended action and the parameters from who the event came, it can ask for a reply, for an information forwarding, can have a fault message's receiver, and the event's receiver can check if it is a reply. Also, an event sent more than once, by error, is detectable.

The ERS uses lists as a way of transmitting and queuing events between the modules, so the number of events waiting to be processed is only limited by the available memory. The ERS uses some techniques to avoid memory fragmentation, because the creation and elimination of new data is a very frequent operation in the modules, as the world is constantly changing. In some cases when the number of events is high, the ERS offers the possibility to give different priorities to the events. Like this, an event sent to a certain module will always pass by all the waiting events of that module which have lower priority than the sent one.

Being capable of both synchronous and asynchronous operations, the asynchronous ones are managed using threads. The synchronous operations can be either freezing or non-freezing for the receiver. For example, on event reading the operation can be a module-freezer or not. In case of the freezing mode, the module freezes until any event arrives for it. After that, the module continues its normal proceeding, as shown in Fig. 5.5(a). This is a very low CPU resource-taking procedure, useful for embedded devices. However, it is not useful for real-time multi-task modules, as this kind of module should not freeze. On the other hand, the non-freezing event reading always receives an event. However, it can be an invalid event. An invalid event means that there were no events for the module, so it can continue processing its other tasks. Obviously, if it is a valid event, the module should process it. This is represented in Fig. 5.5(b).

Asynchronous event triggering is also possible. Callbacks are used to perform this type of operation, as it must occur when it is called. However, the event is not triggered immediately, because of data-protecting, and it should only occur when the module activates an authorization (*mutex*) to allow callbacks, which will possibly change the module's data. Each module has its *mutex* for this matter, and developers who want to enable asynchronous event handling should be very careful with this protection.

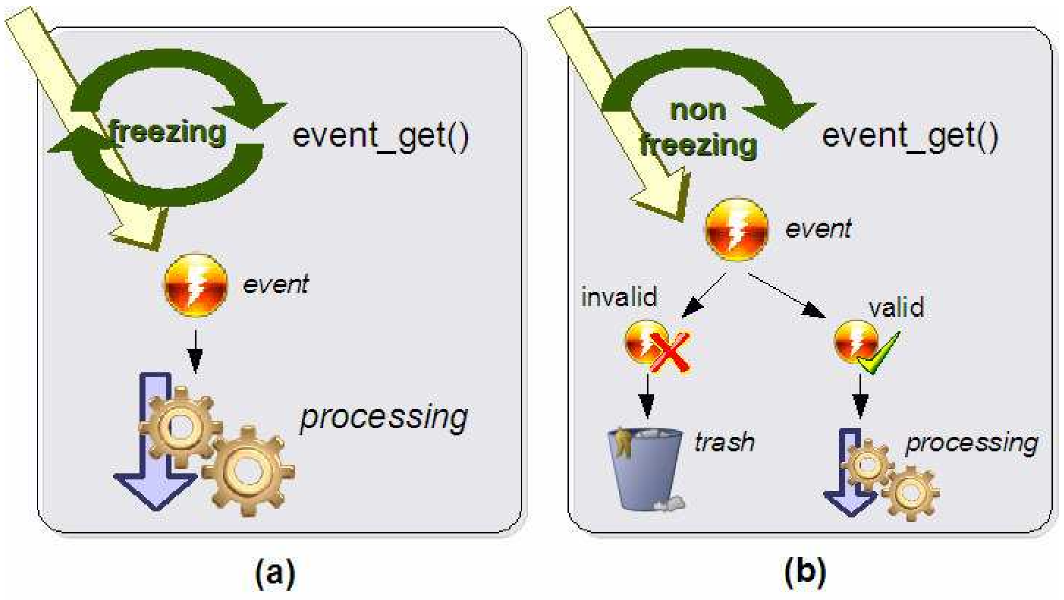


**Figure 5.5 – (a) ERS freezing event get and (b) ERS non-freezing event get operations**

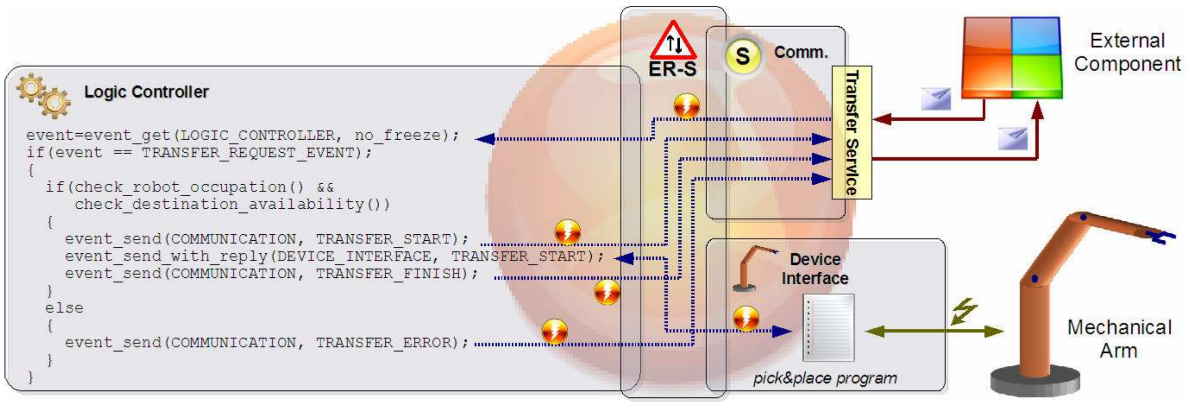
The remaining blocks of Fig. 5.3 are responsible for adjacent tasks of the Scheduling and Routing of Events block, specifically to its and other modules’ management. The Hardware/Software Abstraction provides some functions transparent to the system architecture that can be accessed by all modules. Since the ERS and other modules are in a multi-functional and concurrent environment, a special block of the ERS, namely the Threading and Data Consistency block, introduces simple thread manipulation and data protection (such as *mutex*).

Finally, the Template/Interface for Event-based Modules block provides the basis for creating functional modules and associates them to the ERS. Each module can be programmed independently. This means that it is possible to remove, replace, upgrade or add new modules. This makes a program using the ERS very flexible. The module ID is the module’s identification and it is unique for each module. This variable is what the other modules need to know to send an event to a specific module. It is comparable to the code that the nerves carry to reach some organ. However, it is also possible to search a module by its type like “controller” or “user interface”, as this way is much more practical for a developer to reach a module without many information.

## *5.4 – Event Router-Scheduler – implementation and operation*

A prototype implementation has been done to test the proposed framework, integrally coded using the C programming language and compatible with Windows and GNU/Linux operating systems (targeting also others, such as VxWorks). Some implementation details are given next.

The functions provided by the framework to develop and operate components are explained with an example component representing a mechanical arm (articulated robot to move small objects) made of three modules (besides the ERS), represented in Fig. 5.6. The modules correspond to a subset of the ones in Fig. 5.2 (excluding the Decision and Exception Handler module) that are briefly commented in chapter 5.2. The major difference is that it is connected to the mechanical arm via the Device Interface module, instead of the conveyor of Fig. 5.2.



```
event=event_get(LOGIC_CONTROLLER, no_freeze);
if(event == TRANSFER_REQUEST_EVENT);
{
  if(check_robot_occupation() &&
     check_destination_availability())
  {
    event_send(COMMUNICATION, TRANSFER_START);
    event_send_with_reply(DEVICE_INTERFACE, TRANSFER_START);
    event_send(COMMUNICATION, TRANSFER_FINISH);
  }
  else
  {
    event_send(COMMUNICATION, TRANSFER_ERROR);
  }
}
```

**Figure 5.6 – Mechanical arm controller example using ERS**

In terms of data structures, the ERS includes several structure types for storing and relating different information about modules, events and other aspects. The Module Structure, which represents a module in the program, identifies its module by a unique ID. It also provides storage for local information such as the module’s incoming events list, which is where the module is going to get the events sent by the other modules and a pointer to the module’s callback implemented function, which is triggered by new events when the asynchronous mode is activated. The Event Structure has all the information to handle an event: action name, parameters, who is sending/sent it (module ID), and some variables for reply handling, an ID of the event and ID of the reply. Finally, the Database Structure of the ERS is where pointers to all modules are allocated.

First the modules must be created. Thus, the respective function shall be called, and each module must have an ID and type, as seen below:

```
module_create(1,DEVICE_INTERFACE);
module_create(2,LOGIC_CONTROLLER);
module_create(3,COMMUNICATION);
```

Sending and receiving events is very straightforward. For example, to send an event from module 1 to module 2, the developer must create an event, put the sender, the action and the parameters, and then send it with low or high priority, to the destiny, using the *`event_send()`* function. To read an event, presuming that callbacks are disabled, module 2 must call the synchronous event handle by either freezing while there are not events, or not freezing. This variable is a parameter when calling the event-reading function, as it can be something like *`event_get(2, FREEZE)`* or *`event_get(2, NO_FREEZE)`*.

The not-freezing way of getting an event always returns an event, but it may be an invalid event. On this case, valid events always have valid senders, i.e., the *`from`* variable, which corresponds to the sender ID, is always bigger than zero. So, invalid events have negative sender IDs. If the module 2's callback is `ON` instead, and if the callback mutex allows it, the new event would immediately trigger the callback, so it would run the function pointed on the module 2's structure.

More flexible operations can be done with multicasting and reply to events. In case of multicasting, there is a special function to emit an event to several destination modules: *`event_send_multicast()`*. One of the parameters is a list of destination modules that are intended to receive the event. Some events may expect replies and this can be done in two ways: asynchronously (nonfreezing) using the *`event_send()`* function with the attribute *`reply_id`* and synchronously (freezing) using the special *`event_send_with_reply()`* function.

For the example, the modules of the mechanical arm component have simple functionalities. The Device Interface provides the access to the mechanical arm in sense of calling the programs of pick & place to move the objects from one place to another. Its Communication module uses a service-oriented infrastructure, based on a DPWS implementation, namely SoA for Devices (SOA4D). Through the communication module, the component provides one service, *Transfer*, to be called externally in case objects are available to be transported. Finally, the Process Controller module is responsible for coordinating the components activity, generally synchronizing service calls with the pick & place program execution of mechanical arm.

A simple algorithm is presented in Fig. 5.6 inside the Logic Controller module. Each time a function is required by one module to another one, events are sent through the ERS. In case of the algorithm of Fig. 5.6, an instance of it is executed when the Transfer service is requested and then the Communication module of the component emits an event to the Logic Controller. It is assumed that the *Transfer* service is called when an object is ready to be moved. From the other hand, the operation of pick & place program can only be started if the mechanical arm is not occupied and if the destination where to place the object is free. For the sake of simplification, these checking functions are represented in the algorithm but their behavior is absent in Fig. 5.6., which would involve sending/receiving events to/from the Device Interface and possible also an entity representing the destination place. On successful conclusion of the pick & place program of then Device Interface, an event is sent back to the Logic Controller, and by its turn to the Communication module that then notifies the external component and thus concludes the service usage.

# Chapter 6

# The Dynamic Service

Contents

## *6.1 – DPWS service's static creation*

DPWS allows implementing web-services in electronic devices. These services are generated using the Web-Service Description Language (WSDL) file.

The WSDL is an XML-based language that provides a model for describing web-services. It defines services as collections of network endpoints or ports. The abstract definition of ports and messages are separated from their concrete use or instance, allowing the reuse of these definitions. A port is defined by associating a network address with a reusable binding, and one or more ports define a service. Messages are abstract descriptions of the data being exchanged, and port types are abstract groups of supported operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding, where the operations and messages are then bound to a concrete network protocol and message format. In this way, WSDL describes the public interface to the web service.

The WSDL object hierarchy can be seen in figure 6.1. Here is the explanation of each object:

- Element: consists of a unique name, and data type. The purpose of a WSDL element is to describe the data and define the tag which delimits the data sent in the message parameters.
- Message: corresponds to an operation. The message contains the information needed to perform the operation. The message name attribute provides a unique name among all messages. The part name attribute provides a unique name among all the parts of the enclosing message.
- Operation: can be compared to a method or function call in a traditional programming language. Here the soap actions are defined and the way the message is encoded for example, "literal."
- PortType: defines a web service, the operations that can be performed, and the messages that are used to perform the operation.
- Binding: Specifies the port type. The binding section also defines the operations.
- Port: The port does nothing more than define the address or connection point to a web service. This typically is a represented by a simple http url string.
- Service: can be thought of as a container for a set of system functions that have been exposed to the web based protocols.

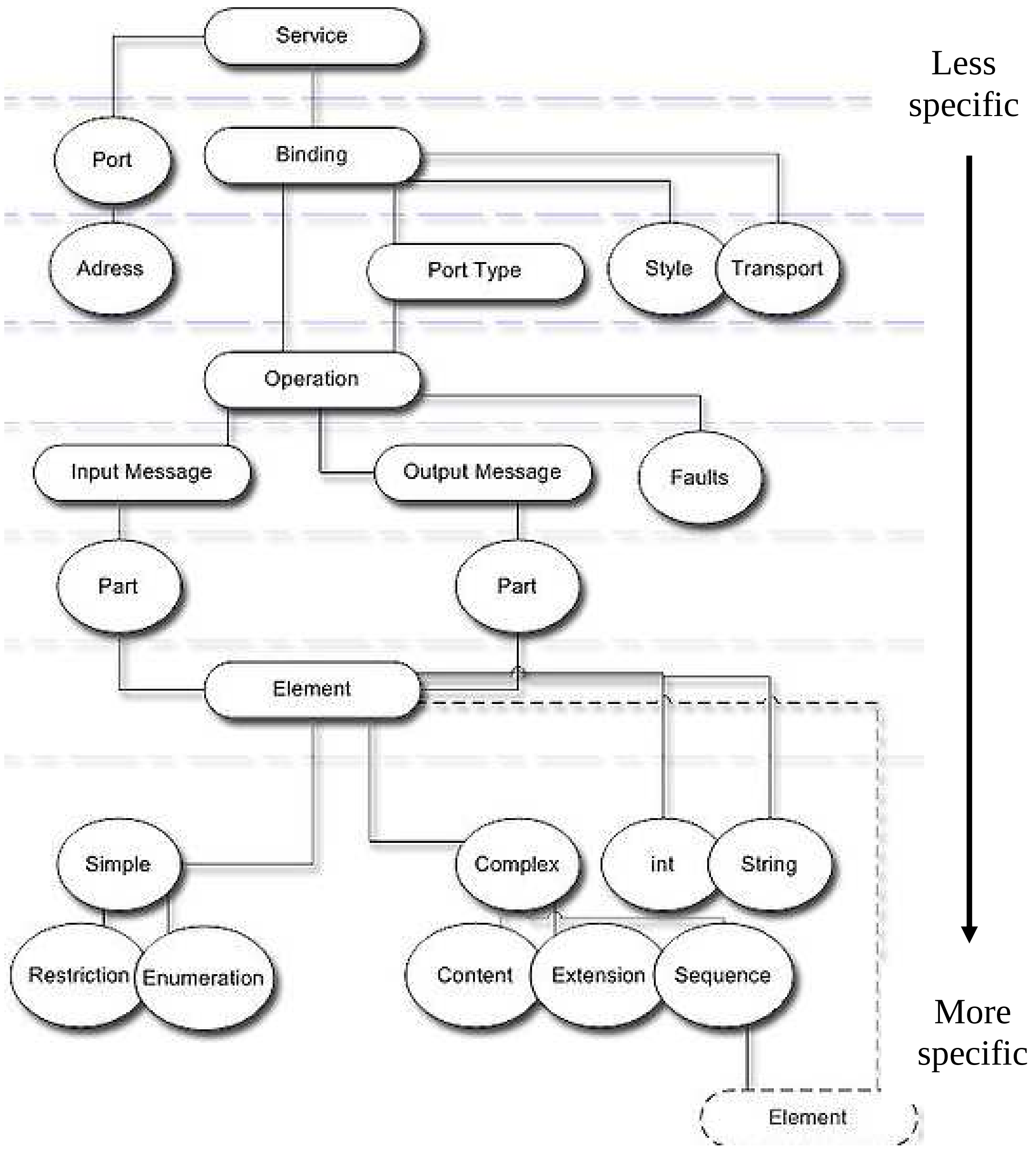


**Figure 6.1 – WSDL 1.1 object hierarchy**

The approach used by the DPWS toolkit is the same used by the gSOAP toolkit. It relies heavily on automatic code generation to map back and forth SOAP envelopes and C structures. Relatively to the service definition, which involves operations, data types used as parameters, and return values to those operations, the DPWS toolkit generates the required code to provide transparent access to the remote operations from a client. The only pieces left to implement by the developer are the implementation of the operations in the server and the implementation of the client that invokes the remote operations, with the respective arguments. The server then must process the new received value.

The proxy, skeleton, and marshalling and de-marshalling (transformation between C structures and SOAP-XML messages) code is completely generated by the toolkit. The marshalling/un-marshalling code is the same in the client and the server.

There are two types of messages. In one-way messages, only the request message is transmitted. On request/reply messages, SOAP responses are returned.

gSOAP provides a code generator that transforms a WSDL document into an annotated header file ready for processing by the gSOAP compiler. The figure 6.2 summarizes the various artifacts involved in Web service development, using gSOAP.

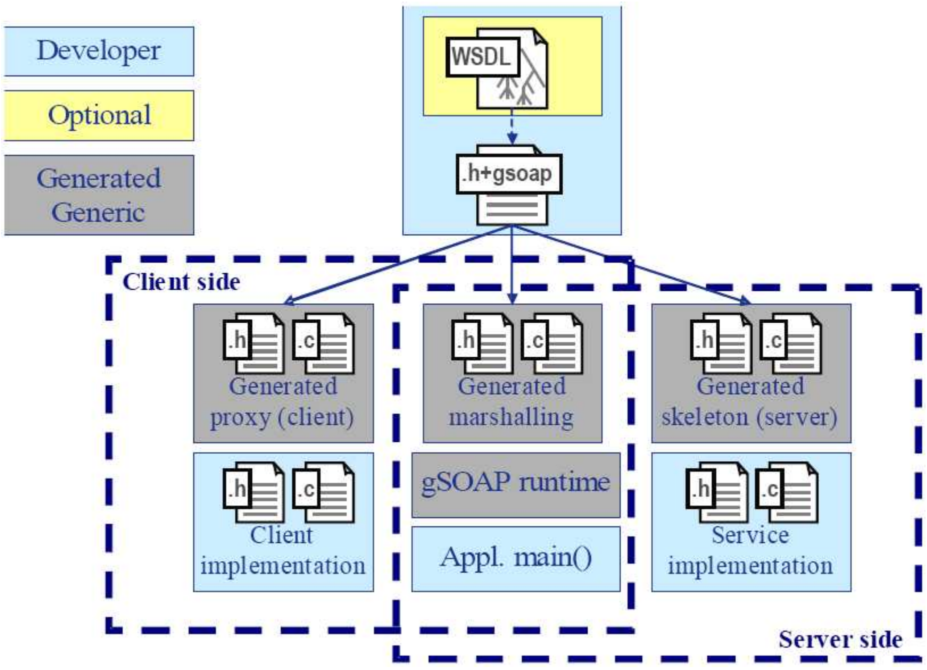


**Figure 6.2 – gSOAP's artifacts involved in web-service development**

The current DPWS toolkit extends the above code generation principles to take into account the WS-Addressing specification. This specification states that a SOAP response and fault message may be redirected at user choice to any endpoint, and not necessarily to the origin of the request. This means that the standard, synchronous approach, which uses the HTTP response to carry the response or fault message associated to a SOAP request sent through an HTTP request, is not always applicable. So, the DPWS toolkit supports asynchronous transfer of response or fault messages to a specified endpoint. Such an endpoint should be ready to receive asynchronously SOAP responses and fault messages. The DPWS toolkit generates header and C files implementing the skeletons for the handlers, following the pattern used by the gSOAP service skeleton generation. A similar approach is used for handling events: endpoints that subscribe to events should be set up as message servers, and use the event handling skeletons generated by the DPWS toolkit to process received events.

## *6.2 – Dynamic service: specifications*

As explained, the DPWS toolkit generates compact code, creating a light weight resource-constraint program structure with a high performance predictive parser for each service, allowing an efficient marshaling and un-marshaling of data. However it offers no capability of interacting with new services that appear at run-time, making it time costly and removing the ability to quickly react to changes since a new code has to be generated and compiled on the client stub to integrate the unprecedented service. For these reasons there was a necessity to do some enhancements over the DPWS toolkit to have a dynamic stub and skeleton that can invoke and receive any kind of message.

This new feature must accomplish some objectives, whose will be explained next.

First, the most obvious, it must be able to read a new service at run-time. This service will be written in a WSDL file and, somehow, passed to the device. So, the service-reading must be a service itself that will receive a WSDL file. So, this service must be built-in.

Second, it must transform the WSDL file into a service's code. This transformation will be made by a de-serialize algorithm implemented on the device. The final result in memory should be identical to the static DPWS. Notice that the WSDL must be validated before loaded. Even if an error-catch algorithm can be implemented, it should be avoided, as the resource-taking from the device, principally the memory, should be minimal, and this kind of operation takes memory and processing time. However, the device must be capable of discard erroneous services, and, very important, warn the user about this service discarding decision, to avoid users to think the service is online and running.

Third, the device must inform the other devices that a new service is available. DPWS informs about its services with the HELLO messages, when it starts running. In this case, the service is already running so it needs a procedure to send an HELLO-like message with, at least, the new service.

Fourth, the device must be able to create, send, read and process a SOAP envelope, using its dynamic structures, just like it does in the original static implementation. For this purpose, there must be implemented a SOAP envelope serialize/de-serialize algorithm. Again, exists the possibility of, somehow, the received envelope from the communication module is containing an error, so this de-serialize algorithm should be capable of send back a message of non-existent service or any other error that can appear.

Fifth, for maintenance purposes, it should be possible to import/export the actual state of the memory that represents the service. This means that the pairs `<element, value>` should be listed and then sent in to some debug application. At extreme debug conditions, it should be possible to send all the structure, as it would be not a difficult or resource-taking procedure, but it looks like unnecessary, as if there were any error in the structure, it would happen on the structure creation. This kind of error should be catch at programming time.

## *6.3 – Dynamic Service: implementation*

According to the previous specifications, a dynamic stub and skeleton were implemented. Not all of them, but in the future it will be. At the time this thesis was written, the implemented specifications are explained next.

Using C structures, a replica of all existent WSDL objects were created. This means that any and all objects that came in a WSDL file can be passed to the device memory, more concretely to C structures. This is illustrated in figure 6.3.

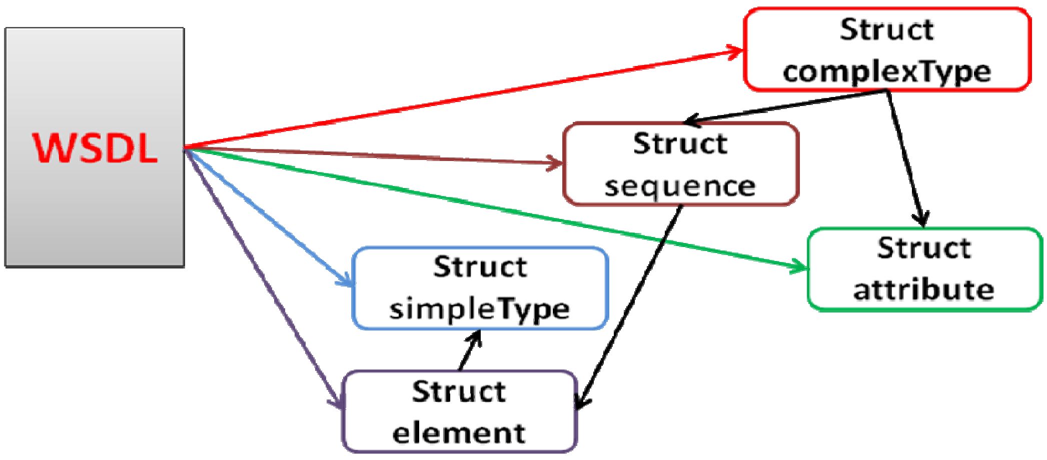


**Figure 6.3 – WSDL to C structures transformation**

This copying operation always respects the WSDL architecture, as the creating procedures it selves do not allow invalid operations. This respect of the service architecture can be used as a WSDL error-catcher, because it cannot implement objects out of order. Figure 6.4 illustrates this situation.

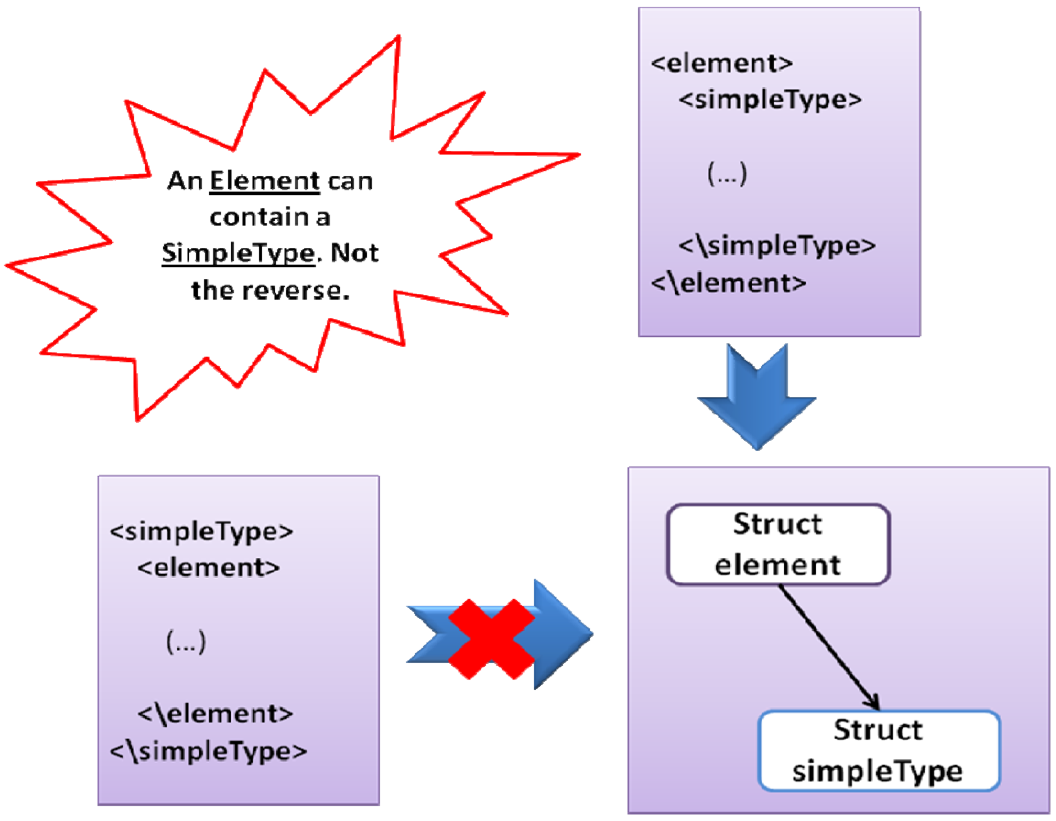


**Figure 6.4 – invalid WSDL cannot be implemented**

To use the dynamic stub, the client creates an URL-like path (URL – Uniform Resource Locator) that includes the service where it refers to, from *Service* down to *Element*. Also, it must send the arguments values of the Element he wants to change. This path corresponds to the Element position in the architecture which is seen in figure 6.1. This topology allows pointing to any element of the structure in memory, and this path allows any device with the same structure in memory to reach a specific object.

The programmer, however, doesn't need to know the entire path to the variable he wants to change. The path serialization and de-serialization is done automatically. If it is needed to change the value of an Element, the programmer only must call the proper procedure referring only that Element. That procedure is capable of getting the rest of the family of objects to the top object. So, the path generation is automatic, as the WSDL C structure present in the module fills the missing information necessary to construct the SOAP message. This path is needed because the server may have *n* Elements with the same name. So, it is necessary to specify the respective path (Operation, PortType, etc) for the server to reach the correct Element.

That means the dynamic server has a mechanism to always reach the correct Element, as the static DPWS can do. The programming over dynamic services is, consequently, as easy or difficult to do as the programming over the static services.

The dynamic skeleton does the opposite process. It receives the SOAP message and with the help of the WSDL service in memory, it automatically de-serialized it into the C structures, following the received path, with the information of the service, operation id, the arguments types and values. Also, it changes the Element value, if it is the case.

The static DPWS uses functions created at the WSDL-to-.h conversion. For example, in the following expression taken from a DPWS sample code,

```
dpws_send___lit__Switch(&dpws, invokationEPR,
lit__PowerState__ON)
```

, the `dpws_send___lit__Switch` procedure name, as well as the `lit__PowerState__ON` command variable, are created when passing from WSDL to the application stub. This means that a different name, if not pre-created, is automatically not recognized and erroneous for the compiler.

The dynamic DPWS uses a generic `dpws_send()` function present at the DPWS toolkit at the time this thesis was written. This procedure allows sending to an End Point Reference (EPR) any message the programmer wants to, and not just a pre-defined message.

Here, the command argument is not pre-defined, so the Element's tree path must be used here, for the reasons explained above. The `send` function, however, builds the SOAP envelop automatically, leaving the programmer to define only the message. But, as said before, the path creation is automatic, and the programmer only must specify the Element that is going to change, and the respective new value. This way, the SOAP message creation is automatic, as is the SOAP envelope, leaving nothing undone. Serializing and de-serializing the SOAP envelopes is, consequently, fully automatic, by the `dpws_send()` function itself.

## *6.4 – Dynamic DPWS: the Simple Service*

The created dynamic DPWS has all the structures needed to recreate the WSDL file services. However, some of the WSDL objects are constants or not used relatively to the shop-floor work at Schneider-Electric GmbH. For example, the binding object can be either RPC-literal or document-literal, but in this case is always document-literal. So, the WSDL binding' respective structure doesn't need to exist. This permits to avoid a structure that must contain pointers to a SOAP binding and the respective Port Type. This avoidance, joint with the others, will short significantly the memory usage.

It was created, then, a shorter service in memory, capable of everything a normal (dynamic or not) service can do, but, besides dynamic, it occupies less memory than the normal dynamic service. It is called the Simple Service.

The WSDL objects avoided on this approach are the HTTP interaction specification ones, that are the WSDL Bindings, the WSDL Operations and the Service Ports. In the dynamic DPWS Structures, however, the WSDL Bindings there are SOAP Bindings. So, this makes a total of five objects avoided in the family tree of an Element, which is what is being always accessed.

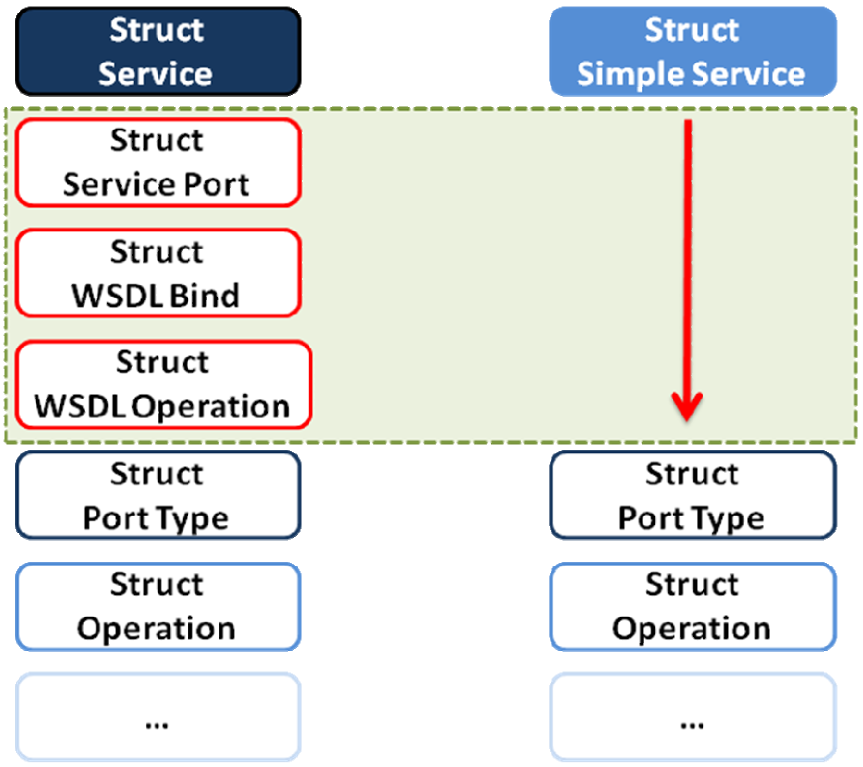


**Figure 6.5 – DPWS Service vs. (Dynamic) Simple Service**

However, the normal Service points to all of the above mentioned objects. The Service must exist, or else the path present is the SOAP message is incomplete, and the receiver doesn’t know which Element the message refers to. So, a simpler Service structure was created, the Simple Service structure. This service links directly to Port Types in the WSDL architecture. In figure 6.5 can be seen the difference between the two architectures that can be used in the dynamic service.

This simplification permits not only memory and time saving, but also an easier comprehension of the service itself by programmers, as the HTTP interaction part of the service is static, and so, it remains invisible to the programmers, who only must care about the services itselves.

## *6.5 – Dynamic Service: Benefits and Disadvantages*

The capability to dynamically add and remove services in real time promotes the reuse of the DPWS component in other programs and increases the agility and life cycle of the software. Advantages compared to the code generating techniques from gSOAP are the time saving and speeding up the development of projects. These factors represent too much money in the industry and should never be ignored.

The service loading can be done from anywhere. It is not needed to be near the device to do it. If the device can be reached by DPWS, everything can be done as a DPWS service (figure 6.6).

The capability of send services from devices to devices (figure 6.6) increases the fast upgrading capability of a shop floor, for example, as the new service configuration version must only be sent to one machine, and the identical machines automatically can ask for it. Besides upgrading, for first installation, the technique can be the same, as only one service installation is needed, because the service-sending and service-receiving must be built-in, the working device can send the service to all the other identical devices. This capability reminds the behavior of a virus, spreading itself, but in this case, with authorization. However, with this feature, security measures must be taken, because some bad-intentioned person can load a service into a device.

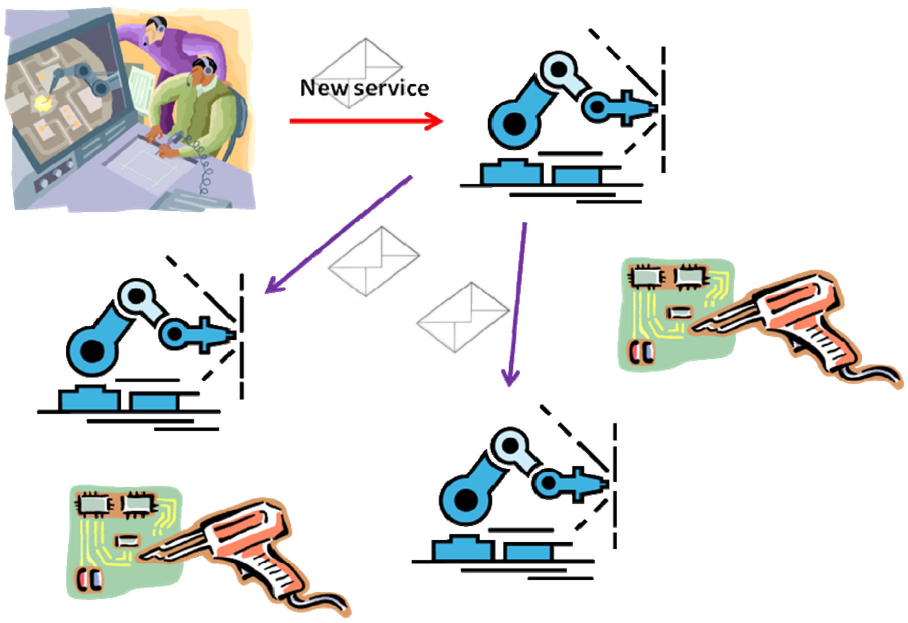


**Figure 6.6 – service self-spreading to the other identical devices**

Reconfiguration of devices when new add-on hardware is available can be done quickly and easily. For example, if a new kind of grip is available to a pick-and-place operation, it is not needed to reconfigure the whole machine, but only the service corresponding to the grip description and working mode must be upgraded.

On the down side, there are some things to point, tough.

First, the WSDL-to-C structures de-serializer must be adapted to any eventual new versions of WSDL. The implemented parser, for example, is compatible with WSDL 1.1. However, WSDL 2.0 has a different structure. Consequently, the WSDL-reading service must be re-implemented. In this case, it cannot be done as a service, because it is built-in. So, here it is obligatory to shut down the devices and re-program its code.

Second, the memory usage is, obviously, bigger than the static DPWS. Embedded memory on devices is short, so is important to try to consume as fewer resources as possible. The static DPWS creates fewer structures in memory, knowing its attributes and types from the variable declarations. On the dynamic DPWS, for example, an `int number` declared in the static DPWS must be a `structure variable` with a string called “name” with the value “number”, the string variable called “type” with the value “integer”, and the value itself. Being a 4-to-1 number of variables in memory to declare just one simple variable, a service with a big number of types will accuse this difference in embedded memory.

# Chapter 7

# SDPWS – Case Study

Contents

## *7.1 – Objective*

An SDPWS implementation was made. It was used a sample from the DPWS toolkit, the lighting sample. The objective was to use the SDPWS, i.e., the device was both client and server, it used the ERS, and the lighting service was re-created using the dynamic service. The priority objective of the experiment was the compatibility between the normal DPWS lighting service and the new dynamic one.

## *7.2 – SDPWS Implementation*

As already said, a re-creation of the lighting sample was made. It was used the Simple Service approach. The original WSDL service is overall very simple, and that's why it was the chosen service to make the experience.

Explaining the service, it has a *SimpleType* named *PowerState* which values can be either *ON* or *OFF*. Then, it has three messages, a *SwitchMsg* (on-way) that takes one argument (ON/OFF), and a two-way Operation, that are in reality two messages, a *GetStatusReqMsg* (request) message and a *GetStatusRespMsg* (response) message. These are used to ask the lamp (server) for its status.

Finally, the PortType existent is named *SwitchPower*. It includes two Operations, one called Switch that uses the *SwitchMsg*, and the other called *GetStatus* that uses the two other messages.

It was this information that was recreated in the dynamic service. The dynamic service was created with the proper implemented procedures but these procedures' calling were hard-coded, because the WSDL reader was not implemented at the time the experience was made. The creation hard-code is presented next:

```
enum1 = enumeration_create();
enumeration_add_value(enum1, "ON");
enumeration_add_value(enum1, "OFF");
rest1 = restriction_create(GEN_ENUMERATION, (void*)enum1);
type1 = simple_type_create("PowerState", GEN_TOKEN);
simple_type_add_restriction(type1, rest1);
```

In the sample, we can see the *PowerState* Type creation. Because this type accepts the constant values *ON* and *OFF* (so it is an enumeration), that was also created, and is called a restriction, as can be seen in figure 6.1. The other elements were created in an analog method. An element containing the Simple Type was created:

```
element1 = element_create("Power", GEN_ARG_TYPE_SIMPLE,
(void*)type1);
```

, where `type1` is the Simple Type created before. The element is called “Power” because it represents the power status of the lamp.

Next, the message must be created. The following code creates and inserts the respective element on message parts, and then creates the messages.

```
part1 = message_part_create("PowerOut", element1);
part2 = message_part_create("PowerIn", element1);
msg1 = message_create("SwitchMsg");
message_add_part(msg1, part1);
msg2 = message_create("GetStatusReqMsg");
msg3 = message_create("GetStatusRespMsg");
message_add_part(msg3, part2);
```

The first message part represents a message that deliveries the Power Status, i.e., sends the Power Output, and so its name is PowerOut. The second one has identical logic. The messages have its names and can have parts, depending on if it is needed any element value. Notice that the requesting status message (`GetStatusReqMsg`, `msg2`) has no message part in it, as it is only a request of a value, so it doesn’t need o transport any element.

Now, the operations which contain the messages must be created:

```
op1 = operation_create("Switch", msg1, 0);
op2 = operation_create("GetStatusReq", msg2, 0);
op3 = operation_create("Status", 0, msg3);
op4 = operation_create("GetStatus", msg2, msg3);
```

There are four operations represented here. The creating function receives three arguments: its name, and the respective request and response messages. So, the first operation is an one-way operation that asks to switch the power ON or OFF. The second one asks for the status of the lamp (server) in a one-way operation, and the third operation is the respective server answer, and is also a one-way operation. These operations 2 and 3 can be in only one 2-way operation. The fourth operation implements it, as it is an operation that waits for the server’s answer.

The next step in the Simple Service creation is the Port Types creation. Is here that the operations will be inserted in. The code presented next shows how it is done. The process is very identical to the previous ones.

```
port1 = port_type_create("SwitchPower");
port_type_add_operation(port1, op1);
port_type_add_operation(port1, op4);
```

As seen, Port Types’ creation is very simple, just giving a name to it and inserting the intended operations.

Finally, the Simple Service itself is created. This only contains one Port Type, but it could be as much as it were needed. Obviously, the Service must have identifiers, that are passed between components for them to know which service is being refered.

```
service_ns.ns_prefix = "lit";

service_ns.ns_uri = "http://www.schneider-
electric.com/DPWS/2006/03/Training/Lighting";

wsdl.target_ns = "http://www.schneider-
electric.com/DPWS/2006/03/Training/Lighting";

wsdl.location = "http://wsdl.schneider-
electric.com/Lighting.wsdl";

serv = simple_service_create("http://www.schneider-
electric.com/DPWS/2006/03/Training/Light1", service_ns, wsdl);

simple_service_add_port_type(serv, port1);
simple_service_add_port_type(serv, port2);
```

As seen, the Simple Service has some properties, a name (declared in the Simple Service creation procedure `simple_service_create`), a namespace prefix and URI (*Uniform Resource Identifier)*, and a WSDL namespace and location. These properties are the same of the normal Service, and some of them are used in the already explained generated path to reach a specific element, like the service name. Also, the WSDL-related ones exist only for compatibility with the respective static DPWS service, as they are obsolete in dynamic-only DPWS services.

The created Simple Service scheme can be seen in figure 7.1.

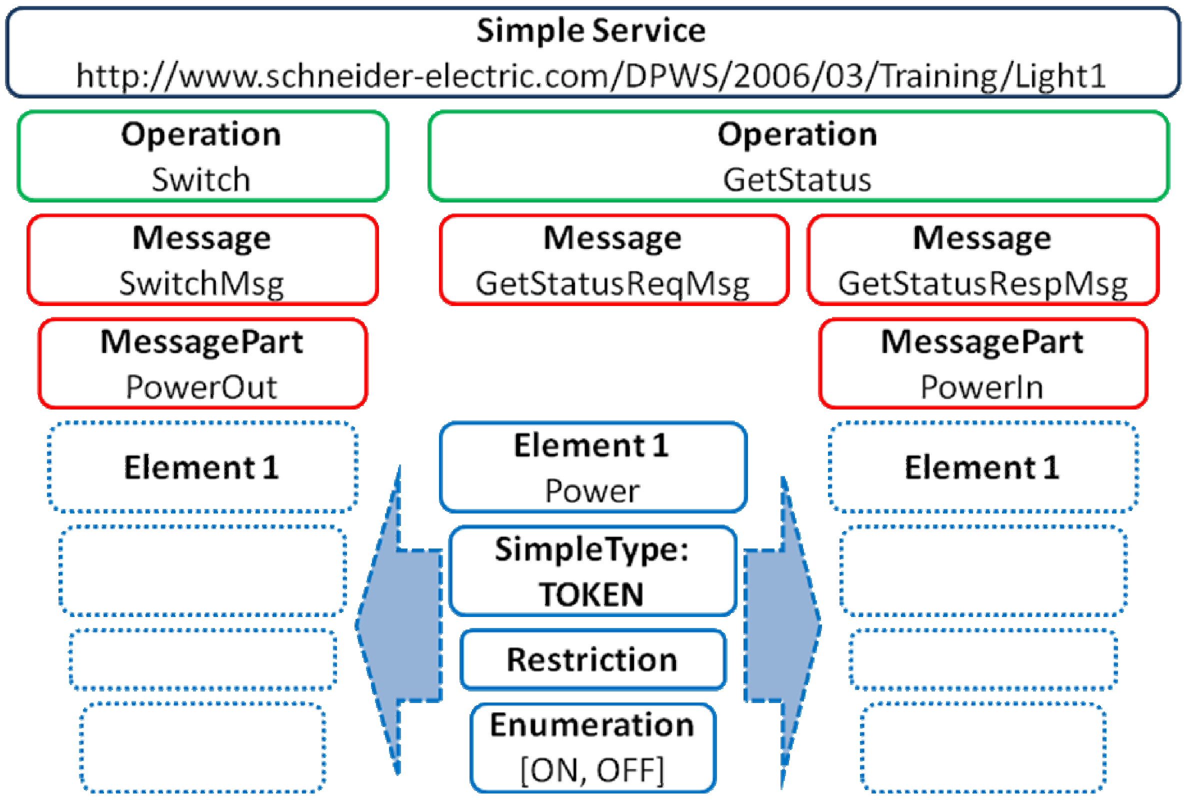


**Figure 7.1 – Implemented Simple Service**

The Event Router-Scheduler was used to the communication between modules. This means the module-programming way was used.

Only two modules were implemented for this example, as it is only to demonstrate it by the simplest way. The code of modules' creation is shown next.

```
module = module_create(1, ERS_USER_INTERFACE);
if(module->id <= 0) {
  printf("\nError creating an module_type!\n");
if(module->id == -1);
  printf("Could not create a list!\n");
if(module->id == -2);
  printf("Could not create a semaphore!\n");
}

module_create(2, ERS_COMMUNICATION);
```

The module creation is very simple. Each module must have two parameters: an ID, that is a unique identifier (number), and a type. The existent types must exist in a proper ERS array. In this case, the two modules are the User Interface module and the Communication module. Notice that the ERS module does not need to be created (obviously), as it is built-in the application. Although the creation of modules is done with the calling of the proper function `module_create`, it can be useful to put some extra code like the one seen on the creation of the first (User Interface) module, for errors-catching, like memory shortage.

Next is a self-explaining piece of code used. It belongs to the dynamic server, and this is specifically the message processing code:

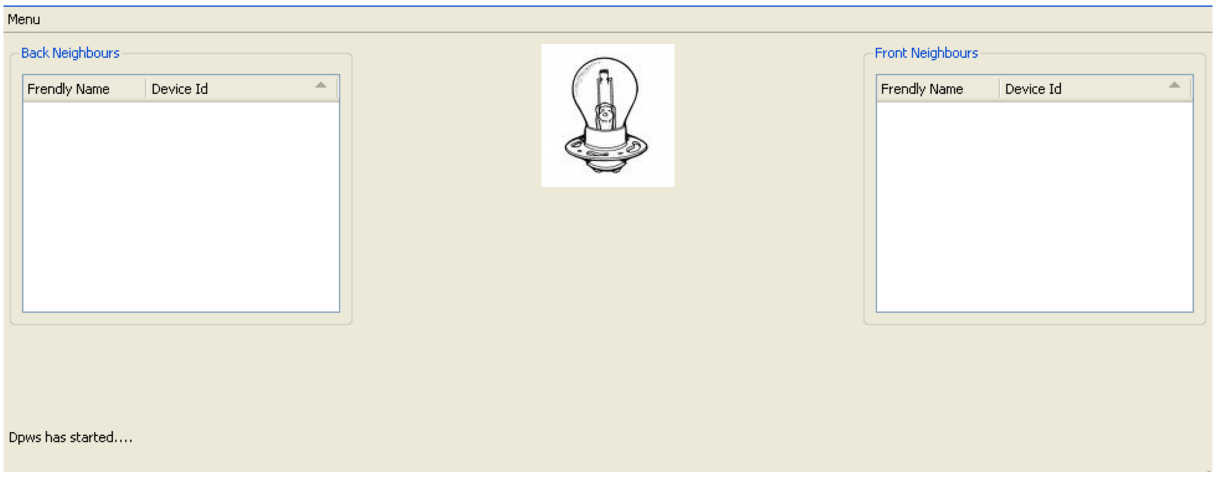
```
void handleMsg(){
  (…)
  while(1){
    msg=event_get(1,ERS_BLOCK);
    switch (msg.action) {
    case DPWS_FAILURE:
      exit(1);
    case RECEIVED_DPWS_MSG:
      controler=(controller_info*)msg.parameters;
      if (!strcmp(controler->service_name, "http://www.schneider-
electric.com/DPWS/2006/03/Training/Light1"))
      {
        elem1 = controller_get_element(controler,"Power");
        value=(char*)element_get_value(elem1);
        printf("->light %s device %d \n", value, controler->device_id);
      }
      controller_info_free(controler);
      break;
    case RECEIVED_RESPONSE_DPWS_MSG:
      event_send(1,ERS_HI,msg);
      break;

    }
  }
}
```

As shown, the lamp module is blocked waiting for a message from the communication module (`msg = event_get(2, ERS_BLOCK)`). The program only advances when a new message is available. Then it checks the action intended to be executed: if it is a failure message, it quits the application, if it is a DPWS message from the communication module, it checks if it is an existent service ("http://www.schneider-electric.com/DPWS/2006/03/Training/Light1") and if it is so, retrieves the value intended and writes on the console. Finally, if it is a message with an answer event, it forwards the message to the User Interface module (module 1), with high-priority (`event_send(2, ERS_HI, msg)`).

As already told, Graphical User Interface (GUI) module was created to be easy to understand the new features. The GUI was started to show the procedures happening with the background DPWS services. This implementation was done by Alexandre Rodrigues.

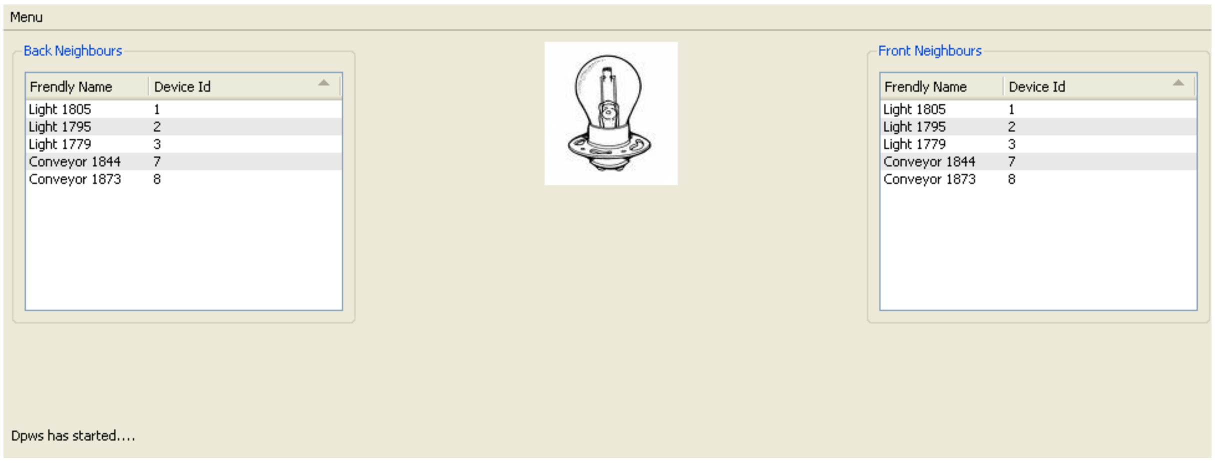
A normal DPWS lighting server service was put running, alone. So, there were no devices. This s shows in figure 7.2, as there are no neighbors.



Dpws has started....

**Figure 7.2 – example GUI with no neighbors detected**

Then, some dynamic services were started, and the devices were found, as seen in figure 7.3.



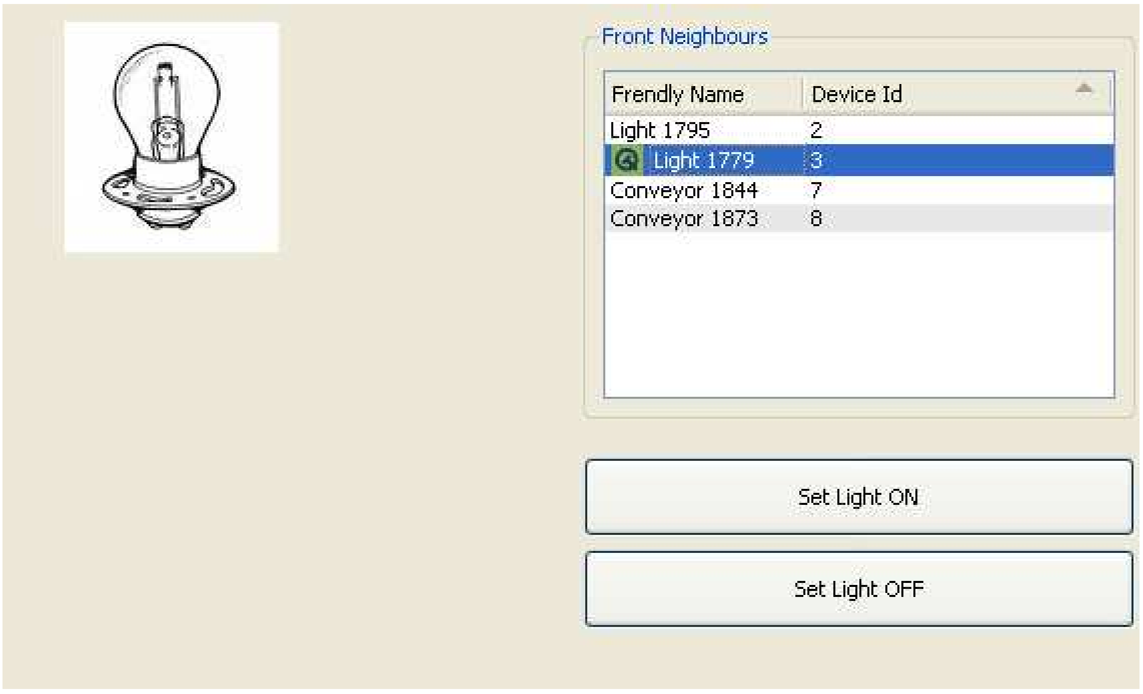
| Frendly Name | Device Id |
|---|---|
| Light 1805 | 1 |
| Light 1795 | 2 |
| Light 1779 | 3 |
| Conveyor 1844 | 7 |
| Conveyor 1873 | 8 |

| Frendly Name | Device Id |
|---|---|
| Light 1805 | 1 |
| Light 1795 | 2 |
| Light 1779 | 3 |
| Conveyor 1844 | 7 |
| Conveyor 1873 | 8 |

Dpws has started....

**Figure 7.3 – example GUI with some neighbors detected**

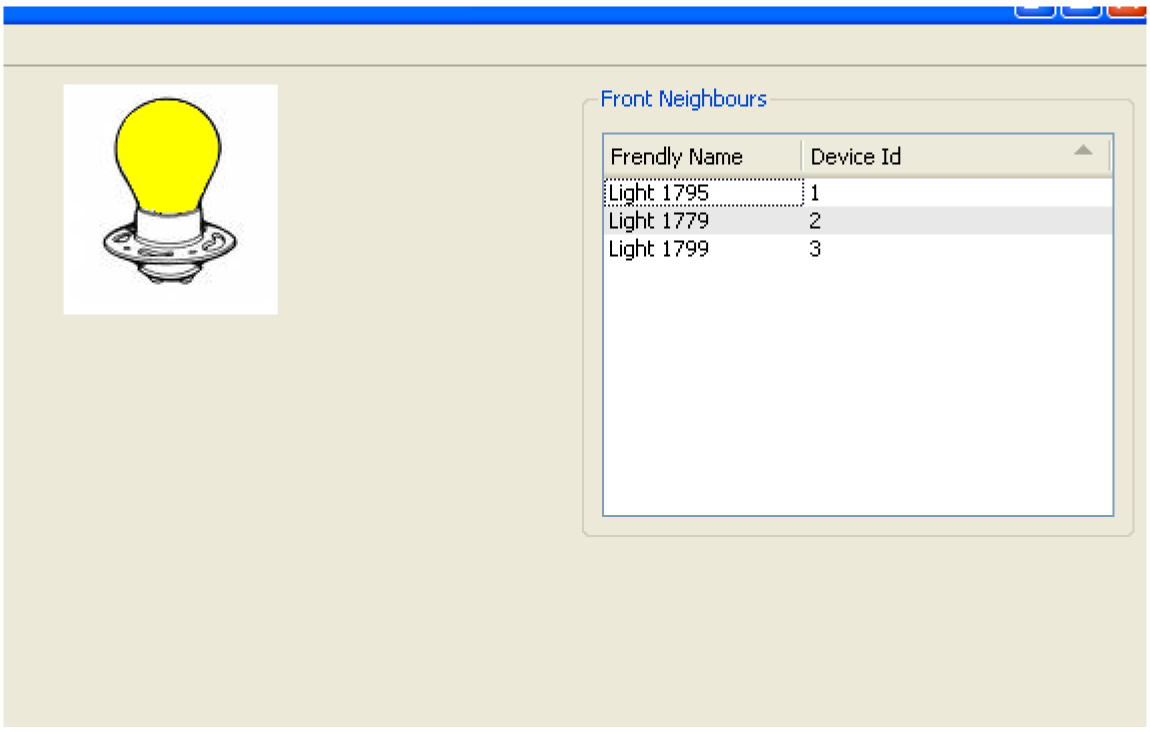
The next step was to select one neighbor, with the respective operational buttons appearing on the GUI. This is illustrated in figure 7.4.



| Frendly Name | Device Id |
|---|---|
| Light 1795 | 2 |
| Light 1779 | 3 |
| Conveyor 1844 | 7 |
| Conveyor 1873 | 8 |

**Figure 7.4 – neighbor selected and respective operating buttons**

Finally, a Switch ON order was given, with the virtual lamp turning ON. This is illustrated in figure 7.5.



| Frendly Name | Device Id |
|---|---|
| Light 1795 | 1 |
| Light 1779 | 2 |
| Light 1799 | 3 |

**Figure 7.5 – light turned ON**

To finalize, next is shown the content of the SOAP envelope passed by the application. It can be seen the service name, the port type and the operation, as well as the intended value.

```
xmlns:lit="http://www.schneider-
electric.com/DPWS/2006/03/Training/Light1">
<SOAP-ENV:Header>
  <wsa:To>
    http://169.254.184.154:9876/d2ee4d54-9853-11dc-8ba9-
001302e329dc
  </wsa:To>
  <wsa:Action>
    http://www.schneider-
electric.com/DPWS/2006/03/Training/Light1/Switch/Power
  </wsa:Action>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
  <wsh:Power Value="ON" />
</SOAP-ENV:Body>
```

The results were very satisfactory, because a most complete possible version of the SDPWS idea was successfully implemented.

# Chapter 8

# Conclusion

The results of this work show the feasibility of the improved support towards easy service creation, that enables novel device networking architectures and holds the promise of ease the development, integration, deployment, maintenance and lifecycle management of devices and services. Advantages of such a system are clear, services can be added directly and components of the device can be changed with minimal reconfiguration and without the need of re-deploy components.

The adoption of a “bio-inspired” modular structure makes possible to design and develop modules with distinct and independent functions but complementary to each other, forming complex, intelligent and social components. The resulted component’s structure may help in decreasing the development time and effort in the integration into the system. The prototype development shows the feasibility and features of the concept, providing the possibility to develop reusable and functional modules and deploy them into service-oriented components. Developers just don’t need to care with inter-module synchronization, which is one of the main problems of real-time interaction.

The dynamic creation of new services allows machine reconfiguration and reprogram without shutting down the system, and eases the maintenance. This strongly increases the shop-floor agility.

# Future Work

The main challenge is to deploy these techniques in real devices and proof their applicability in industrial automation systems. Other requirements are to stabilize the implementation and improve the interaction concepts among distributed components to meet the objectives of flexible production and automation.

Also, it is needed to enhance both concept and development of the ERS and the Dynamic Service.  A special case is to enhance the flexibly in the deployment of components and its modules, by developing a specification of metadata for modules that would permit the creation of them without worrying about how the information comes from the other modules.

# References

[1] Jammes, F., Mensch, A., Smit, H., 2005, Service-oriented device communications using the devices profile for web services, Proc. of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, ACM Press, 1-8

[2] Jennings, N. R., Wooldridge, M., 1998, Applications of intelligent agents, Springer-Verlag New York, Inc., 3-28

[3] Schoop, R., Neubert, R., Colombo, A., 2001, A multiagent-based distributed control platform for industrial flexible production systems, 27th Annual Conference of the IEEE Industrial Electronics Society, 1: 279-284

[4] Colombo, A., Neubert, R., Schoop, R., 2001, A solution to holonic control systems, Proceedings of the 8th IEEE International Conference on Emerging Technologies and Factory Automation, 2: 489-498

[5] Colombo, A., Jammes, F., Smit, H., Harrison, R., Lastra, J., Delamer, I., 2005, Service-oriented Architectures for Collaborative Automation, 32nd Annual Conference of IEEE Industrial Electronics Society, 6

[6] Moran, N. A., 2006, Symbiosis, Current Biology, Cell Press, Elsevier Inc., 16/20: 866-871

[7] Wooldridge, M., 2002, Introduction to MultiAgent Systems

[8] Mendes, J. M., Leitão, P., Colombo, A. W., Restivo, F., 2008, Service-oriented Control Architecture for Reconfigurable Production Systems, to appear in the Proceedings of the 6th IEEE International Conference on Industrial Informatics

[9] Gray, H., 2000, Gray's Anatomy of the Human Body, 20th Edition (Original by Philadelphia: Lea and Febiger, 1918), New York: Bartleby.com

[10] Cachapa, D, Colombo, A, Feike, M, Bepperling, A, "An approach for integrating real and virtual production automation devices applying the service-oriented architecture paradigm", Proc. of the IEEE Conference on Emerging Technologies & Factory Automation, pp. 309-314, 2007

[11] Ribeiro, L, "A Diagnostic Infrastructure for Manufacturing Systems", 2007, Master Thesis, New University of Lisbon

[12] Barisic, D, Krogmann, M, Stromberg, G, Schramm, P, "Making Embedded Software Development More Efficient with SOA," ainaw,pp.941-946, 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07), 2007

[13] European 6th Framework Project. PROMISE - Product Lifecycle Management and Information Tracking using Smart Embedded Systems. http://www.promise.no/, 2005-2008

[14] SIRENA. Service Infrastructure for Real-time Embedded Networked Applications. http://www.sirena-itea.org, 2005

[15] SODA. Service Oriented Device and Delivery Architecture. http://www.soda-itea.org, 2006

[16] European 6th Framework Project. SOCRADES – Service Oriented Cross-layer Infrastructure for Distributed smart Embedded Devices. http://www.socrades.eu/, 2006

[17] Hayashi, H., 1993, The IMS International Collaborative Program, Proceedings of the 24th ISIR, Japan Industrial Robot Association

[18] J. Barata, L. Ribeiro and A. Colombo, "Diagnosis using Service Oriented Architectures", Proceedings of the IEEE International Conference on Industrial Informatics, Vol. 2, pp.1203-1208, 2007