DPWS Core version 2.1

User Guide

Version 1.0 – April 14, 2009

DOCUMENT HISTORY

Version	Date	Editors	Comments
0.9	31/03/2009	A. Mensch S. Rouges	Initial version covering all the features of DPWS Core version 2.1.0
1.0	14/04/2009	A. Mensch	Addition of reviewers feedback

TABLE OF CONTENTS

Introduction **7**

The DPWS Core toolkit 7

The Devices Profile for Web Services 7

Document guide 9

Definitions and notations 10

namespaces and prefixes in use 10

Style conventions 10

References 10

DPWS Core development principles **12**

Web Services and DPWS principles 12

The SOAP protocol 12

Main message exchange patterns 13

Mapping SOAP messages to operation invocations 14

Marshalling/unmarshalling principles 14

The Web Services Description Language 15

The SOAP binding 16

Code generation principles 17

WS-Addressing introduction 18

WS-Discovery introduction 19

WS-Eventing introduction 19

The DPWS device and hosted services model 20

Development process use cases 21

Services and device development 21

Pure client development 21

Peer-to-peer client development 21

Asynchronous and eventing client development 22

Services and device development **23**

Development process overview 23

Service identification guidelines 23

Service interface specification in WSDL 24

PortTypes, operations and messages definition 24

Types definition 26

Bindings definition 26

Code generation 27

wsdl2h 28
soapcpp2 29
Service implementation 30
Implementation of service functions 30
Use of events 31
Generated notification functions 32
Sending event notifications 32
Service deployment and device configuration 33
Role of the registry 33
The registry object model 34
Devices and hosted services configuration 36
Persistent information management 39
Implementing the server architecture 40
Stack initialization 40
Configuring the server and its listeners 41
Implementing the server loop 42
Exiting the server loop 43
Compiling and linking 43
The DPWS Core libraries 43
Generated server and application files 44
Pure client development 45
Development process overview 45
Code generation 45
Code generation tools 45
Generated stubs 45
Client implementation 46
Client-side initialization 47
Devices and services discovery and metadata access 47
Service stub invocation 50
Request context clean up 52
Compiling and linking 53
The DPWS Core libraries 53
Generated client and application files 53
Peer-to-peer client development 54
Development process overview 54
Code generation 54
Integrating server and client code 55

Cache initialization and update 55
Invoking a remote operation from a service function 56
Compiling and linking 56
The DPWS Core libraries 56
Generated server, generated client and application files 56

Asynchronous and eventing client development 58

Development process overview 58
Code generation 58
Handler implementation 59
Server configuration 60
Handler deployment 60
Event subscription management 61
Implementing the server architecture 62
Compiling and linking 62
The DPWS Core libraries 62
Generated handler and application files 62

Advanced features 64

Customizing code generation 64
Mapping XML Schema types to specific C types 64
Editing the gSOAP annotated header file 65
Implementing custom marshallers/unmarshallers 65
Generic invocation 66
EPX API 66
Generic stubs 69
Generic skeletons 69
Compiling and linking 70
Advanced registry features 71
Dynamic registry modification API 71
Advanced cache features 71
Cache content control 71
Lifecycle callbacks 72
XML configuration 72
Registry configuration format 73
Cache configuration format 74
Subscription Manager configuration format 75
Usage 75
Compiling and linking 76

Dynamic deployment 77
Server-side support 77
Client-side support 79
Multiple network interfaces and IP protocols 80
Advanced stack customization 81
Operation timeouts 81
Connection keep-alive 82
HTTP chunked mode 83
MTOM support 83
Basic Profile 1.1 support 84
HTTP GET support 85
Advanced eventing features 86
Subscription management configuration 86
Monitoring event delivery failures 86
External Web server integration 87
Server configuration 87
HTTP request processing 88
Compiling and linking 89
Appendices **90**
Error management 90

INTRODUCTION

THE DPWS CORE TOOLKIT

The DPWS Core (DC) toolkit provides a runtime environment and associated code generation tools that enable the development and deployment of Web Services implemented in the C language. Because it uses compact and portable C code for both its libraries and the generated code, the DC toolkit is well-suited for the development of Web Services applications in small embedded systems. To reinforce its focus on embedded devices, the DC toolkit also supports the Devices Profile for Web Services [**DPWS**], a specification that extends traditional Web Services with network discovery, plug-and-play and asynchronous messaging features. DPWS support means that devices and applications developed using the DC toolkit will automatically benefit from increased communication capabilities and interoperability with other devices, mobile phones and PDAs, home and office PCs and enterprise systems.

Main features of the DC toolkit include:

- A standard Web Services stack and code generators based on the gSOAP [**gSOAP**] open-source software.
- A runtime container that allows the implementation and deployment of devices and Web Services in accordance with the DPWS model.
- DPWS-compliant built-in services that provide support for network discovery, device metadata access and event subscription management in client applications.

Advanced features include:

- Support for multiple network adapters and multiple IP protocols.
- XML-based and dynamic configuration of devices and services.
- Customizable transport layer supporting connection keep-alive, HTTP chunked mode [**HTTP/1.1**], MTOM attachments [**MTOM**], and a pluggable architecture allowing the replacement of the internal HTTP server by an external one.

THE DEVICES PROFILE FOR WEB SERVICES

A proposal for using Web Services protocols for device networking, entitled "Devices Profile for Web Services" [**DPWS**], was submitted in May 2004 by a group of companies. This subset of the Web Services protocol suite was originally designed to become the next major version of the popular UPnP (Universal Plug-n-Play) Device Architecture [**UPnP**]. It may still be eventually proposed as such, but for reasons of market strategy related to the lack of backward compatibility between the UPnP and DPWS protocol stacks, no date is set for this transition. Meanwhile, DPWS is set to become an OASIS standard in 2009.

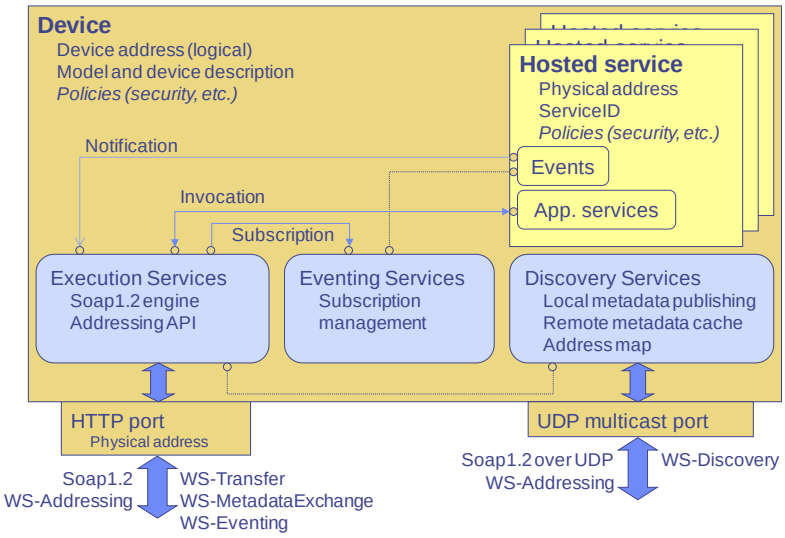
The advantages of using Web Services for device-to-device and device-to-workstation communication relate both to operational aspects and to the development process:

- Unify protocols so that a single stack communicates with both devices and other Web Services.
- Enable seamless integration of device networks, e.g. in plant floors, into enterprise-wide information systems.
- Unify developer experience, knowledge and tools.

It may also be noted that DPWS is natively supported by Windows Vista and Windows 7. This makes devices compliant with the DPWS specification easily discoverable by PCs running one of these operating systems.

DPWS provides a small and efficient framework for peer-to-peer device interactions, fully compatible with the Web Services family of specifications.

The DPWS specification defines an architecture that distinguishes two types of services: devices and hosted services. Devices play an important part in the discovery and metadata exchange procedures. Hosted services are application-specific Web services that provide the functional behavior of the device. They rely on their hosting device for discovery. The deployment of hosted services on a DPWS device is the primary extensibility mechanism provided by the specification.



DPWS also specifies a set of built-in services:

- Discovery services [**WS-Discovery**]: these services are used by a device connected to a network to advertise itself and by clients to discover devices. WS-Discovery uses SOAP over UDP [**SOAP-over-UDP**] and a multicast address to broadcast and listen to the discovery messages.
- Metadata exchange services [**WS-Transfer**] [**WS-MetadataExchange**]: these services can be used by a client to retrieve a device metadata, including the device hosted services, and the metadata of those hosted services, such as WSDL [**WSDL 1.1**] or XML Schema [**XML Schema, Part 1**] [**XML Schema, Part 2**] documents.
- Events publish/subscribe services [**WS-Eventing**]: these services combine extensions of application-defined services with built-in services, and allow clients to subscribe to asynchronous messages (events) produced by a given application-defined service and to manage the resulting subscriptions.

A DPWS client endpoint typically performs the following tasks:

- Discovery: discover relevant devices on the network. Discovery is based on device types and scopes, which can be used to characterize a device with application-specific, hierarchical information (typically, a geographical location or a network location). Discovery is limited to the devices, and does not involve services.
- Description: retrieve the device description, get the list of hosted services, and select relevant services and service descriptions from this list. Service description relies on the standard WSDL language, which is supported by a large number of development tools.
- Control: invoke operations on selected services to control the device.
- Eventing: subscribe to the service event sources.

DOCUMENT GUIDE

This document is the user guide for the DPWS Core toolkit. It covers the general principles of Web Services programming with the DC toolkit, and also provides detailed guidance on the use of most of the toolkit features. The intended audience for this guide includes software architects and software developers.

After this introduction, the next chapter, "DPWS Core development principles", provides a high-level view of Web Services and DPWS development principles, including an overview of the main standards and technologies involved in the DPWS specification. The section is concluded by a description of four development scenarios that should cover most of the DPWS Core toolkit use cases.

The four following chapters describe in details the development process for each of the previously introduced scenarios:

- The "Services and device development" scenario describes the development of a device that only provides services to clients.
- The "Pure client development" scenario describes the opposite case of an application which only consumes services from service providers.
- The "Peer-to-peer client development" scenario describes the development of a device that also acts as a service consumer towards other providers.
- The "Asynchronous and eventing client development" addresses the specific case of a client that sets up a server architecture to receive asynchronous messages, such as event notifications.

The last chapter of the guide describes the "Advanced features" of the DPWS Core toolkit. It is not particularly structured, and should be considered as a set of recipes that can be read in (almost) no specific order.

Readers with experience in Web Services development may skip the first chapter, even if it contains important information about DPWS and some of its supporting specifications that are less "mainstream" than vanilla Web Services. Depending on their use case, they may want to go directly to the "Services and device development" or the "Pure client development" chapter. The two other development scenarios are not recommended as entry points for first-time readers, as they strongly depend on information provided in the two previous chapters.

Returning readers looking for a specific issue resolution may want to jump directly to one of the "Advanced features" sections, such as:

- "Customizing code generation": a deeper look into the various code generation features of the DPWS Core toolkit.
- "Generic invocation": a mechanism that supports efficient SOAP messages processing without requiring code generation.
- "XML configuration": a feature that provides an XML-based configuration language for initializing a DPWS Core application.
- "Dynamic deployment": a feature that supports dynamic reconfiguration of a running platform through the modification of the set of devices and services that it executes.
- "Advanced stack customization": advanced features such as connection keep-alive, operation timeout management, MTOM support and HTTP chunked mode support.

This User Guide is complemented by the DPWS Core API reference manual [**DC API**], which provides a detailed description of all the functions and structures used to develop an application with the DPWS Core toolkit.

DEFINITIONS AND NOTATIONS

NAMESPACES AND PREFIXES IN USE

The following table shows the namespaces and associated prefixes that are used in this document. The choice of the prefixes is not semantically significant, and is overwritten by declarations found in actual XML documents.

Namespace prefix	Namespace URI
soap	http://www.w3.org/2003/05/soap-envelope
wsa	http://schemas.xmlsoap.org/ws/2004/08/addressing
dpws	http://schemas.xmlsoap.org/ws/2006/02/devprof
wse	http://schemas.xmlsoap.org/ws/2004/08/eventing
wsdl	http://schemas.xmlsoap.org/wsdl/
xsd	http://www.w3.org/2001/XMLSchema
wsoap	http://schemas.xmlsoap.org/wsdl/soap/
wsoap12	http://schemas.xmlsoap.org/wsdl/soap12/
xop	http://www.w3.org/2004/08/xop/include

In addition, the 'tns' prefix is used in some WSDL and XML Schemas samples to refer to the target namespace being defined by the sample document under consideration. It is not semantically significant either.

STYLE CONVENTIONS

This document uses the following conventions to identify special paragraphs:

```
Code and data file snippets use this style.
```

Warning: complex or unclear aspects that are often the source of errors are identified with this style.

Advanced use:
Advanced features that do not need to be fully understood when getting started with the DPWS Core toolkit are described using this style.

In addition, references to API functions, structs or constants, as well as references to XML elements or attributes use `this font`.

REFERENCES

[BP 1.1]: K. Ballinger, et al, "Basic Profile Version 1.1", August 2004. (See http://www.ws-i.org/Profiles/BasicProfile-1.1.html)

[DC API]: "DPWS Core API reference manual", March 2009. Available in the DPWS Core distribution package.

[DPWS]: S. Chan, et al, "Devices Profile for Web Services", February 2006. (See http://schemas.xmlsoap.org/ws/2006/02/devprof/)

[gSOAP]: R. van Engelen, "SOAP C/C++ Web Services". (See http://www.cs.fsu.edu/~engelen/soap.html). Note that this site contains the latest version of the gSOAP documentation, which is not the one the DPWS Core toolkit is built upon.

[**gSOAP Guide**]: R. van Engelen, “gSOAP 2.7.6 User Guide”, September 2005. Available in the DPWS Core distribution package.

[**HTTP/1.1**]: R. Fielding, et al, "Hypertext Transfer Protocol -- HTTP/1.1", June 1999. (See http://www.ietf.org/rfc/rfc2616.txt)

[**MTOM**]: N. Mendelsohn, et al, "SOAP Message Transmission Optimization Mechanism", January 2005. (See http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/)

[**RFC 4122**]: P. Leach, et al, "A Universally Unique IDentifier (UUID) URN Namespace", July 2005. (See http://www.ietf.org/rfc/rfc4122.txt)

[**SOAP 1.1**]: D. Box, et al, “Simple Object Access Protocol (SOAP) 1.1", May 2000. (See http://www.w3.org/TR/2000/NOTE-SOAP-20000508/)

[**SOAP 1.2, Part 1**]: M. Gudgin, et al, "SOAP Version 1.2 Part 1: Messaging Framework", June 2003. (See http://www.w3.org/TR/2003/REC-soap12-part1-20030624/)

[**SOAP 1.2, Part 2, Section 7**]: M. Gudgin, et al, " SOAP Version 1.2 Part 2: Adjuncts, Section 7: SOAP HTTP Binding", June 2003. (See http://www.w3.org/TR/2003/REC-soap12-part2- 20030624/#soapinhttp)

[**SOAP-over-UDP**]: H. Combs, et al, "SOAP-over-UDP", September 2004. (See http://schemas.xmlsoap.org/ws/2004/09/soap-over-udp)

[**UPnP**]: “UPnP Device Architecture 1.0”, April 2008. (See http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf)

[**WS-Addressing**]: D. Box, et al, "Web Services Addressing (WS-Addressing)", August 2004. (See http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/)

[**WS-Discovery**]: J. Beatty, et al, "Web Services Dynamic Discovery (WS-Discovery)", April 2005. (See http://schemas.xmlsoap.org/ws/2005/04/discovery)

[**WSDL 1.1**]: E. Christensen, et al, "Web Services Description Language (WSDL) 1.1", March 2001. (See http://www.w3.org/TR/2001/NOTE-wsdl-20010315)

[**WSDL Binding for SOAP 1.2**]: K. Ballinger, et al, "WSDL Binding for SOAP 1.2", April 2002. (See http://schemas.xmlsoap.org/wsdl/soap12/)

[**WS-Eventing**]: L. Cabrera, et al, "Web Services Eventing (WS-Eventing)", August 2004. (See http://schemas.xmlsoap.org/ws/2004/08/eventing/)

[**WS-MetadataExchange**]: K. Ballinger, et al, "Web Services Metadata Exchange (WS-MetadataExchange)", September 2004. (See http://schemas.xmlsoap.org/ws/2004/09/mex/)

[**WS-Transfer**]: J. Alexander, et al, "Web Service Transfer (WS-Transfer)", September 2004. (See http://schemas.xmlsoap.org/ws/2004/09/transfer/)

[**XML Schema, Part 1**]: H. Thompson, et al, "XML Schema Part 1: Structures", May 2001. (See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/)

[**XML Schema, Part 2**]: P. Biron, et al, "XML Schema Part 2: Datatypes", May 2001. (See http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/)

DPWS CORE DEVELOPMENT PRINCIPLES

WEB SERVICES AND DPWS PRINCIPLES

The provision and consumption of Web Services can be considered from two different perspectives:

- An abstract level, in which a Web Service provider exposes some business functionality as an interface, or "contract", encapsulating and hiding the implementation details. Each interface features one or several related operations relevant to the exposed business functionality. Web Services consumers use the published contract to access to the business functionality.
- A concrete level, in which each Web Service operation invocation is translated into one message exchange on the wire between the Web Service consumer and the Web Service provider.

The role of the DPWS Core (DC) toolkit is to bridge the gap between the two perspectives, by providing both a runtime environment and code generation tools allowing:

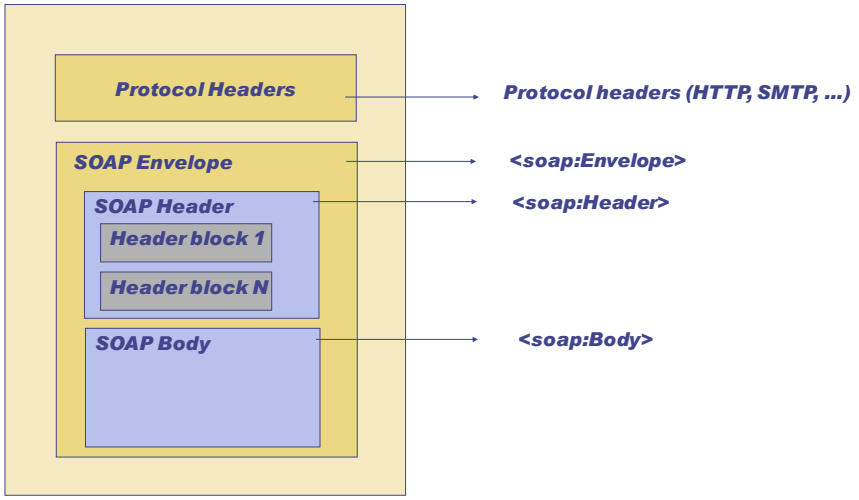
- Web Services providers to focus on the definition of the Web Services contracts and the implementation of the business functionality, without worrying about the message transport aspects and the translation from the wire representation of messages to the corresponding operation invocation.
- Web Services consumers to easily map the published Web Services contracts to remote functions that can be directly invoked from client code, again without specific knowledge about the underlying messaging mechanisms.

Although the DC toolkit goal is to hide the complexity of Web Services mechanisms and protocols from the developer, this section nevertheless provides an overview of the main principles and messaging mechanisms involved in Web Services applications, as understanding those mechanisms may prove useful when tuning or debugging an application.

THE SOAP PROTOCOL

Web Services use SOAP [**SOAP 1.1**] [**SOAP 1.2, Part 1**] as their messaging protocol. It is a simple, extensible, XML-based protocol that represents each message as a **`soap:Envelope`** element containing an optional soap:Header element and a mandatory **`soap:Body`** element:

- The **`soap:Header`** element can contain one or several header blocks, which are XML elements used to control how the message is transmitted from its sender to its intended receiver, possibly going through some intermediate nodes. The addition of header blocks is the primary extensibility mechanism for the SOAP protocol: additional specifications use this mechanism to normalize the use of header blocks for routing, reliability, security…
- The **`soap:Body`** element contains the application-specific message payload. It is normally defined by each application, the exception being faults (normally sent by a service provider in case of error), which are represented by a predefined **`soap:Fault`** element.



The SOAP protocol is transport-agnostic, and can therefore be used in combination with different transport layers. The set of rules defining the association of the SOAP protocol with a specific transport protocol is called a binding. The SOAP protocol defines an extensible binding framework that provides general rules for specifying new bindings. Most applications use the standard HTTP binding, defined in the SOAP specification [**SOAP 1.1**] [**SOAP 1.2, Part 2, Section 7**], but DPWS also requires the use of the SOAP-over-UDP binding [**SOAP-over-UDP**] for discovery.

There are two versions of the SOAP protocol, which are very similar from the functional point of view, but not interoperable on the wire:

- SOAP 1.1: although not formally a standard, this version is currently the most widely used. It is referenced by several Web Services standards, most notably WS-I Basic Profile 1.1, which is an ISO standard aiming at achieving the broadest interoperability across Web Services stack vendors.
- SOAP 1.2: this version is a W3C standard. It is widely supported by recent Web Services stacks, but not always by older toolkits. The DPWS specification requires devices and hosted services to use SOAP 1.2.

The DC toolkit uses by default the SOAP 1.2 version, as required by the DPWS specification. However, it is possible to configure the DC stack to also support WS-I Basic Profile 1.1 [**BP 1.1**], and thus achieve interoperability with a large range of Web Services tools and applications.

Main message exchange patterns

The SOAP protocol defines a message construct and a processing model for a single exchange between a sender and a receiver. Single exchanges can then be further combined to create more complex message exchange patterns.

The DC toolkit supports three predefined message exchange patterns (MEP):

- One-way: a single message sent by a service consumer to a service provider. This MEP corresponds to the invocation of a service operation with no expected response.
- Request/Response: a request message sent by a service consumer to a service provider, followed by a response message sent by the provider to the consumer. This MEP corresponds to the invocation of a service operation that returns a response.

- Notification: a message sent by an event source (service provider) to an event sink (service consumer). This MEP is less common than the two previous ones and is defined by the WS-Eventing [**WS-Eventing**] specification.

Mapping SOAP messages to operation invocations

Web Services invocation involves mapping the messages described above to actual service operations.

In the case of requests (either one-way or two-way) and notifications, enough information must be provided in the message to allow its dispatch on the receiver side to the appropriate message handler. This information must include:

- The receiver endpoint identification: A service exposes one or more endpoints (URLs) where the service is available. As messages directed to several services can be received through the same Web Services host, each message must contain either the URL of the target service, or some other logical id that can be mapped to the appropriate service by the host.
- The operation identification: a message relates to a specific operation in the target service interface. This operation must therefore be identified in the request message.

Depending on the transport protocols and SOAP extensions used to transmit the message, the above information can take several forms:

- The receiver endpoint can be carried as part of the HTTP request, or as a specific SOAP header block, for instance when using WS-Addressing [**WS-Addressing**].
- The operation identification can be specified in a HTTP header, in a SOAP header block or directly in the message body.

In the case of responses, the amount of required information depends on the way the response is returned to the service consumer:

- When the response is sent back synchronously to the service consumer, e.g. when using the standard HTTP binding, there is no real need for specific information, as the service consumer is waiting for the response and knows which message to expect.
- When the response is sent back asynchronously and needs to be dispatched to an appropriate handler, either an identification of the receiver and the operation or a correlation id referring to the request message may be required.

The DC toolkit uses code generation techniques to encapsulate the construction (on the sender side) and the dispatch (on the receiver side) of messages. The generated code ensures that the appropriate information is associated to each exchanged message.

Marshalling/unmarshalling principles

The second aspect that must be addressed when mapping messages to operation invocations is the transformation between the message payload (the `soap:Body` content) and its native code representation: on the sender side, the operation parameters are serialized (marshalling) as the message payload, and are transformed back (unmarshalling) into parameters on the receiver side before the invocation of the service operation.

There are two general approaches used to map XML documents to native data structures, independently from the programming language in use:

- Generic mapping: in this approach, the XML document content is directly represented in memory as either a generic object structure (e.g. the standard Document Object Model – DOM) or as a stream of "events", each event representing a piece of information in the XML document (XML element start and end, character data, attribute…).
- Specific mapping: In this approach, each XML document is mapped onto a specific native object (C struct, C++, Java or C# class…), depending on the type of the XML document. This approach generally requires code generation to efficiently perform the transformation in both directions.

The DC toolkit supports both approaches, using an event streaming API for the first case and generated code for the second. When starting with the DC toolkit, it is recommended to start with the code generation approach, which is much simpler to use.

The Web Services Description Language

The previous sections have introduced the messaging mechanisms used by Web Services stacks to implement the concrete mapping between operation invocations and messages exchanged over the wire. This section focuses on the abstract Web Services perspective, and more specifically on the language used to describe Web Services interfaces and operations, and to relate those operations to concrete messages.

The Web Services Description Language [**WSDL 1.1**] is a specification of an XML language used for describing services as a set of endpoints exposing their capabilities through operations processing messages. As for SOAP, two versions of WSDL are available:

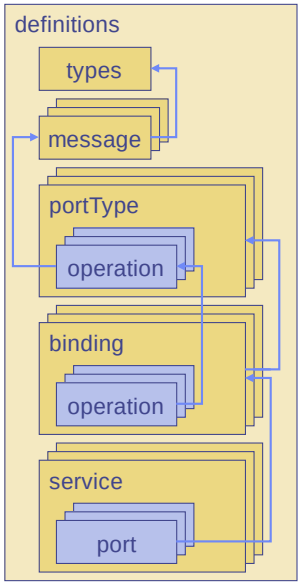
- WSDL 1.1: although not formally a standard, this version is currently the only one supported by most Web Services toolkits, including recent ones. It is referenced by WS-I Basic Profile 1.1, which clarifies and restricts its use to ensure interoperability between applications developed using different Web Services toolkits.
- WSDL 2.0: this version is a W3C standard. It is currently not supported by mainstream Web Services toolkits.

The DC toolkit currently only supports WSDL 1.1.

WSDL 1.1 introduces the following concepts:

- Target namespace: each WSDL document defines a new namespace, which is used to qualify the name of components defined in the document, in a way similar to the one used for XML Schema components.
- Types: they provide the basic data types definitions that are used to build message constructs. Although in theory types can be expressed using different type systems, in practice XML Schema is the only one supported by most tools. WSDL types must therefore be XML Schema [**XML Schema, Part 1**] [**XML Schema, Part 2**] types and elements, which can be either defined inline in the WSDL document or imported from an external XML Schema document.
- Messages: they define the abstract structure of the messages received and sent by the services. A message has zero or more logical parts, each part having a name and being associated to either a XSD type or element.
- PortTypes and abstract operations: portTypes define the interfaces of the services. They are named collections of abstract operations, each operation having a name and being associated with input and/or output messages, as well as possible faults.

- Bindings and concrete operations: while portTypes provide the abstract definition of the service interfaces, bindings define the concrete format and protocol used by a portType operations and messages. A binding references a single portType, and is used to associate format and protocol-specific information to operations and messages defined in the portType. Although WSDL is designed to be extensible and to support numerous binding types, in practice most tools, including the DC toolkit, only support the definition of bindings for SOAP 1.1 and SOAP 1.2 over HTTP [**WSDL Binding for SOAP 1.2**]. Details about these bindings are given below.
- Services and ports: services are defined as a collection of ports, each port representing the association of a network address and a specific binding: for the commonly used SOAP over HTTP bindings, the address must be an HTTP URL.



The production of the WSDL documents describing the services of an application is a critical step in the application development process: the DC toolkit is based on a contract-first approach, and requires WSDL and XML schemas documents as inputs to the code generation both on the client and on the server side.

THE SOAP BINDING

The SOAP binding (either for SOAP 1.1 or SOAP 1.2) is the most commonly used binding in WSDL documents, and the only one currently supported by the DC toolkit. The use of this binding means that messages received and sent on the endpoint associated to the binding are SOAP messages (either SOAP 1.1 or SOAP 1.2 depending on the version of the binding used). The SOAP binding specifies the following information:

- Transport protocol: the transport used by the SOAP messages. In practice only the HTTP transport is supported by most tools.
- Operation action: the action URI associated to each operation. When specified, this URI should be used as the value of the SOAPAction HTTP header. It is intended to be uniquely associated to a given operation and to be used for dispatching incoming request messages on the server side.
- Operation style: this controls how the parts of the operation messages are assembled inside the message Body. The style can be either “document” or “rpc”, and can be specified globally for the binding or individually for each operation. Document style means that each part of the operation messages is serialized as

an XML element inside the message Body. Rpc style means that each part of the operation messages is either a parameter or a return value and appears inside a wrapper element, representing the operation, within the message Body.

- Message encoding: this controls whether message parts are serialized in XML according to their XML schema definition ('literal' use), or whether a specific encoding is used ('encoded' use).

Because the above rules allow for a lot of flexibility in the format of messages, they have induced quite a few interoperability problems. Therefore, the WS-I Basic Profile 1.1 [**BP 1.1**] has added the following restrictions to the SOAP binding:

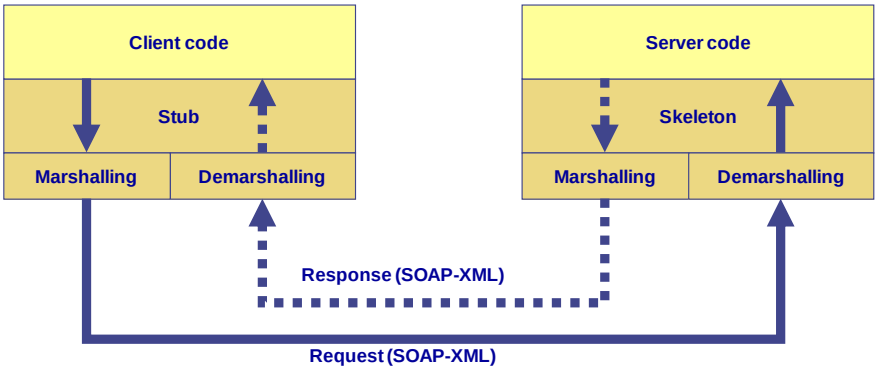
- Only the HTTP transport can be used.
- The SOAPAction HTTP header should not be used for message dispatch on the server side. Rather, all operations in a binding are required to have distinct signatures, where the operation signature is defined as the child element of the Body of the operation input message.
- Only the literal encoding is allowed, in combination with either the 'document' or 'rpc' style.

The BP 1.1 rules apply only to the SOAP 1.1 protocol and binding. The DPWS specification defines the same restrictions for the SOAP 1.2 protocol and binding, with the addition that each service must feature at least a document/literal binding.

In order to maximize interoperability, it is therefore recommended to use document/literal SOAP bindings when developing Web Services with the DC toolkit.

Code generation principles

Code generation is used in Web Services development to bridge the gap between the abstract service interfaces described through WSDL documents and concrete SOAP messages exchanged on the wire. The following diagram shows the relation between user-defined code, on the server and on the client, and generated code:



On the server side, generated code is composed of:

- Skeleton: this code is used by the message server, once it has identified the destination service, to dispatch incoming messages to the appropriate operation handler. It is also used to send back the response message for request/response operation.
- Marshalling/unmarshalling: generated marshalling/unmarshalling code is used by the server to translate the contents of XML messages into native objects; and vice versa.

The user-defined server code consists of functions implementing the service operations. These functions are called by the skeleton, and take as input and output parameters the native objects that are transformed by the marshalling/unmarshalling code.

On the client side, generated code is composed of:

- Stub: this code provides the remote service interface (i.e. set of functions) that can be used by clients to invoke the service operations. Its role is to translate the function invocation into a message and send it to the service, and receive and process the response message (in case of request/response operations).
- Marshalling/unmarshalling: this code has the same role on the client side as on the server side.

The user-defined client code uses the stub generated functions to invoke remote service operations. Input and output parameters of these functions are native objects that are transformed by the marshalling/unmarshalling code.

WS-ADDRESSING INTRODUCTION

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

- **Destination** (URI): mandatory address of the receiver of the message.
- **Action** (URI): mandatory unique identifier for the semantics of the message, e.g. its associated operation.
- **Reply endpoint** (EPR): reference of the receiver of replies.
- **Message id** (URI): a unique identifier for a message.
- **Relationship** (QName, URI): a pair of values which indicates how the message relates to another. The « Reply » predefined relationship type is used to refer in a reply to the message id property of the request.
- **Source endpoint** (EPR): Optional reference of the sender endpoint.
- **Fault endpoint** (EPR): Optional reference of the receiver for faults related to the message.

The first two properties are required and can be used by a Web Services server to dispatch the message to the appropriate service and operation handler.

The key abstraction underlying WS-Addressing is the Endpoint Reference (EPR), composed of an Address and optional Reference Parameters and Metadata description, which identifies a resource. The Address can be a logical or physical address of the service (a URI). The Reference Parameters are state information that the service uses to disambiguate resources (e.g. a session context), and are opaque to the caller. A caller obtains an EPR for the resource and uses the Address field for sending a message to the resource, by placing that Address in the corresponding SOAP header and adding the Reference Parameters as additional SOAP headers, allowing the service receiving that message to route it to the appropriate resource.

WS-Addressing also specifies an extension to the WSDL 1.1 language, as a new attribute that allows actions to be associated to operations input and output messages in portTypes definitions. This attribute is optional, and a simple syntactic rule can be

used by tools to generate a unique action URI for each message from the service target namespace, the portType name and the operation name.

WS-Addressing has been adopted as a W3C standard in May 2006. DPWS however references an earlier draft of the specification (from August 2004). This earlier draft is the one used by the DC toolkit.

WS-DISCOVERY INTRODUCTION

WS-Discovery [**WS-Discovery**] is based on SOAP 1.2 and builds on WS-Addressing: all discovery messages are SOAP 1.2 messages extended with WS-Addressing headers.

The WS-Discovery specification defines a multicast discovery protocol to search for and locate resources or, more specifically, Target Services available to network-connected clients. The primary mode of discovery is a client searching for one or more Target Services. The search can either specify the type of the Target Service or a scope in which the Target Service resides or both, and is materialised as a Probe message sent to a multicast group. Target Services that match the probe send a Probe Match response in unicast mode. Target Services can also be localized by name, through a similar protocol exchange involving a multicast Resolve message and a unicast Resolve Match response.

To minimize the need for polling, when a Target Service joins the network, it announces itself by sending a multicast Hello message. By listening to the multicast group, clients can detect newly-available Target Services without repeated probing. When leaving the network in an orderly manner, a Target Service announces this through a Bye message.

The WS-Discovery protocol messages are sent over UDP [**SOAP-over-UDP**] in order to minimise network traffic overhead.

Multicast-based discovery is limited to local subnets. In order for discovery to be scalable to enterprise-wide scenarios, WS-Discovery introduces the notion of discovery proxy (DP). A DP has two functions: suppressing multicast discovery (to reduce network traffic) and extending the network reach for the discovery protocol beyond the local subnet. When a DP detects a probe or resolution request sent by multicast, the DP sends a Hello for itself. By listening for these announcements, clients detect DPs and switch to use a DP-specific protocol. However, when a DP is unresponsive, clients revert to use the ordinary discovery protocol. While the definition of a DP-specific protocol is beyond the scope of WS-Discovery, it is expected that any such protocol would define search messages that clients send directly (in unicast mode) to the DP rather than to a multicast group; a DP could thus be deployed and automatically manage a network without any changes to the deployed services. The Discovery Proxy mechanism is an optional feature of WS-Discovery, currently not supported by the DPWS Core toolkit.

WS-Discovery metadata information is limited to the strict minimum, in order to reduce the size of the multicast UDP messages. Therefore, an additional protocol is used by DPWS to obtain a more complete description of a Target Service, once it has been discovered. This protocol is further detailed in the section introducing the DPWS device model.

WS-EVENTING INTRODUCTION

WS-Eventing [**WS-Eventing**] describes a protocol allowing one Web Service (called an "event sink") to register interest (called a "subscription") with another Web Service (called an "event source") in receiving messages about events (called "notifications"). To improve robustness, the subscription is leased by an event source to an event sink,

and the subscription expires over time. An event source may allow an event sink to renew the subscription. This publish-subscribe specification is not very complex, yet quite powerful; and is intended to enable implementation of a range of applications, from device-oriented eventing to enterprise-scale publish-subscribe systems, on top of the same substrate.

WS-Eventing is based on SOAP and builds on the WS-Addressing standard. By re-using WS-Addressing on top of SOAP, together with its underlying resource model, WS-Eventing does not need to define any new protocol. It merely specifies the WSDL definitions associated with the subscription management operations: Subscribe, Unsubscribe, Renew (used by an event sink) and SubscriptionEnd (used by an event source when terminating its event notification service).

Event notification messages themselves are one-way messages, the content of which is not constrained by WS-Eventing: the service WSDL document describes both the event messages, which may include any data of any type, and the events, which are declared in the service portType as output-only operations referencing the event messages. In theory, output-input operations (also called Solicit-Response operations) could also be defined, but they are not currently supported in the DC toolkit.

An event source may support filtering to limit the amount of notifications sent to the event sink. If it does and a subscribe request contains a filter, the event source sends only notifications that match the requested filter.

THE DPWS DEVICE AND HOSTED SERVICES MODEL

The DPWS specification references all the specifications described in previous sections and profiles them for use in embedded devices.

DPWS profiles WS-Discovery by introducing a distinguished type of service, called a "device", which must be a compliant Target Service. Devices are designed to host and advertize other services.

A device exposes the standard WS-Discovery information:

- Endpoint reference: a WS-Addressing EPR that should feature a stable URI as its address field. The DPWS specification recommends the use of a UUID.
- Types: a set of qualified names (QNames). Each type identifies a set of messages that the device can receive or send. DPWS introduces a predefined dpws:Device type that identifies DPWS-compliant devices. Additional application-defined types may be added to this set: those types may be abstract, i.e. represent a functionality provided by their hosted services (e.g. a MultiFunctionPrinter device that exposes both a Print service and a Scan service), or concrete, e.g. a WSDL portType, in which case the functionality is directly offered by the device endpoint. The latter use is not recommended, as it does not follow the device and hosted services model proposed by DPWS.
- Scopes: a set of URIs representing application-specific logical grouping. Scopes can be used for instance to represent a geographical location, a position in a network topology, management information…
- Transport addresses: the set of URLs which can be used to reach the device.

A device acts as a metadata resource for both itself and its hosted services. By including the predefined `dpws:Device` type in its types, a device indicates that its endpoints (defined by the device transport addresses) support the WS-Transfer Get operation [**WS-Transfer**] to allow clients to retrieve its metadata.

The device metadata is returned as a WS-MetadataExchange [**WS-MetadataExchange**] document, and contains the following information:

- Device model: this includes manufacturer name and URL, model name, number and URL, and the device presentation URL (i.e. device home page).
- Device instance: this includes the device "friendly name", its serial number and its firmware version.
- Hosted services: this includes the list of hosted services. The description of each hosted service contains a set of WS-Addressing EPRs which can be used to contact the service, the set of types (normally WSDL portTypes) implemented by the service, and a URI representing a service id.
- WSDL documents: this includes the WSDL documents describing the Web Services that are directly offered by the device endpoints.

Hosted services can also be queried for their metadata, using the same WS-Transfer Get operation on the hosted service endpoints. Service metadata includes:

- Relationship to the hosting device: this includes a description of the device endpoint reference and types, and of the hosted service endpoint references, types and service id.
- WSDL documents for the hosted service.

DEVELOPMENT PROCESS USE CASES

This section introduces the main development scenarios in which the DPWS Core toolkit can be used. The development process for each scenario is then further detailed in a separate chapter of this user guide.

SERVICES AND DEVICE DEVELOPMENT

This scenario corresponds to the development of a stand-alone device that exposes its functionalities through hosted services. This requires the definition and implementation of the device hosted services, which follow the usual Web Services development pattern: identification of the Web Services interfaces, description of the Web Services using the WSDL language, implementation of the services and setup of the server message dispatch mechanisms. It also involves the configuration of the device with metadata information required for the discovery and plug-and-play mechanisms, as well as the use of the specific mechanisms offered by the DC toolkit for event publishing.

PURE CLIENT DEVELOPMENT

This scenario corresponds to the development of a simple client that discovers devices and consumes Web Services. Pure clients running on workstations and enterprise servers are not the primary target of the DC toolkit, as other Web Services technologies based on the Java or .NET platforms are available and often preferred to C stacks in such environments. However, the DC toolkit might still be useful for the development of simple test clients on development platforms, as well as in cases where network discovery and plug-and-play are required, as WS-Discovery is not yet supported in all mainstream products.

PEER-TO-PEER CLIENT DEVELOPMENT

This scenario corresponds to the development of a device that needs to consume other Web Services (provided by either another device or an external Web Services application) to provide its business functionality. In such cases, the device will need to use both the server part and client part of the DC toolkit: business operations exposed

as Web Services will use in their implementations the discovery and service invocation capabilities offered by the DC client stack.

This development scenario is useful when designing and implementing a full service-oriented architecture for devices, in which devices interoperate in a peer-to-peer manner. It is also useful in hierarchical architectures, in which higher-level devices orchestrate the behavior of lower-level ones.

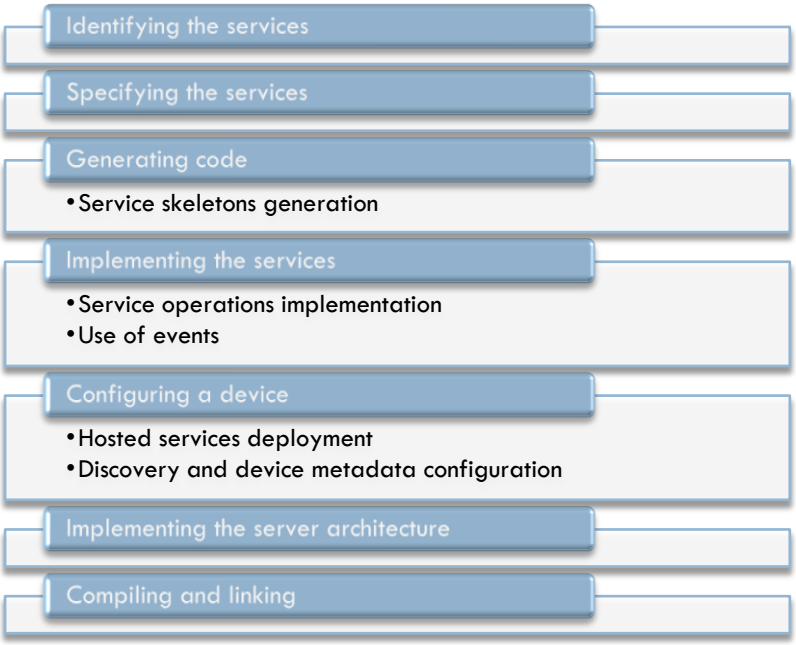
ASYNCHRONOUS AND EVENTING CLIENT DEVELOPMENT

This scenario corresponds to the development of a device (acting as a client) or a pure client that needs to receive asynchronous messages from service providers. This can occur in two cases: (i) when using the explicit reply address feature of WS-Addressing, or (ii) when subscribing to event sources compliant with WS-Eventing. In both cases, the client will need to use part of the DC server stack, in order to listen to incoming asynchronous messages, receive them and dispatch them to the appropriate message handlers. When the client is a device, the server stack is already present and only needs to be configured with the asynchronous message handlers. When the client is a pure client, its architecture must be extended with the server message dispatch mechanisms.

SERVICES AND DEVICE DEVELOPMENT

DEVELOPMENT PROCESS OVERVIEW

The following figure highlights the main steps in the development of a device and its hosted services using the DC toolkit.



The first four steps are required in the development of any Web Services applications, and, except for the use of events, do not feature any aspects specific to device development.

The fifth step on the other hand is specific to a DPWS device development, as it consists in configuring the device with appropriate information to allow it to participate in the discovery exchanges.

The last two steps are also standard steps in Web Services application development.

The above steps are further detailed in the following sections.

SERVICE IDENTIFICATION GUIDELINES

The first step in the development of a device using the DC toolkit is the identification of the hosted Web Services that will be deployed on the device. Examples of Web Services include:

- *Control services*: these services are used to expose the primary functionality of a device. Typical examples of control operations exposed by control services include for instance a `switchOn(boolean)` operation for a lighting device or a `startCycle(temperature, duration)` operation for a washing machine.
- *Management services*: these services are used to expose the configuration, monitoring and diagnosis capabilities of the device.

Although the above distinction may be useful at design time, all Web Services are handled in the same way by the DC toolkit.

Service identification may be easier when the device under development must comply with existing standards: in such a case, services and their interfaces may be already

defined, and the first two steps of the development process may be ignored. This is the case for instance when using standard management services such as the ones defined by WS-Management, or when implementing control services for a standard device in a vertical domain (e.g a Print service for a standard DPWS printer).

However, in many cases, the device developer must identify the services to be exposed by the device. The following guidelines may be used to help identify and define these services:

- Identify stable interfaces: one of the key features of service-oriented design is the strong separation between interfaces and implementations. This means that the implementation of a service can easily be modified while preserving the same interface. In order to leverage this benefit, it is important to design the service interface in a way that guarantees its stability over time. Stability can be achieved by considering a set of use cases large enough to encompass the anticipated use of the service over its complete life cycle.
- Define coarse-grained operations: although the granularity of service operations is directly dependent on the specific service functionality and must be defined on a case-by-case basis, it is generally recognized that defining coarse-grained operations rather than fined-grained ones is amenable to more flexible architectures. Coarse-grained operations hide more details of an implementation: compare for instance a **`startCycle`** operation with a set of operations that would control each individual washer variable involved in the washing cycle operation. They also reduce the need for service consumers to maintain server state when executing a sequence of fine-grained operations: the state is instead maintained by the server while executing the larger operation.
- Promote loose coupling: reducing the coupling between service providers and service consumers is another fundamental goal of a well-designed service-oriented architecture. It allows services to be reused by new clients that were not taken into consideration at service design time, and clients to continue working with new versions of services. Stable interfaces and coarse-grained operations are two properties that contribute to loose coupling, as are other properties such as platform independence (e.g. avoiding the use of platform-specific types in XML serialization) and the use of asynchronous communications.

SERVICE INTERFACE SPECIFICATION IN WSDL

Once the services are identified, the second step requires the formal specification of these services using WSDL and XML Schemas documents. This involves:

- The definition of the service portTypes, operations and message ;
- The definition of the types used in messages, as XML Schema types and elements ;
- The definition of the service bindings.

Note that the above list is ordered by the logical steps required in the design of the service specification, not by the order in which corresponding XML elements appear in the WSDL document.

PORTTYPES, OPERATIONS AND MESSAGES DEFINITION

The service interface is represented by its portTypes. A **`wsdl:portType`** element has a '**`name`**' attribute and contains a set of **`wsdl:operation`** children. Each **`wsdl:operation`** element has a 'name' attribute, and contains a **`wsdl:input`** and/or a **`wsdl:output`** child representing messages used by the operation.

The following example is an extract from a complete WSDL document that focuses on the portType declaration for a washing machine service. Complete WSDL examples can be found in the samples directory of the DC toolkit package:

```
<wsdl:portType name="Wash" wse:EventSource="true">
  <wsdl:documentation>
    This port type defines operations for launching washing cycles.
  </wsdl:documentation>
  <wsdl:operation name="LaunchCycle">
    <wsdl:documentation>
      Start a washing cycle.
    </wsdl:documentation>
    <wsdl:input message="tns:LaunchCycleMsg" />
  </wsdl:operation>
  <wsdl:operation name="GetCycleStatus">
    <wsdl:documentation>
      Returns progress information on the running cycle.
    </wsdl:documentation>
    <wsdl:input message="tns:GetCycleStatusReqMsg"
      wsa:Action="http://www.soa4d.org/WashingMachine/GetCycleStatus"/>
    <wsdl:output message="tns:GetCycleStatusRespMsg"
      wsa:Action="http://www.soa4d.org/WashingMachine/CycleStatus"/>
  </wsdl:operation>
  <wsdl:operation name="CycleEnded">
    <wsdl:documentation>
      Event sent when a cycle has ended.
    </wsdl:documentation>
    <wsdl:output message="tns:CycleEndMsg" />
  </wsdl:operation>
</wsdl:portType>
```

This example shows the three types of operations that are supported by the DC toolkit:

- The **`LaunchCycle`** operation is a one-way operation that starts a washing cycle. It has only a **`wsdl:input`** message element.
- The **`GetCycleStatus`** operation is a request/response operation that returns the current cycle state. It has a **`wsdl:input`** message element followed by a **`wsdl:output`** message element.
- The **`CycleEnded`** operation is a notification operation that represents an event. It has only a **`wsdl:output`** message element. The output-only operation is interpreted as an event because the '**`wse:EventSource`**' attribute is set to true in the **`wsdl:portType`** element.

This example also shows the use of the optional '**`wsa:Action`**' attribute for associating action URIs to both input and output messages. When using WS-Addressing, an action URI must be included in the **`wsa:Action`** header block for all SOAP messages. Therefore, when the '**`wsa:Action`**' attribute is not explicitly specified in the WSDL document, a default action URI for a message is generated from the target namespace, the portType name and the input or output message name:

default action URI = [target namespace]/[portType name]/[input | output name]

When the input or output message does not have an explicit name attribute (as is the case in the example above), the message name is obtained from the operation name:

- For operations using a single message, the message name is the operation name.
- For request/response operations, the input (resp. output) message name is the operation name with "Request" (resp. "Response") appended.

All **`wsdl:input`** and **`wsdl:output`** elements used in operation definitions must have a '**`message`**' attribute that contains a reference to a message definition. Message

definitions describe the abstract structure of the messages received and sent by operations. When using a document/literal SOAP binding, as recommended by the DPWS specification, the role of message definitions is simply to reference XML Schema elements, as shown in the example below:

```
<wsdl:message name="LaunchCycleMsg">
  <wsdl:part name="body" element="tns:LaunchCycle" />
</wsdl:message>
<wsdl:message name="GetCycleStatusReqMsg" />
<wsdl:message name="GetCycleStatusRespMsg">
  <wsdl:part name="body" element="tns:CycleStatus" />
</wsdl:message>
<wsdl:message name="CycleEndMsg">
  <wsdl:part name="body" element="tns:CycleEnd" />
</wsdl:message>
```

Three of the four messages have a single part that references a global XML Schema element through its element attribute. When using the document/literal SOAP binding style, this means that the **soap:Body** of the corresponding SOAP message will contain the referenced XML Schema element as its only child. The **GetCycleStatusReqMsg** message has no parts, as it corresponds to a SOAP message with an empty soap:Body element.

TYPES DEFINITION

While the portTypes, operations and messages describe the set of messages that a service will receive and send, types are used to precisely define the structure of each message. Types are referenced as QNames in message definitions, and may be either:

- Global XML Schema element declarations: these declarations should be used in messages when document/literal SOAP bindings are used.
- Global XML Schema type definitions: these definitions may only be used in messages when rpc/literal SOAP bindings are used.

The XML Schema definitions can appear either inline inside the WSDL document, or in a separate XML Schema document, which must be imported using the **xsd:import** element. In both cases, a **xsd:schema** element must appear as a child element of the **wsdl:types** element.

The following snippet shows a global element declaration of the **LaunchCycle** element, which is used by the message definitions in the previous example:

```
<wsdl:types>
  <xsd:schema
    targetNamespace="http://www.soa4d.org/DPWS/Samples/WashingMachine"
    xmlns:tns="http://www.soa4d.org/DPWS/Samples/WashingMachine">
  …
    <xsd:element name="LaunchCycle">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="Temperature"
                       type="tns:WaterTemperature" />
          <xsd:element name="SpinDryingSpeed" type="tns:SpinSpeed" />
        </xsd:sequence>
        <xsd:attribute name="Name" type="tns:Cycle" use="required" />
      </xsd:complexType>
    </xsd:element>
  …
  </xsd:schema>
</wsdl:types>
```

BINDINGS DEFINITION

In order to complete a service definition, one or several bindings must be defined for each service portType. A binding provides protocol and message information that

completely defines the concrete messages exchanged between a service consumer and a service provider. The DPWS specification requires each service to expose at least a binding for SOAP 1.2 over HTTP, using the document/literal style.

The following example shows such a binding definition for the Wash portType previously introduced:

```
<wsdl:binding name="Wash" type="tns:Wash">
  <wsoap12:binding style="document"
                   transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="LaunchCycle">
    <wsdl:input>
      <wsoap12:body use="literal" />
    </wsdl:input>
  </wsdl:operation>
  <wsdl:operation name="GetCycleStatus">
    <wsdl:input>
      <wsoap12:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <wsoap12:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="CycleEnded">
    <wsdl:output>
      <wsoap12:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

Noteworthy aspects of this binding definition include:

- Use of the "http://schemas.xmlsoap.org/wsdl/soap12/" namespace URI and associated prefix (wsoap12) to identify the SOAP 1.2 binding.
- Specification of the HTTP transport.
- Use of the 'document' style for the complete binding and the 'literal' encoding for each message body.
- No use of the SOAPAction attribute in operation elements, as action URIs are associated to input and output messages in the portType definition, using the WS-Addressing extension.

It can also be noted that, because DPWS constrains the type of bindings that must be supported by hosted services, there is little added value in the binding definition presented above: all the information can be derived from previous messages and portTypes definitions. However, in order to perform correctly, the DC toolkit code generation tools require the binding to be completely defined.

Advanced use:
This section has given an example of WSDL definitions using the SOAP 1.2 over HTTP binding, in compliance with the DPWS requirements. The DC toolkit also supports the use of alternative WSDL authoring styles, including RPC mode and WS-I Basic Profile 1.1.

Code generation

A service implementation relies on code generated from the service WSDL document for the translation of incoming requests into service function invocations, and for the translation of the functions results into response messages.

The DC toolkit provides two code generation tools derived from those available in the gSOAP open-source product. These tools have been modified to support DPWS extensions such as WS-Addressing and WS-Eventing:

- **wsdl2h**: this tool can be used to translate WSDL and XML Schema files into annotated header files containing gSOAP-specific annotations.
- **soapcpp2**: this tool takes as input a gSOAP annotated header file and generates several C files, containing the skeletons, the stubs and the marshalling code.

WSDL2H

The **wsdl2h** tool takes a single WSDL file (or URL) as input and translates it into a single annotated header file, including in the same output file information gathered from imported WSDL and XML schema documents. A typical invocation of the command-line tool is shown below:

```
wsdl2h -c -o example.gsoap -t typemap.dat example.wsdl
```

In the above example:

- The **-c** option is used for generating annotated header files for pure C code (and not C++).
- The **-o** option specifies the name of the output file. In the DC toolkit samples, the convention is to use the **.gsoap** extension to identify gSOAP annotated header files. gSOAP normally uses a **.h** extension for these files.
- The **-t** option specifies the name of an optional type mapping file, often called **typemap.dat**. When this option is not explicitly specified, the tool looks for a file with that name in the current directory. If none is present, default values are used.

One role of the type mapping file is to declare namespace prefixes to be used in generated structures and functions names. By default, the tool generates ns1, ns2… prefixes, but this can be overridden to use more meaningful prefixes.

The generated annotated header file contains:

- Type and structure definitions generated from XML Schema (XSD) types: each simple type definition becomes a reference to a native C type or enumeration; each complex type definition becomes a C struct in the generated file. The name of the type is derived from the XSD type name: an XSD type called **ns:type** (where **ns** is the prefix for the XML Schema target namespace) becomes a C type called **ns__type** (note the double underscore).
- Operation declarations generated from WSDL binding declarations: each operation appearing in a binding becomes a function prototype. The name of the prototype is derived from the operation name and the prefix of the WSDL target namespace. An operation called **oper** becomes a function called **ns__oper** when using RPC style, and **__ns__oper** when using document style (where **ns** is the prefix for the WSDL target namespace).

Message parts in the WSDL documents are mapped to parameters in the function prototypes:

- Document/literal operations can have one or two parameters: the last parameter represents the response message, which must be of type **void** in case of one-way operations or of type pointer to a **struct** representing the type of the message part element in case of request/response operations. The parameter representing the request must be omitted when the request uses an empty message, or be of type pointer to a **struct** representing the type of the message part element otherwise.

- RPC/literal operations follow the same rules, except that each input message part is individually mapped on a single parameter using a type corresponding to the message part type.

The following table summarizes the main mapping between XML/WSDL entities and C entities appearing in the annotated header file. Note that these entities are not directly used in C code: a second code generation step is required to produce code that can be compiled.

XML/WSDL entity	XML QName	C entity and name	Comment
XSD simple type	ns:stype	typedef: ns__stype enum: ns__stype	
XSD complex type	ns:ctype inside ns:elem	struct: ns__ctype struct: _ns__elem	Global type Internal type
XSD element	ns:elem	struct: ns__elem	Normally not used
doc/lit operation	ns:oper	int __ns__oper(in, void) int __ns__oper(in, out) int __ns__oper(out)	One-way or notification Request/response
rpc/lit operation	ns:oper	int ns__oper(p1, …, pn, void) int ns__oper(p1, …, pn, out) int ns__oper(out)	One-way or notification Request/response

Advanced use:
The type mapping file can also define a mapping of XML Schema types to specific C types, instead of using the default mapping. Such a mapping is required for instance when using MTOM encoding for transporting binary data. See "Mapping XML Schema types to specific C types" for more details.

Warning: the **wsdl2h** tool uses the GNU Public License (GPL), which means that code generated with this tool cannot be included in software not distributed as open source under the same license.

SOAPCPP2

The second phase of code generation uses the **soapcpp2** stub and skeleton code generator. The tool takes as input a gSOAP annotated header file, and produces several C header and source files. A typical invocation of the command-line tool is shown below:

```
soapcpp2 -2ucn -pws -d gen example.gsoap
```

In the above example:

- The **-2** option specifies that the generated code should use SOAP 1.2 (and not SOAP 1.1).
- The **-u** option specifies that DPWS extensions should be used.
- The **-n** and **-p** options enable the support for multiple services in the stack. The prefix specified after the **-p** option (**ws** in this example) is used by the generator as a prefix for the name of generated files. When not specified, the default prefix used by the tool is **soap**.
- The **-d** option specifies the output directory for generated files.

Several C header and source files are generated, using the specified prefix (`ws` in this example):

- *ws*C.c *and wsH.h* contain marshalling/demarshalling code;
- *wsStub.h* contains stub & skeleton declarations (derived from the information of the annotated header file), including all required type declarations and function prototypes.
- *ws.nsmap* contains prefix and namespace definitions for the web service.
- *wsServer*.c contains the skeleton code that handles SOAP and DPWS processing and calls the user service implementation. It also contains stubs for sending event notifications.
- *wsClient*.c contains the stub code.
- *wsHandler*.c contains the code for handling asynchronous messages received on the client side that may have been initiated by a server replying to a specific endpoint or sending event notifications. It is a kind of skeleton specific to DPWS.
- *wsServerLib*.c, *wsClientLib*.c and *wsHandlerLib*.c are versions of stubs & skeletons that include a 'static' version of the marshalling/demarshalling code. They are useful when deploying multiple web services in a single program.

When implementing a service on the server side, only the following files are required: *ws*C.c, *wsH.h*, *wsStub.h*, *ws.nsmap*, *wsServer*.c and *wsServerLib*.c.

Advanced use:
Instead of using a WSDL document as the input to the code generation procedure, it is also possible to directly edit and use the gSOAP annotated header file as input. This approach provides greater flexibility to control the code generation. See "Editing the gSOAP annotated header file" for more details.

SERVICE IMPLEMENTATION

Once the service skeleton is generated, it is necessary to implement the service functions that are called by the skeleton. The reunion of the generated code and the service functions represent the complete service implementation, also called a *service class* in the following sections.

One specific aspect of services developed with the DC toolkit is their support for events, as detailed in this section.

IMPLEMENTATION OF SERVICE FUNCTIONS

The generated code on the server side relies on service functions that must be implemented by the service developer for the generated skeletons to work. Each operation declared in a gSOAP annotated header file is associated with a generated skeleton function that invokes a service function. The name and parameters of the service function are derived from the operation declaration:

- The name of the function is the name of the operation (including the double underscore in case of doc/literal operations).
- The first parameter is a pointer to a `struct dpws`, passed to the service function by the DC runtime environment to provide access to the execution context of the current Web Service request.
- The remaining parameters are the same as those declared in the annotated header file, except for `void` parameters that are dropped.

The following is an example of a one-way service function declaration invoked by a generated skeleton:

```
int __wsh__LaunchCycle(
      struct dpws*,
      struct _wsh__LaunchCycle *wsh__LaunchCycle);
```

The implementation of service functions is completely application-dependent, but a few rules and guidelines apply:

- *Return value*: the service function must return an **`int`** value. This value must be **`DPWS_OK`** (i.e. 0) when the function returns normally. Because one-way operations do not return faults, their service functions should usually return **`DPWS_OK`**.
- *Fault*: when a processing error occurs in the service function implementing a request/response operation, the service should return a SOAP fault to the sender. This is notified to the DC runtime by calling the **`dpws_fault`** function and returning its return value as error code from the service function. Note that the **`dpws_fault`** function, like many other DC runtime functions, takes the **`struct dpws`** pointer as first parameter.
- *Response message allocation*: in case of request/response operations, the service function is responsible for constructing the response object that will be marshalled into the SOAP response message by the DC runtime. When this object is complex, dynamic allocation may be required. Because the allocated data must be available to the generated skeleton after the service function returns, and because control is not explicitly returned to application code after the service function returns, the DC runtime provides a specific memory allocator function, **`dpws_malloc`**. Memory allocated by this function is managed by the DC runtime, and is freed after each request is processed using the **`dpws_end`** function (the use of the **`dpws_end`** function is further discussed in the server architecture section).
- *Access to application data*: when developing products featuring multiple devices and services, it is often required to associate specific application data to a device or a service. This association is performed at deployment and configuration time (see relevant section for details). The DC runtime provides functions, called **`dpws_get_device_user_data`**, **`dpws_get_service_class_user_data`** and **`dpws_get_endpoint_user_data`** respectively, to retrieve within a service function the application data associated to the device, service class or service to which the current request has been directed.

Advanced use:
In addition to the use of generated skeletons, the DC toolkit also provides support for generic message processing functions. Instead of using generated C structs to handle the XML content of the SOAP messages, these functions use a streaming XML API to access to this content. See "Generic invocation" for details.

USE OF EVENTS

One of the specific aspects of DPWS development is the use of events to support asynchronous notifications from a service to one or several subscribers. Event notifications are one-way messages sent by an event source to registered event sinks. Unlike standard one-way message interactions, event notifications do not require the message sender to know in advance the message receiver endpoint and its WSDL description. Rather, it is the event source (message sender) that publishes the event description in its WSDL document, and let interested receivers subscribe to the event. This kind of publish/subscribe architecture promotes more dynamic and flexible links between message senders and receivers.

In order to act as an event source, a service implementation must:

- Expose a **subscribe** operation on its endpoints, to allow subscribers to declare their interest: this is entirely taken care of by the DC runtime and the generated skeletons for a service marked as event source.
- Generate event notifications: this requires the invocation of generated notification functions.

Advanced use:
The DC toolkit defines configuration properties for managing subscriptions and their life cycle on the server side. See "Advanced eventing features" for details.

GENERATED NOTIFICATION FUNCTIONS

In addition to the skeleton functions used to dispatch incoming messages for a service, the generated code for the server side (included in the generated *wsServer*.c file) also contains stubs that can be used to send event notifications to subscribers. For each notification (output-only) operation declared in the service annotated header file, a corresponding **dpws_notify_<operation name>** function is generated (the operation name is of the form **__ns__oper** for document/literal operations). The generated function has the following parameters:

- The first parameter (*dpws*) is a pointer to a **struct dpws** that must be initialized by the user code.
- The second parameter (*event_source*) is a reference to the service object that internally represents the event source. This handle reference must be obtained at service deployment and configuration time, as described in the deployment and configuration section.
- The remaining parameters are the same as those declared in the annotated header file, except for the final **void** parameter that is dropped. They represent the body of the message and will be serialized into XML using the marshalling code.

The following is an example of a generated event notification stub declaration:

```
int dpws_notify___wsh__CycleEnded(
      struct dpws*,
      short,
      struct _wsh__CycleEnd *);
```

SENDING EVENT NOTIFICATIONS

The invocation of the event notification stubs is similar to the invocation of one-way operation stubs on the client side, as described in the client development section. The basic steps required to perform such an invocation involve:

- Creating and initializing a request context, i.e. a **struct dpws** object: this object can be allocated statically, on the stack or on the heap. Once the object has been allocated, it must be initialized using the **dpws_client_init** function.
- Invoking the notification function: this is performed as a standard function call, passing a pointer to the initialized request context as first parameter.
- Cleaning up the request context: this is performed by calling the **dpws_end** function with a pointer to the request context as parameter.

Generated stub invocation guidelines and usual pitfalls also apply to event notification. They are further detailed in the client development section.

The following code snippet shows an example of event notification:

```
…
int status;
```

```
short hEventSource;
struct dpws dpws;
struct  wsh  CycleEnd cycleEnd;
…
dpws client init(&dpws, NULL);
cycleEnd.CycleName = "GENTLE";
…
// send an event to subscribers using the client dpws structure
status = dpws notify   wsh  CycleEnded(&dpws, hEventSource, &cycleEnd);
if (status)
       fprintf(stderr, "Could not send 'cycle end' event\n");
dpws_end(&dpws);      // free memory allocated for event message
…
```

Advanced use:
An event notification stub sends a separate event notification message to all active subscribers, thus often sending more than one message during one invocation. This makes network error handling more complex, as more than one message delivery may fail. The DC toolkit provides a way to register a callback function to receive individual notification of message delivery failures. See “Advanced eventing features” for details.

SERVICE DEPLOYMENT AND DEVICE CONFIGURATION

Once services are implemented, the next step requires these services to be deployed and attached to devices, and devices to be configured with appropriate metadata to support the discovery and metadata exchange protocols.

The DC toolkit uses an internal structure, called the registry, to hold all the required information about devices, service implementations (called service classes) and hosted service endpoints. This section describes this registry structure, the object model on which it relies to organize the information, the API functions that are provided to create and configure registry objects, and additional bookkeeping information that must be managed to allow the DC runtime environment to operate properly.

ROLE OF THE REGISTRY

The role of the registry is to maintain on the server side metadata information about devices and services supporting:

- The dispatch of incoming SOAP messages to the targeted service or device, based on the endpoint URL.
- The advertisement of devices and the retrieval of matching devices in accordance with the WS-Discovery specification.
- The retrieval of device and service metadata in accordance with the DPWS specification.

The DC registry also provides features that go beyond the basic DPWS requirements:

- The registry can manage multiple devices of different types, each device hosting multiple services. This feature is useful for instance when developing DPWS gateways that expose several field bus devices as individual DPWS devices.
- The registry can be dynamically modified while devices and services are running. This feature is useful to support the dynamic deployment of new devices or hosted services.

THE REGISTRY OBJECT MODEL

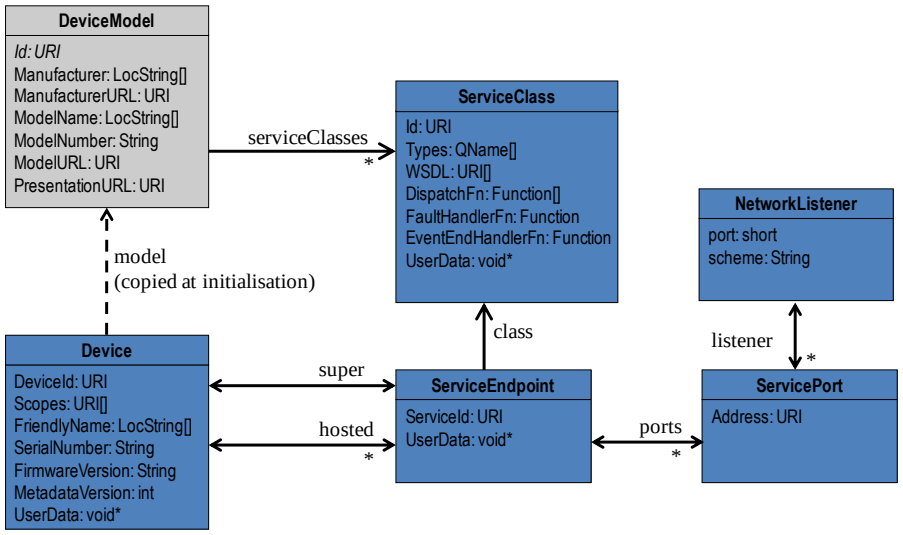
The registry relies on an internal object model to structure all the required information about devices and services. This object model is exposed to the developer through the DC API. It is organized around four main classes of runtime objects:

- *ServiceClass*: objects of this class represent service implementations and their associated metadata.
- *ServiceEndpoint*: objects of this class represent service instances. A service endpoint is always associated to a service class.
- *Device*: objects of this class represent device instances. A device is a kind of service endpoint (hence the *super* link), and hosts a set of hosted services, which are also represented as service endpoints.
- *ServicePort*: objects of this class represent addressable network endpoints (URLs), to which SOAP messages can be sent. Service ports are associated to one service endpoint.

In addition to those four classes, the registry also defines:

- *DeviceModel*: objects of this class act as prototypes for instantiating devices. Their content is copied in the device instance at instance creation time. Device models also reference a set of service classes.
- *NetworkListener*: objects of this class represent server endpoints listening for incoming connection requests. This is a singleton object in the current version, only supporting the http scheme.

The following class diagram shows the main classes and associations in the registry object model:



ServiceClass objects have the following attributes:

- *Id*: a URI that identifies the service class.
- *Types*: a set of QNames that represent the types implemented by this *ServiceClass*. Types are usually the names of the portTypes defined in the service WSDL document.
- *WSDL*: a set of URI referring to WSDL documents describing this service class. Although a single WSDL document is normally enough to describe a service class, the DC toolkit allows the use of several documents, to support the cases where more than one Web Service is exposed on a single network endpoint.

- *DispatchFn*: a set of callback functions that will be called by the DC runtime when dispatching a message sent to a service which instantiates this *ServiceClass*. The usual case is to use the dispatch function generated by the skeleton generator from the service class WSDL document. User-defined functions using a generic skeleton may also be used.
- *UserData*: a pointer to a user-defined object that can be used to associate application-specific information to the *ServiceClass*. This data can be retrieved at execution time in the service functions.

ServiceClass objects have two additional callback function properties, *FaultHandlerFn* and *EventEndHandlerFn*. These properties are used on the client side when implementing asynchronous and eventing clients. Their use is described in more details in the relevant chapter.

ServiceEndpoint objects are associated to a *ServiceClass* describing their implementation, and to a hosting *Device*. They have the following attributes:

- *ServiceId*: a URI that uniquely identifies the service instance within its hosting device. By default, it uses the *Id* attribute of its *ServiceClass*.
- *UserData*: a pointer to a user-defined object that can be used to associate application-specific information to the *ServiceEndpoint*. This data can be retrieved at execution time in the service functions.

ServicePort objects are associated to a *ServiceEndpoint* and have the following attribute:

- *Address*: a URL that represents the network address on which the service endpoint associated to the service port is available. This URL is usually relative, as the corresponding absolute URL is built from the server IP address and network listener scheme (currently limited to http) and port number. When not specified, a generated UUID is used as relative URL.

Device objects are more complex, as a device is a kind of service, and has an implicit service class and service port associated to it. Therefore, most of the attributes described above are applicable to device objects, with the following restrictions and specificities:

- *DispatchFn* and *WSDL*: in the DPWS model, devices are normally only used as a metadata resource for supporting the discovery of their hosted services. However, in some cases, it might be necessary to associate some functional behavior to the device endpoint. In such cases, the *DispatchFn* and *WSDL* attributes may be set on a device as they would be on a service class. In all the other cases, they should not be set.
- *Types*: a set of types representing the device functional capabilities. In the DPWS model, device types represent the overall functional behavior, and should not reference specific service portTypes. The exception to this rule is when a specific service is deployed on the device endpoint (as described above). In such a case, the service portTypes should be included in the device type set.
- *ServiceId*: this attribute cannot be set on a device.
- *Address*: this attribute represents the device transport address. This URL is usually relative, as the corresponding absolute URL is built from the server IP address and network listener scheme (currently limited to http) and port number. When not specified, the device id (a UUID) is used as relative URL.

Device objects also have their own specific attributes. They can be split into two groups: model attributes and instance attributes.

Device model attributes include:

- *Manufacturer*: a set of localized strings representing the device manufacturer name in different languages. Localized strings (`struct localized_string` in the API) associate a string with a language, represented by a standard language code (e.g. `en`, `fr`…), as used by the standard `xml:lang` attribute.
- *ManufacturerURL*: the URL of the manufacturer Web site.
- *ModelName*: a set of localized strings representing the device model name in different languages.
- *ModelNumber*: a manufacturer model reference for the device.
- *ModelURL*: the URL of the model Web page.
- *PresentationURL*: the URL of the device home page. Although it belongs to the model attributes (in accordance with the DPWS specification), this URL is in fact specific to the device instance. The usual practice is to use a relative URL for this attribute, the absolute URL being then constructed from the device transport address and the relative URL.

Those attributes can either be set directly on a device, or be set on a DeviceModel object, in which case they will be automatically copied into each new device created from the model.

Device instance attributes include:

- *DeviceId*: a URI that must be stable over time, over reboots and across network interfaces. The usual practice is to use a UUID for this attribute. When not specified, the UUID is generated from one of the platform network interface MAC addresses and additional information specified at device creation time.
- *Scopes*: a set of URIs representing application-specific logical grouping. Scopes can be used for instance to represent a geographical location, a position in a network topology, management information…
- *FriendlyName*: a set of localized strings representing the device friendly name in different languages. Friendly names can be used for instance as display names in network explorers.
- *SerialNumber*: a string representing the device serial number.
- *FirmwareVersion*: a string representing the device firmware version.
- *MetadataVersion*: an unsigned integer representing the current version of the device metadata. This number should be incremented each time the device metadata information is modified. Further details on the management of this attribute are given in a following section.
- *UserData*: a pointer to a user-defined object that can be used to associate application-specific information to the device. This data can be retrieved at execution time in the service functions.

DEVICES AND HOSTED SERVICES CONFIGURATION

The configuration of the devices and hosted services in the registry is done through the DC configuration API which includes:

- Creation functions for all registry objects.
- Attribute setters and getters for registry objects.
- Device enabling and disabling functions.

All registry objects are exposed to the developer through handle references, represented as **short** integers, which are returned by the creation functions. This approach allows the DC runtime to internally maintain reference counters on all registry objects, thus supporting dynamic changes to the registry while running. The API provides a few functions for handle management, including a **dpws_release_handle** function that should be called on handle references when not used anymore by the application code.

The DC API provides the following generic attribute setters and getters, which always take a handle reference as first parameter and an attribute key as second parameter:

- **dpws_set_ptr_att** and **dpws_set_int_att**: sets a pointer (resp. int) value in a mono-valued attribute.
- **dpws_add_ptr_att**: adds a pointer value to a multi-valued attribute. There is no way to remove or clear a multi-valued attribute, but it is possible to use the **dpws_set_ptr_att** function to reinitialize the attribute with a single value.
- **dpws_get_ptr_att** and **dpws_get_int_att**: gets a pointer (resp. int) value from a mono-valued attribute.
- **dpws_get_att_count and dpws_get_ptr_att_item**: these two functions can be used to retrieve the number of entries in a multi-valued attribute and retrieve individual values.

The setter functions always make a copy of the value to be set. They are also available as macros, using the same name in uppercase. Setter and getter functions may return an error code if the handle reference is invalid, if the specified attribute key does not correspond to a valid attribute for the specified object, or if the specified attribute value is invalid.

The usual creation order for registry objects is the following:

- Creation of service classes: this is performed by calling the **dpws_create_service_class** function. Service classes must be configured after creation using the attribute setters, as shown in the code snippet below:

```
hServClass = dpws create service class();
// Configure the service class attributes.
DPWS ADD PTR ATT(hServClass, DPWS PTR PREFIXED TYPE,
                 &SwitchPowerPortType);
DPWS_ADD_PTR_ATT(hServClass, DPWS_PTR_WSDL, &LightingWsdl);
DPWS_ADD_PTR_ATT(hServClass, DPWS_PTR_HANDLING_FUNCTION,
                 &lit serve request);
DPWS SET STR ATT(hServClass, DPWS STR ID,
                 "http://www.soa4d.org/DPWS/Samples/Lights/Light1");
```

- Creation of device models: this is performed by calling the **dpws_create_device_model** function. The use of device models is optional, and mainly useful when developing gateways that feature several devices of the same model. Once a device model is created, model attributes can be configured, and specific service classes can be registered in the model, using the **dpws_register_service_class** function.
- Creation of devices: this is performed by calling the **dpws_create_device** or **dpws_create_custom_device** function. The first function takes a device model reference as parameter and will automatically generate one hosted service and one associated service port for each service class registered with the model. The second function can work without a model, and will not generate default hosted services. Both functions take as first parameter a numeric identifier, which must be unique and which is used as input for the device UUID generation. The following

code snippet shows an example of device creation and configuration without a device model:

```
hKitchenLight = dpws_create_custom_device(0, -1);
// Configure the mandatory device attributes.
DPWS_ADD_PTR_ATT(hKitchenLight, DPWS_PTR_TYPE,
                 &SimpleLightType);
DPWS_SET_INT_ATT(hKitchenLight, DPWS_INT_METADATA_VERSION, 1);
ls.s = "Kitchen light";
DPWS_SET_STR_ATT(hKitchenLight, DPWS_PTR_FRIENDLY_NAME, &ls);
ls.s = "BrightBulb";
DPWS_SET_STR_ATT(hKitchenLight, DPWS_PTR_MODEL_NAME, &ls);
ls.s = "Electrical SA";
DPWS_SET_STR_ATT(hKitchenLight, DPWS_PTR_MANUFACTURER, &ls);
// Configure the optional device attributes.
DPWS_ADD_STR_ATT(hKitchenLight, DPWS_STR_SCOPE, diversified_scope);
DPWS_SET_STR_ATT(hKitchenLight, DPWS_STR_FIRMWARE_VERSION, "1.0");
DPWS_SET_STR_ATT(hKitchenLight, DPWS_STR_SERIAL_NUMBER, "56080001");
DPWS_SET_STR_ATT(hKitchenLight, DPWS_STR_MODEL_NUMBER, "1.0");
DPWS_SET_STR_ATT(hKitchenLight, DPWS_STR_MODEL_URL,
                 "http://www.electrical.com/BrightBulb.html");
DPWS_SET_STR_ATT(hKitchenLight, DPWS_STR_PRESENTATION_URL,
                  "LightsOverview.html");
DPWS_SET_STR_ATT(hKitchenLight, DPWS_STR_MANUFACTURER_URL,
                 "http://www.electrical.com");
// User data that will be accessible in the service implementation
DPWS_SET_PTR_ATT(hKitchenLight, DPWS_PTR_USER_DATA, &simpleLightState);
```

- Creation of services: in the case where the device is created without hosted services (as in the example above), hosted services must be explicitly created and attached to the device. This is done using the **dpws_create_hosted_service** function, which takes as parameters a device reference and a service class reference. The following code snippet shows an example of service creation. There is no specific configuration required, as the service will use the service class Id as service Id:

```
hSPServ = dpws_create_hosted_service(hKitchenLight, hServClass);
```

- Creation of service ports: the last step in the registry configuration is the creation of service ports for all configured services. Service ports are created using the dpws_create_service_port function, and then associated to a service using the dpws_bind_service function. The following code snippet shows an example of service port creation, configuration and binding:

```
hSPServPort = dpws_create_service_port();
DPWS_SET_STR_ATT(hSPServPort, DPWS_STR_ADDRESS, "SwitchPowerService");
dpws_bind_service(hSPServ, hSPServPort);
```

All object creation functions return an invalid handle (-1) if an error occurred.

Once devices and their hosted services are created and configured, they can be published on the network by calling the **dpws_enable_device** function. When using a static configuration, as described in this section, the role of this function is to schedule the WS-Discovery Hello messages before starting the main server loop. These messages will then be sent at server start time.

```
status = dpws_enable_device(hKitchenLight);
… // Error handling code
dpws_release_handle(hKitchenLight);
```

Once a device is enabled, its handle ownership is transferred to the registry, and the handle can be safely released by the application.

Advanced use:
This section has introduced a static configuration mechanism for initializing devices and hosted services. The DC toolkit provides more advanced mechanisms for managing the registry, including:

- The use of an XML configuration file. See "XML configuration" for details.
- An API for dynamically changing the contents of the registry while running. See "Dynamic registry modification API" for details.
- A built-in management service for dynamically changing the contents of the registry at runtime. See "Dynamic deployment" for details.

PERSISTENT INFORMATION MANAGEMENT

One of the tricky aspects of the DC runtime configuration is the management of persistent state information that is required for the proper implementation of the WS-Discovery and DPWS protocols:

- The device identifier must be unique and stable over reboots.
- Discovery messages sent by devices must include a sequence number that must be incremented at each reboot.
- The metadata version associated to each device must be incremented each time the device or hosted service metadata is modified.
- Some device attributes are used to "personalize" a given device implementation for a specific device. Such information cannot be stored in the implementation code and requires specific external storage.

The following table shows the set of device and hosted service attributes that are controlled by the device metadata version and require a metadata version increment each time they change. It also distinguishes device attributes that are attached to the implementation and those that are specific to a given device instance.

Device attributes	Device personalization attributes	Hosted service attributes
Types	DeviceId	Address
Manufacturer	Scopes	Types
ManufacturerURL	FriendlyName	ServiceId
ModelName	SerialNumber	WSDL
ModelNumber		
ModelURL		
PresentationURL		
FirmwareVersion		

Although it does not explicitly appear in the table, a change to the set of services hosted by a device also requires a metadata version increment.

Warning: some changes in a device state may occur spontaneously and require a metadata version increment:

- A hosted service address normally uses a HTTP URL that contains the IP address of its hosting device: as this address may change upon reboot or renewal of a DHCP lease, the hosted service address may change as well.
- When using the default configuration mechanism provided by the DC API, each hosted service is associated to a service port which uses a generated UUID as relative address. This will therefore change upon each reboot.

The basic configuration API of the DC toolkit only provides limited support for managing persistent information:

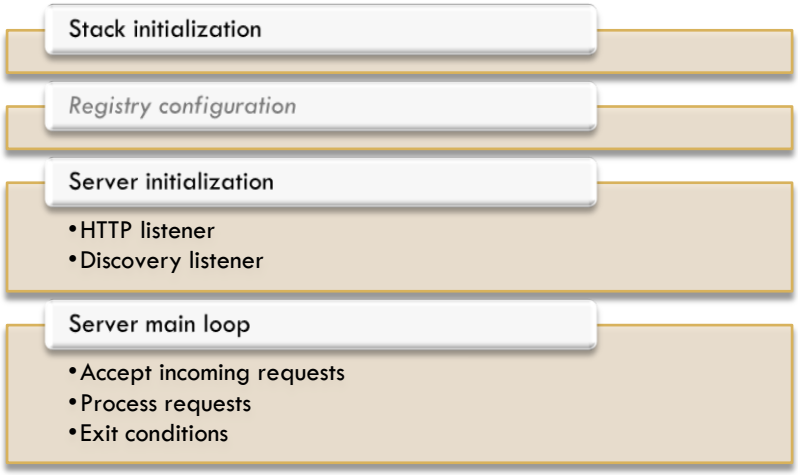
- The boot sequence number must be explicitly set when initializing the DC stack. It is the responsibility of the application to store and increment it at each reboot. When running the DC stack on a platform that has access to calendar time, the number of elapsed seconds since the epoch can be used as boot sequence number.
- The metadata version of a device must be stored and incremented by the application each time the device metadata is changed. When the metadata changes at each reboot (due for instance to a change of IP address), the boot sequence number may be used as metadata version. Note however that such a dynamic behavior is not recommended: frequent changes in metadata induce additional metadata requests from clients, which often use a caching mechanism for optimizing their access to metadata.
- The device identifier may be automatically generated from the MAC address of one of the device network interfaces, thus ensuring uniqueness and stability. The rest of the "personalization" attributes (Scopes, FriendlyName and SerialNumber) must be stored and managed by the application.

Advanced use:
The XML configuration file can be used to store all the required persistent information. See "Registry configuration format" for details.

IMPLEMENTING THE SERVER ARCHITECTURE

The last step in the development of a device and its hosted services is the implementation of the server behavior, i.e. the loop that is waiting for incoming requests to process them.

The following diagram shows the main sequence of operations that must be executed in the device implementation code to set up the server. In this figure, the second step (registry configuration) has already been extensively discussed in the previous section. The three other steps are further detailed in this section.



STACK INITIALIZATION

The first DC API function that must be called in all applications is one of the stack initialization functions, `dpws_init`, `dpws_init6` or `dpws_init_ex`. The first two functions initialize the stack for the IPv4 or IPv6 protocols respectively, using one of the available network interfaces.

Advanced use:
The **dpws_init_ex** can be used to configure the stack for using both IPv4 and IPv6 at the same time and more than one network interface. See "Multiple network interfaces and IP protocols" for details.

Warning: the selection of the default interface when using the **dpws_init** or **dpws_init6** function can lead to unexpected behavior on systems featuring several interfaces (e.g. a PC under Windows). In such a case, one should either deactivate all the unused interfaces, or use the **dpws_init_ex** with the appropriate filter to select the right interface.

Calling a stack initialization function after one of them has already been called has no effect.

Once the stack is initialized, it is necessary to configure global DC registry parameters:

- Boot sequence: see discussion of this parameter in the previous section.
- HTTP port number: this is the port of the network listener on which incoming Web Services requests will be received. It defaults to 80 (standard HTTP port).

Warning: the above parameters are used during device configuration, so must be set before the registry configuration phase.

Global parameters configuration is performed by using the attribute setter functions described previously on a special predefined handle reference.

The following code snippet shows an example of stack initialization:

```
status = dpws_init();
… // Error handling code
DPWS_SET_INT_ATT(DC_REGISTRY_HANDLE, DPWS_INT_HTTP_PORT, port);
DPWS_SET_INT_ATT(DC_REGISTRY_HANDLE, DPWS_INT_BOOT_SEQ, bootSeq);
```

CONFIGURING THE SERVER AND ITS LISTENERS

This step is performed after the registry configuration. It is used to configure and create the server objects that listen for incoming discovery and Web Services messages.

The DC API provides two functions for initializing the server, **dpws_server_init** and **dpws_server_init_ex**. Both functions take a **struct dpws** pointer as first parameter, and configure it for server operations. Other server configuration parameters, available when using the **dpws_server_init_ex** function, include:

- HTTP server activation flag: this flag controls whether the internal HTTP server of the DC toolkit should be used or not. It is set by default.
- HTTP server backlog: this parameter controls the size of the HTTP socket backlog.
- HTTP server TCP bind flags: these flags are passed to the HTTP socket upon creation.
- Discovery server activation: This flag controls whether the discovery listener should be started. It is set by default.

The following code snippet shows an example of server initialization:

```
status = dpws_server_init(&dpws, NULL);
… // Error handling code
```

Advanced use:
Additional server configuration mechanisms are available for customizing the DC server behavior once the server has been initialized. These optional features include:

- Support of connection keep-alive on the server side. See "Connection keep-alive" for details.
- Support for MTOM attachments. See "MTOM support" for details.
- Support for WS-I Basic Profile 1.1. See "Basic Profile 1.1 support" for details.
- Use of an external Web server in replacement to the internal DC Web server. See "External Web server integration" for details.

IMPLEMENTING THE SERVER LOOP

The server normally runs inside an infinite loop, repeatedly performing the following actions:

- Accepting new requests: this is a blocking call that waits until a new request is available on either the discovery listener or the HTTP listener. This function sets up a request context for processing the request.
- Processing the current request using the request context previously set up.
- Cleaning up the request context.

The DC toolkit API supports the execution of the above loop in both mono-threaded and multi-threaded mode.

In mono-threaded mode, the **`dpws_accept`** must be used to wait for incoming requests. This function takes as single parameter the **`dpws struct`** pointer that has previously been configured with the **`dws_server_init`** call. Upon successful return of the function, the **`dpws struct`** pointer also represents the request context of the incoming request. This pointer can then be used to process the request, by calling the **`dpws_serve`** function. This function is responsible for dispatching the request and invoking the appropriate skeleton, which ultimately calls the user-defined service function with the request context as parameter. The request context must be cleaned up after processing by calling the **`dpws_end`** function with the request context as argument. This function frees all resources and memory used for processing the request. After clean up, the **`dpws struct`** pointer can still be used by the next **`dpws_accept`** call.

The following code snippet shows an example of a mono-threaded server loop:

```
while (TRUE) {
       status = dpws accept(&dpws);
… // Error handling code
       status = dpws_serve(&dpws);
… // Error handling code
       dpws end(&dpws);       // frees transient message memory.
}
```

In multi-threaded mode, each request is processed in a separate thread, using an independent request context, while the request listener runs in the main server thread, using the **`dpws_accept_thr`** function to wait for incoming requests. This function takes two **`dpws struct`** pointers as parameters: the first one represents the server context (previously initialized using **`dpws_server_init`**), the second one represents the request context to be initialized by the function call. Upon successful return of the function, the request context can be used in a separate thread as a parameter to the **`dpws_serve`** and **`dpws_end`** functions.

Warning: in multi-threaded mode, it is absolutely mandatory that each thread uses a separate **`dpws_struct`** pointer as request context, as the internal fields of this struct contain request-specific data and are not protected against race conditions.

An example of a simple multi-threaded server is given in the package samples.

EXITING THE SERVER LOOP

The above discussion has introduced the server loop as an infinite loop. However, it is sometimes necessary to exit the server loop in an orderly manner. The DC toolkit provides the following mechanism for such a case:

- The **`dpws_stop_server`** and **`dpws_stop_server_no_wait`** functions can be used to asynchronously interrupt the **`dpws_accept`** (or **`dpws_accept_thr`**) function.
- The status code returned by the dpws_accept function can be tested for the specific interrupt code, and the loop exited with no error, as shown in the code snippet below.

```
while (TRUE) {
        status = dpws_accept(&dpws);
        if (status == DPWS_ERR_SERVER_STOPPED) {
                break;  // Exit loop
        } else if (status) {
                … // Error handling code
        }
        status = dpws_serve(&dpws);
… // Error handling code
        dpws_end(&dpws);       // frees transient message memory.
}
```

After the server loop is exited, it is possible to clean up the server configuration by calling the **`dpws_server_shutdown`** function. The complete DC stack can also be shut down, freeing all resources, by calling the **`dpws_shutdown`** function.

COMPILING AND LINKING

The DC runtime environment is provided as a set of static or shared libraries that must be linked with both the generated code for the implemented Web services and the application code to produce an executable server.

THE DPWS CORE LIBRARIES

The following table shows the names of the static and shared libraries that implement the DC runtime. The names are given for the Windows platform, but equivalent ones are defined for the Linux platform.

Static libraries	Shared libraries
al.lib	dcruntime.dll
common.lib	
dcpl.lib	
dpwslib.lib	

All the necessary functions and structs declarations are provided in the **`dc/dc_Dpws.h`** header file, which must be included in all application files using the DC toolkit API.

***Advanced use*:**
The DC toolkit provides additional libraries implementing advanced features. These libraries are described in their respective sections of the "Advanced features" chapter.

GENERATED SERVER AND APPLICATION FILES

The set of files that must be compiled and linked with the DC runtime libraries include:

- For each Web Service called **`ws`**, the generated file called *wsServerLib*.c. This file includes both the skeleton code (*wsServer*.c) and the marshalling/unmarshalling code (*ws*C.c).
- For each Web Service called **`ws`**, the file containing the implementation of the service functions. This file should also include the *ws.nsmap* generated file (note that this file contains global variable declarations and should therefore be included only once in an application).
- The file(s) containing the registry configuration code and the implementation of the server main loop.

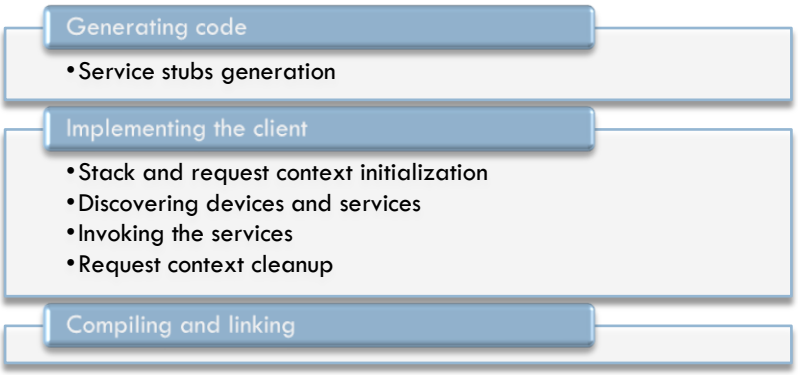
Examples of projects for both the Windows and the Linux platforms are provided in the package samples.

Warning: forgetting to include the *ws.nsmap* file or including it more than once is a usual source of link errors when getting started with the DC toolkit.

PURE CLIENT DEVELOPMENT

DEVELOPMENT PROCESS OVERVIEW

The following figure highlights the main steps in the development of a pure client using the DC toolkit.



The client development process is much simpler than the server development process and requires much less implementation work.

The process starts from a WSDL document describing the service to be invoked. It is similar to a standard Web Services client development process, extended with the use of discovery mechanisms.

The main steps of this process are further detailed in the following sections.

CODE GENERATION

Generating the code for a Web Service client involves:

- Retrieving the service WSDL file;
- Generating code from the WSDL file, including the service stubs and the marshalling/unmarshalling code.

CODE GENERATION TOOLS

The code generation for the client uses the **`wsdl2h`** and **`soapcpp2`** tools already introduced for server-side code generation.

The following commands are required for generating the client code from a WSDL document:

```
wsdl2h -c -o example.gsoap -t typemap.dat example.wsdl
soapcpp2 -2ucn -pws -d gen example.gsoap
```

These commands and options are the same as those used for the server-side code generation, which are detailed in "Code generation". Only the following generated files are required when implementing a service client: *wsC.c*, *wsH.h*, *wsStub.h*, *ws.nsmap*, *wsClient.c* and *wsClientLib.c*.

GENERATED STUBS

The generated code for the client side (included in the generated *wsClient.c* file) contains the service stubs that can be used to invoke service operations. Two types of stubs are generated, depending on the type of operation.

For each one-way (input-only) operation declared in the service annotated header file, a corresponding **`dpws_send_<operation name>`** function is generated (the

operation name is of the form **__ns__oper** for document/literal operations). The generated function has the following parameters:

- The first parameter (*dpws*) is a pointer to a **struct dpws** that must be initialized by the user code.
- The second parameter (*to*) is a pointer to a **struct wsa_endpoint_ref** that represents the destination of the message. This endpoint reference usually contains the network address (i.e. URL) of the invoked service for a hosted service, but can also contain the logical address of a DPWS device when the invoked service is exposed by the device endpoint.
- The remaining parameters are the same as those declared in the annotated header file, except for the final **void** parameter that is dropped. They represent the body of the message and will be serialized into XML using the marshalling code.

The following is an example of a generated one-way stub declaration:

```
int dpws_send__wsh__LaunchCycle(
      struct dpws*,
      struct wsa_endpoint_ref*,
      struct _wsh__LaunchCycle *wsh__LaunchCycle);
```

For each request/response (input-output) operation declared in the service annotated header file, a corresponding **dpws_call_<operation name>** function is generated. The generated function has the following parameters:

- The first parameter (*dpws*) is a pointer to a **struct dpws** that must be initialized by the user code.
- The second parameter (*to*) is a pointer to a **struct wsa_endpoint_ref** that represents the destination of the message. It follows the same rules as those described for one-way operations.
- The third parameter (*replyTo*) is another pointer to a **struct wsa_endpoint_ref** that represents the network address to which the response must be sent.
- The remaining parameters up to the last one are the same as those declared in the annotated header file. They represent the body of the request and will be serialized into XML using the marshalling code.
- The last parameter is the same as the last parameter declared in the annotated header file. It is an output parameter that represents the body of the response message.

The following is an example of a generated request/response stub declaration:

```
int dpws_call__wsh__GetCycleStatus(
      struct dpws*,
      struct wsa_endpoint_ref*,
      struct wsa_endpoint_ref*,
      struct _wsh__CycleStatus *wsh__CycleStatus);
```

CLIENT IMPLEMENTATION

The four main steps required for invoking a service from a client are:

- Initialization of the DC runtime environment and of the request context.
- Discovery of the devices and services.
- Invocation of the discovered services.
- Request context clean up.

CLIENT-SIDE INITIALIZATION

As for the server side, the first DC API function that must be called in client applications is one of the stack initialization functions, **`dpws_init`**, **`dpws_init6`** or **`dpws_init_ex`**. The first two functions initialize the stack for the IPv4 or IPv6 protocols respectively, using one of the available network interfaces.

Advanced use:
The **`dpws_init_ex`** can be used to configure the stack for using both IPv4 and IPv6 at the same time and more than one network interface. See "Multiple network interfaces and IP protocols" for details.

Warning: the selection of the default interface when using the **`dpws_init`** or **`dpws_init6`** function can lead to unexpected behavior on systems featuring several interfaces (e.g. a PC under Windows). In such a case, one should either deactivate all the unused interfaces, or use the **`dpws_init_ex`** with the appropriate filter to select the right interface.

The next step requires the creation and initialization of a request context, i.e. a **`struct dpws`** object, which will be passed to all API and generated functions that require network access: this object can be allocated statically, on the stack or on the heap. Once the object has been allocated, it must be initialized using the **`dpws_client_init`** function.

Warning: although it is possible to allocate the **`struct dpws`** object on the stack, this may create problems in small embedded devices, as the object holds several large buffers (adjustable at compile time when building the DC runtime libraries) and can therefore exceeds the allocated stack space.

Advanced use:
The DC toolkit provides additional request context configuration mechanisms for customizing the stack behavior. These optional features include:

- Support of keep-alive connections on the client side through the use of a connection pool. See "Connection keep-alive" for details.
- Support for MTOM attachments. See "MTOM support" for details.
- Support for WS-I Basic Profile 1.1. See "Basic Profile 1.1 support" for details.

DEVICES AND SERVICES DISCOVERY AND METADATA ACCESS

The DC runtime environment provides an API allowing clients to discover and retrieve metadata information about devices and hosted services available on a local network. This API is based upon the set of discovery and metadata transfer messages defined by the WS-Discovery, WS-MetadataExchange and DPWS specifications. In order to limit the number of message exchanges over the network, the DC runtime uses a cache mechanism to store discovery and metadata query results.

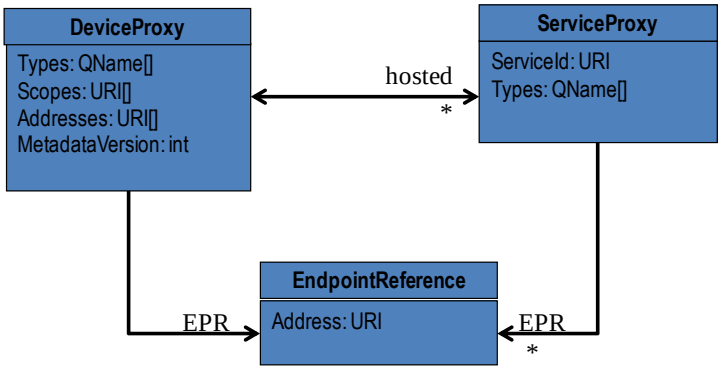
The DC cache has the following characteristics:

- The cache manages proxies for both devices and services: these proxies are exposed to the developer through proxy handle references, represented as **`short`** integers. This approach allows the DC runtime to internally maintain reference counters on all cache proxies, thus supporting asynchronous changes to the cache induced by its dynamic nature. A number of API functions are provided

to manage proxies and their references, including the **dpws_pin_proxy** and **dpws_release_proxy** functions, which respectively increment and decrement a proxy reference counter. A proxy with a reference counter equal to zero can be discarded at any time by the cache.

- All discovery query results go through the cache before being returned to the user: while the API provides a mechanism to choose whether querying the cache first or directly querying the network, query results are always used to update the cache and create proxies before being returned.
- When connected to a discovery listener, the cache is asynchronously and dynamically updated upon reception of Hello and Bye messages. It is not the case in pure client scenarios, but it is in peer-to-peer or asynchronous client scenarios, as described in the following chapters. Because of this asynchronous update mechanism, a proxy may become invalid while still in use by the application. An invalid proxy will however not be discarded by the cache while its reference counter is positive.
- The cache does not store some of the informative metadata for devices: only device discovery information (types, scopes, transport addresses and metadata version) and hosted services information (types, service ids and addresses) are maintained in the cache. Device model and instance information, as well as hosted services WSDL information are retrieved from the network upon request.

The following diagram shows a model of the main objects stored in the cache.



Advanced use:
The cache API provides some advanced configuration features supporting:

- The definition of callback functions called upon asynchronous modification of the cache through Hello and Bye messages. See “Lifecycle callbacks” for details.
- The definition of cache size control mechanisms, based on filters or maximum size. See “Cache content control” for details.

The DC API provides three different ways to discover devices:

- *Using types and scopes*: the **dpws_lookup** and **dpws_lookup_ex** functions can be used to retrieve a given number of devices matching the set of types (expressed as expanded QNames) and scopes provided as parameters. When not enough matching results are available in the cache, these functions generate WS-Discovery Probe messages to perform their queries. Both functions return an array of device proxy references, together with the array size. When no devices match the query, the array is empty and its size equal to zero. Because the underlying Probe messages use a multicast protocol, a timeout is used to limit the interval during which ProbeMatches messages are expected. The

dpws_lookup_ex function supports more flexible filters, as well as a user-defined timeout and a mechanism to bypass the cache.

- *Using a device logical id (UUID)*: the **dpws_lookup_by_id** function can be used to retrieve a device based on its logical id. When the device is not in the cache, this function generates WS-Discovery Resolve messages to perform its query. Here also, a timeout is used for the reception of the ResolveMatches message. This function returns a device proxy reference, or an invalid reference (-1) if no matching device is found.
- *Using a device transport address*: the **dpws_probe_address** or **dpws_probe_address_ex** functions can be used to retrieve a device based on its transport address (which should be an HTTP URL), also checking that it has the expected types and scopes. When no matching device is in the cache, this function generates a WS-Discovery DirectedProbe message (using HTTP) to perform its query. This function returns a device proxy reference, or an invalid reference (-1) if no matching device is found.

All these discovery functions take as first parameter a pointer to the previously initialized request context. This context is used to store transient and dynamically allocated data while sending and receiving messages, but also to maintain the list of returned proxy references. All returned proxies have their reference count automatically incremented, and are released only after the next call to the **dpws_end** function. When the application still needs to use a proxy after a call to **dpws_end**, it should explicitly increment the proxy reference counter by calling **dpws_pin_proxy**.

Warning: some of the discovery functions described above call the **dpws_end** function internally. Therefore, proxies returned by previous calls should be considered released after a new call unless explicitly pinned.

The following code snippet shows an example of use of the **dpws_lookup** function:

```
status = dpws init();
… // Error handling
status = dpws_client_init(dpws, NULL);
… // Error handling
nb devices = 3;       // Max number of devices
device proxies = dpws lookup(dpws,
                 "http://www.soa4d.org/DPWS/Samples/Lights",
                 "SimpleLight",
                 "urn:myScope",
                 &nb devices);
if (nb devices < 1) {
… // No device found
}
device_proxy = device_proxies[0];
status = dpws pin proxy(device proxy);
… // Error handling
dpws_end(dpws);
```

Once a device proxy has been retrieved, it is possible to obtain the device hosted services from the proxy:

- *Retrieving hosted services by type*: hosted services for a device can be retrieved using the **dpws_get_services** function. This function takes as parameters a device proxy and an optional type (as an expanded QName), and returns an array of matching service proxies, along with the array size.
- *Retrieving hosted services by id*: a named hosted service for a device can be retrieved using the **dpws_get_service_by_id** function. This function takes as parameters a device proxy and a service id, and returns a service proxy reference, or an invalid reference (-1) if no matching service is found.

Both functions take as first parameter a pointer to the previously initialized request context. The rules for the management of the returned service proxy references are the same as those exposed for the device proxies.

The following code snippet shows an example of use of the **dpws_get_services** function. It assumes that a device proxy has already been obtained:

```
nb_services = 1;      // Max number of services
service proxies = dpws get services(dpws, device proxy,
                 "http://www.soa4d.org/DPWS/Samples/Lights",
                 "SwitchPower",
                 &nb_services);
if (nb_services != 1) {
… // No service found
}
service proxy = service proxies[0];
status = dpws pin proxy(service proxy);
… // Error handling
dpws_end(dpws);
```

The DC API also provides a number of functions for accessing device and hosted services metadata. Some of these functions directly access information stored in the cache, while others perform a service request to retrieve the information from the remote device or service. A detailed description of these functions is provided in the client API section of the reference manual.

SERVICE STUB INVOCATION

In the DC toolkit, service invocation is based on WS-Addressing endpoint references that contain the addresses of the devices and services. Endpoint references are represented by **struct wsa_endpoint_ref** objects. Endpoint references can be built manually, but in most cases will be obtained from the device and service proxies in the cache. Two functions are provided:

- **dpws_get_endpoint_refs**: this function returns the list of endpoint references associated to a proxy. In the case of a device proxy, a singleton endpoint reference should be returned, containing as address the device logical id. In the case of a service proxy, more than one endpoint reference may be returned.
- **dpws_get_default_endpoint_ref**: this function returns the default endpoint reference associated to a proxy. In the current implementation, it is simply the first endpoint reference in the list returned by the **dpws_get_endpoint_refs** function.

Both functions take as first parameter a pointer to the previously initialized request context, which may be **NULL**. When this parameter is set, the returned endpoint references are allocated on the transient memory associated to the request context and will be freed by the next call to **dpws_end**; otherwise, they are allocated on the process heap and must be explicitly freed after use using the **dpws_endpoint_ref_free** and **dpws_endpoint_ref_array_free** functions.

Warning: in the current version of the DC toolkit, endpoint references should either be device endpoint references, using the device logical id as address, or service endpoint references using HTTP URLs as address. In addition, the DC toolkit provides only limited support for WS-Addressing reference parameters in endpoint references: only reference parameters predefined in the DPWS specification are currently supported.

One-way operation invocations use the generated **dpws_send_XXX** stubs. The parameters to those functions are:

- A pointer to the previously initialized request context.

- A pointer to the endpoint reference representing the service endpoint. It is generally an endpoint reference obtained from a service proxy, but it can also be obtained from a device proxy, when services are exposed on a device endpoint. The endpoint reference may also be built manually.
- The parameter(s) representing the message body, which are operation-specific.

Although the operations are one-way, they use the HTTP protocol as underlying transport mechanism, and the call will therefore block until it receives the expected empty HTTP response.

Request/response operation invocations use the generated **`dpws_call_XXX`** stubs. The parameters to those functions are:

- A pointer to the previously initialized request context.
- A pointer to the endpoint reference representing the service endpoint. It is generally an endpoint reference obtained from a service proxy, but it can also be obtained from a device proxy, when services are exposed on a device endpoint. The endpoint reference may also be built manually.
- A pointer to the endpoint reference representing the address (*replyTo* endpoint) to which the response must be sent. This parameter can be **`NULL`**. When specified, it is normally built manually, as it does not generally correspond to a cached service endpoint. When this parameter is set to **`NULL`**, the response message will be sent synchronously to the client: this corresponds to the usual behavior of request/response operations using SOAP over HTTP. Otherwise, the server will send the response as a one-way message to the address specified in the endpoint reference, by opening a new channel to the *replyTo* address.
- The parameter(s) representing the message body, which are operation-specific.
- The output parameter representing the message response, which is also operation-specific. The content of this parameter is dynamically allocated on the transient memory associated to the request context, and will be freed by the next **`dpws_end`** call. This parameter will not be filled and may be set to **`NULL`** when the *replyTo* parameter is not **`NULL`** (as the response is not received synchronously).

Request/response operations use the HTTP protocol as underlying transport mechanism, and the call will therefore block until it receives the expected HTTP response. In the usual case where the *replyTo* parameter is **`NULL`**, the HTTP response contains the response message; otherwise, the server will send the response message to the *replyTo* endpoint in a separate connection, and the HTTP response should be empty. When the *replyTo* endpoint is exposed by the client itself, as described in the asynchronous client chapter, this mechanism provides an easy way to implement an asynchronous request/response message exchange pattern.

Warning: the DPWS specification explicitly disallows the use of the *replyTo* endpoint in request/response operations. Although devices developed with the DC toolkit support this advanced feature, other implementations may not and may therefore reject service invocations using a non-**`NULL`** *replyTo* parameter.

The following code snippet shows an example of service endpoint reference retrieval and service stub invocation:

```
servEndPt = dpws_get_default_endpoint_reference(dpws, service_proxy);
if (!servEndPt) {
… // Error handling
}
status = dpws_send___lit__Switch(dpws, servEndPt, lit__PowerState__ON);
if (status) {
… // Error handling
```

```
}
// Clean up
dpws_end(dpws);
dpws_release_proxy(device_proxy);
dpws_release_proxy(service_proxy);
```

Advanced use:
In addition to the use of generated stubs, the DC toolkit also provides support for generic message processing functions on the client side. Instead of using generated C structs to handle the XML content of the SOAP messages, these functions use a streaming XML API to access to this content. See "Generic invocation" for details.

Both one-way and request/response stubs return a status that can be checked for error handling. Errors can be of three types:

- Local API errors: these can occur when invalid parameters are passed to the generated stubs.
- Remote SOAP faults: these can only occur for request/response stubs and correspond to errors on the server side.
- Network errors: these can occur when the remote peer becomes unreachable.

In the latter case, the proxy associated to the unreachable endpoint reference is automatically marked as invalid. Invalid proxies are discarded from the cache as soon as their reference counter becomes null. Proxies may also be explicitly marked as invalid using the **`dpws_invalidate_proxy`** function. A proxy status may be checked using the **`dpws_check_proxy`** function.

REQUEST CONTEXT CLEAN UP

The final step in the service invocation from a client is the cleanup of the request context and other objects used during the invocation.

The request context clean up is performed by calling the **`dpws_end`** function. The role of this function is:

- To free all transient memory allocated during message processing, including the response structure contents returned as output parameter by stubs.
- To free all transient memory associated to the request context allocated by API functions.
- To release all proxies that have been returned by device and service lookup functions.

Because the **`dpws_end`** function has a direct impact on the amount of dynamic memory consumed by an application, it is important to call it often enough to avoid large peaks of memory use.

In addition, applications should also:

- Free all transient information returned by API functions that is dynamically allocated on the process heap. The API reference manual clearly identifies the few functions that perform dynamic memory allocation on the heap.
- Release all proxies that have been explicitly pinned.

Compiling and linking

The DC runtime environment is provided as a set of static or shared libraries that must be linked with both the generated code for the invoked Web services and the application code to produce an executable client.

The DPWS Core libraries

The following table shows the names of the static and shared libraries that implement the DC runtime. The names are given for the Windows platform, but equivalent ones are defined for the Linux platform.

Static libraries	Shared libraries
al.lib	dcruntime.dll
common.lib	
dcpl.lib	
dpwslib.lib	

All the necessary functions and structs declarations are provided in the `dc/dc_Dpws.h` header file, which must be included in all application files using the DC toolkit API.

Advanced use:
The DC toolkit provides additional libraries implementing advanced features. These libraries are described in their respective sections of the "Advanced features" chapter.

Generated client and application files

The set of files that must be compiled and linked with the DC runtime libraries include:

- For each Web Service called **`ws`**, the generated file called *wsClientLib*.c. This file includes both the stub code (*wsClient*.c) and the marshalling/unmarshalling code (*ws*C.c).
- The file(s) containing the client code used to invoke the generated stubs. One of these files should also include the *ws.nsmap* generated file for each used Web Service (note that this file contains global variable declarations and should therefore be included only once in an application).

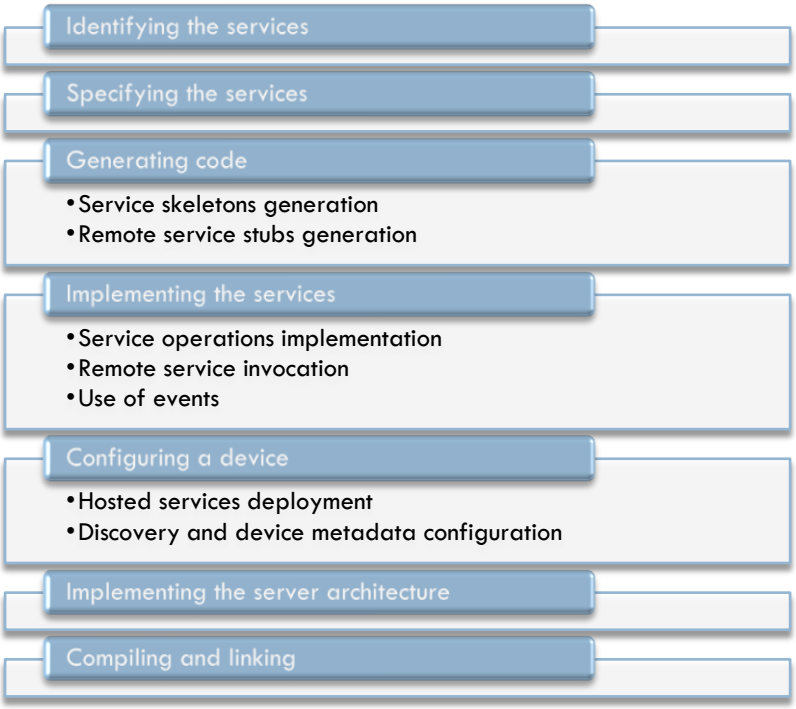
Examples of projects for both the Windows and the Linux platforms are provided in the package samples.

Warning: forgetting to include the *ws.nsmap* file or including it more than once is a usual source of link errors when getting started with the DC toolkit.

PEER-TO-PEER CLIENT DEVELOPMENT

DEVELOPMENT PROCESS OVERVIEW

The following figure highlights the main steps in the development of a device that also acts as a peer-to-peer client, using the DC toolkit.



These steps are almost the same as those defined for a simple device development, except for:

- The additional generation of remote services stubs.
- The services implementation, which requires the integration of the remote service stubs invocation inside the service function.
- The compilation and link stage, during which both server-side and client-side generated code must be combined.

The three above steps are further detailed in the following sections. The description of the other steps can be found in the "Services and device development" chapter.

CODE GENERATION

The code generation phase for a device acting as a peer-to-peer client combines the code generation of a server and a client:

- For each Web Service **`ws`** implemented by the device, the service skeletons must be generated.
- For each Web Service **`rws`** invoked by the device, the service stubs must be generated.

The tools and command line arguments previously described in "Code generation" are still applicable in this scenario. The following generated files are needed by the implementation:

- For each Web Service **`ws`** implemented by the device: *wsC.c*, *wsH.h*, *wsStub.h*, *ws.nsmap*, *wsServer.c* and *wsServerLib.c*.

- For each Web Service **`rws`** invoked by the device *rwsC.c*, *rwsH.h*, *rwsStub.h*, *rws.nsmap*, *rwsClient.c* and *rwsClientLib.c*.

INTEGRATING SERVER AND CLIENT CODE

The invocation of service stubs within service implementation code is similar to the simple service invocation described in the "Client implementation" section, and requires the following steps:

- Request context initialization.
- Devices and services discovery.
- Service invocation through generated stubs.
- Request context clean up.

However, there are a few differences induced by the fact that the client is executed within a device. They are detailed below.

CACHE INITIALIZATION AND UPDATE

A device acting as a peer-to-peer client uses the discovery mechanisms offered by the DC toolkit to discover its peers and the hosted services they expose. Device peers and their services therefore appear as proxies in the device cache.

Unlike a pure client, a device always has a running discovery listener, in order to answer to discovery requests sent by clients. In the DC toolkit, this listener is also used to dynamically update the cache when Hello and Bye messages are received.

Advanced use:
In order to monitor and control the number of proxies stored in the cache, the cache API provides some advanced configuration features supporting:

- The definition of callback functions called upon asynchronous modification of the cache through Hello and Bye messages. See "Lifecycle callbacks" for details.
- The definition of cache size control mechanisms, based on filters or maximum size. See "Cache content control" for details.

Several strategies can be used to retrieve the peer service endpoint references required by the remote service stubs:

- Initialize the cache at start-up time, and store the retrieved peer endpoint references for future use in the service functions. With this approach, changes in the network topology may result in invalid endpoint references. The service invocation code must therefore be prepared to handle communication errors and update the cache and the endpoint references by performing a discovery lookup after a failure. Registering appropriate callback functions in the cache to monitor asynchronous changes may help reduce the number of unexpected failures.
- Invoke the discovery functions before each service stub invocation: this approach is less efficient, but ensures that peer proxies and endpoints are up-to-date. In most cases, because the cache is asynchronously updated by discovery announcement messages, the lookup request will retrieve the requested information in the cache and will not generate a network request, thus limiting the performance penalty.

***Advanced use*:**
The XML-based configuration mechanism provides support for declaring peer references and for binding them at runtime. See "XML configuration" for details.

Invoking a remote operation from a service function

Service stub invocations inside a service function follow the rules previously described for pure clients. Care must however be taken to avoid possible confusion between two distinct request contexts, represented as `struct dpws` pointers:

- The server request context: this context is passed to the service function by the DC runtime, and can be used to retrieve information about the request being processed, as well as to allocate memory (using `dpws_malloc`) to populate the service function response structure.
- The client request context: this is a separate request context, which must be allocated and initialized by the application code before being passed as parameter to the stub invocation. It is used as described in the pure client chapter, and must generally be cleaned up after use using the `dpws_end` function. A client request context may be reused in subsequent invocations, as long as it is used by no more than one thread at a time. The `dpws_client_init` function only needs to be called once before the first use.

Warning: Mixing up server and client request contexts in API calls will lead to unpredictable results.

Compiling and linking

The DC runtime environment is provided as a set of static or shared libraries that must be linked with the generated code for both the implemented Web services and the invoked Web services and with the application code to produce an executable server.

The DPWS Core libraries

Libraries and associated header files are the same as those listed for a simple device development.

Generated server, generated client and application files

The set of files that must be compiled and linked with the DC runtime libraries include:

- For each Web Service called **ws**, the generated file called *wsServerLib*.c. This file includes both the skeleton code (*wsServer*.c) and the marshalling/unmarshalling code (*ws*C.c).
- For each remote Web Service called **rws**, the generated file called *rwsClientLib*.c. This file includes both the stub code (*rwsClient*.c) and the marshalling/unmarshalling code (*rws*C.c).
- For each Web Service called **ws**, the file containing the implementation of the service functions. The implementation should contain the client code used to invoke the generated stubs for the remote Web Services. This file should also include the *ws.nsmap* generated file, as well as the *rws.nsmap* generated file for each invoked remote Web Service (note that these *nsmap* files contain global variable declarations and should therefore be included only once in an application).
- The file(s) containing the registry configuration code and the implementation of the server main loop.

Note that it is possible to include in the same device both the skeleton and the stub code for the same service.

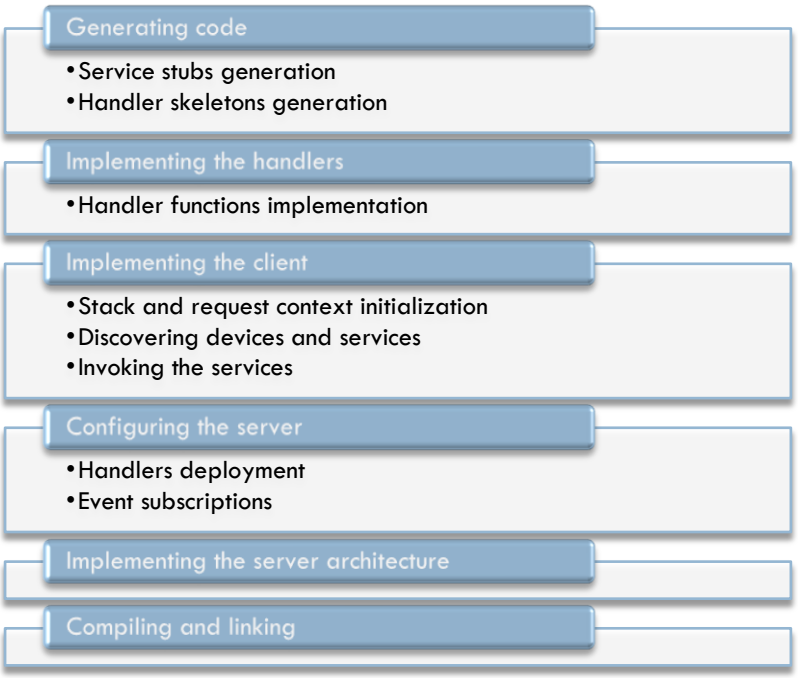
Examples of projects for both the Windows and the Linux platforms are provided in the package samples.

Warning: forgetting to include a *ws.nsmap* or *rws.nsmap* file or including it more than once is a usual source of link errors when getting started with the DC toolkit.

ASYNCHRONOUS AND EVENTING CLIENT DEVELOPMENT

DEVELOPMENT PROCESS OVERVIEW

The following figure highlights the main steps in the development of client that supports the reception of asynchronous responses and events.



With respect to a standard client development, several steps are added or extended:

- Handler skeletons are generated in addition to service stubs. Handler skeletons implement the dispatch mechanisms for handling asynchronous response messages and events.
- Handler skeletons invoke handler functions, which must be implemented by the application developer.
- The asynchronous reception of messages requires a server mechanism: both a server configuration step and the implementation of a server loop are required.
- The compile and link stage is modified to accommodate the new generated handler files.

These steps are further detailed in this chapter.

Note that when the client is also a device (case of the peer-to-peer device described in the "Peer-to-peer client development" chapter), the above steps extend equivalent steps in the device development process, as described in the "Services and device development" chapter. Therefore, the guidelines described in the following sections also apply to that scenario.

CODE GENERATION

Generating the code for asynchronous message handlers involves:

- Retrieving the WSDL file(s) where these handlers are specified;
- Generating code from the WSDL files, including the handler skeletons and the marshalling/unmarshalling code.

Selecting the right set of WSDL files for which handler skeletons must be generated can be more complicated than for a simple Web Service client:

- In the usual case where clients subscribe to events for themselves and request asynchronous responses to be sent back to them in a separate channel, the WSDL files used to generate service stubs and handler skeletons are the same.
- When asynchronous messages and events are received as a result of requests or subscriptions performed by third parties, the set of WSDL files to be retrieved depends on those third party actions.

The code generation for the client uses the **`wsdl2h`** and **`soapcpp2`** tools already introduced for server-side code generation.

The following commands are required for generating the handler skeletons from a WSDL document:

```
wsdl2h –c –o example.gsoap -t typemap.dat example.wsdl
soapcpp2 –2ucn –pws –d gen example.gsoap
```

These commands and options are the same as those used for the server-side code generation, which are detailed in the "Code generation" section. Only the following generated files are required when implementing handlers: *wsC.c*, *wsH.h*, *wsStub.h*, *ws.nsmap*, *wsHandler.c* and *wsHandlerLib.c*. In the usual case where both cient stubs and handler skeletons are used in the same client, the *wsClient.c* and *wsClientLib.c* files must be added to this list.

HANDLER IMPLEMENTATION

Once the handler skeleton is generated, it is necessary to implement the handler functions that are called by the skeleton. The reunion of the generated code and the handler functions represent the complete handler implementation.

The generated skeleton relies on two types of handler functions:

- Each output operation (Notification) declared in a gSOAP annotated header file is associated with a generated skeleton function that invokes an event handler function.
- Each request/response operation declared in a gSOAP annotated header file is associated with a generated skeleton function that invokes a response handler function.

The name and parameters of an event handler function are derived from the operation declaration in the gSOAP annotated header file:

- The name of the event handler function is the name of the output operation (including the double underscore in case of doc/literal operations).
- The first parameter is a pointer to a **`struct dpws`**, passed to the handler function by the DC runtime environment to provide access to the execution context of the current Web Service request.
- The remaining parameters are the same as those declared in the annotated header file, except for the **`void`** parameter that is dropped.

The name and parameters of a response handler function are derived from the operation declaration in the gSOAP annotated header file:

- The name of a response handler function is the name of the request/response operation, followed by **`_handler`**.

- The first parameter is a pointer to a **`struct dpws`**, passed to the handler function by the DC runtime environment to provide access to the execution context of the current Web Service request.
- The second parameter is the last parameter in the annotated header file declaration, which corresponds to the operation response parameter.

The following is an example of an event handler and a response handler declaration invoked by a generated handler skeleton:

```
int   wsh__CycleEnded(
      struct dpws*,
      struct _wsh__CycleEnd *wsh__CycleEnd);

int   wsh__GetCycleStatus handler(
      struct dpws*,
      struct _wsh__CycleStatus *);
```

The rules and guidelines for the implementation of the handler functions are the same as those expressed for service functions implementing one-way operations (handler functions never produce a response message).

Warning: Even in the case where only events or only asynchronous responses are expected to be received by an asynchronous client, all handler functions must be implemented, as link errors would occur otherwise.

Advanced use:
Generic message processing functions may be used in place of the generated handler skeletons. See "Generic invocation" for details.

SERVER CONFIGURATION

Once handlers are implemented, the next step requires the deployment of these handlers and the creation of the event subscriptions.

HANDLER DEPLOYMENT

Like service deployment, handler deployment is done through the configuration of registry objects.

The main registry object used for handler deployment is called an *Endpoint*. An *Endpoint* is a kind of anonymous *ServiceEndpoint*, with an implicit *ServiceClass* and no metadata, as *Endpoints* are not supposed to be advertized to others. *Endpoint* objects are the destination of asynchronous responses and event notifications. They have a subset of the attributes of a *ServiceEndpoint* and *ServiceClass*:

- *DispatchFn*: a set of callback functions that will be called by the DC runtime when dispatching a message sent to the *Endpoint*. The usual case is to use the dispatch function generated by the handler skeleton generator from a WSDL document. User-defined functions using a generic skeleton may also be used.
- *FaultHandlerFn:*a callback function that will be called upon reception of an asynchronous fault. This can be useful as asynchronous response messages may contain a SOAP fault instead of the expected response body.
- *EventEndHandlerFn:* a callback function that will be called upon reception of *SubscriptionEnd* messages. These messages are sent by event sources when they must cancel a subscription before its expected termination.

- *UserData:* a pointer to a user-defined object that can be used to associate application-specific information to the *Endpoint*. This data can be retrieved at execution time in the handler functions.

ServicePort objects can be associated to an *Endpoint* to specify the network addresses on which asynchronous responses and event notifications can be received. The Address attribute of Endpoint service ports follows the same rules as those presented in the "The registry object model" section of the device development chapter.

The DC toolkit API provides the following mechanisms to configure handler endpoints:

- *Endpoint* objects can be created with the **`dpws_create_endpoint`** function. This function does not take any parameters. Besides the endpoint, this function also creates a *ServicePort* object attached to the *Endpoint* and configured with a generated UUID as address. The function returns a endpoint handle reference.
- The *DispatchFn*, *FaultHandlerFn* and *EventEndHandlerFn* callbacks can be set by using the generic attribute setters with the endpoint handle reference as parameter. Note that only the first callback is mandatory.
- When required, the generated *ServicePort* address attached to the *Endpoint* may be overridden by retrieving the service port handle reference using the **`dpws_get_default_service_port`** function and setting its address with the appropriate setter.

The following code snippet shows the creation and initialization of an endpoint used for asynchronous message handling:

```
hEndpoint = dpws create endpoint();
// Configure the endpoint attributes.
DPWS_ADD_PTR_ATT(hEndpoint, DPWS_PTR_HANDLING_FUNCTION,
                 &wsh handle event);
DPWS_ADD_PTR_ATT(hEndpoint, DPWS_PTR_CALLBACK_EVENT_END,
                 &subscription_end);
```

EVENT SUBSCRIPTION MANAGEMENT

Clients subscribe to events by sending a subscription request to an event source, i.e. a Web Service that exposes events. In accordance with the WS-Eventing specification, any service that features a **`wse:EventSource`** attribute with a **`true`** value in at least one of its portType declarations can act as an event source. A successful subscription request results in a subscription identifier being returned to the client as an endpoint reference. This endpoint reference can subsequently be used by the client to manage the subscription. Subscription creation and management are based on remote service operations invocation, and require an initialized client request context (represented by a pointer to a **`struct dpws`** object) to operate properly.

The DC toolkit API provides the **`dpws_event_subscribe`** and **`dpws_event_subscribe_ex`** functions to create a subscription. Both functions take similar parameters:

- The first parameter is a pointer to the client request context.
- The second parameter is the endpoint reference representing the event source, to which the subscription request is sent. This endpoint reference is a pointer to a **`struct wsa_endpoint_ref`** object, and can either be obtained from service proxies using the discovery mechanisms previously described or be built manually.
- The third parameter is the event sink endpoint reference, to which event notifications will be sent.

- The fourth parameter is the endpoint reference to which subscription end messages will be sent in case of abnormal termination. This parameter is optional (i.e. it may be **`NULL`**). It is often the same as the previous one.
- The fifth parameter is an optional filter that may contain a list of WS-Addressing action URIs representing a subset of the notification operations exposed by the event source.
- The sixth parameter is an in-out parameter that represents the subscription duration. The input value is the requested duration, which may be **`NULL`** to denote an unlimited duration. The output value is the duration granted by the event source, which may be smaller than the requested value.

The two functions differ only by the type of their third and fourth parameters. The **`dpws_event_subscribe`** function uses pointers to **`struct wsa_endpoint_ref`** objects that may be explicitly constructed from service port handle references or built manually (the latter case is useful when the notification endpoint is a third-party remote endpoint). The **`dpws_event_subscribe_ex`** function directly uses service port handle references. The second form is recommended, as it allows the DC runtime environment to dynamically build the appropriate endpoint reference when more than one network interface is in use.

The DC runtime provides additional API functions for cancelling a subscription, renewing a subscription and obtaining a subscription status, including its expiration date. These functions take as parameters a client request context and the endpoint reference returned by the subscription creation function.

IMPLEMENTING THE SERVER ARCHITECTURE

The implementation of the server architecture for a client that receive asynchronous responses and events is strictly the same as the architecture for a device, which is described in the "Implementing the server architecture" section of the device development chapter.

COMPILING AND LINKING

The DC runtime environment is provided as a set of static or shared libraries that must be linked with the generated code for both the implemented Web services and the invoked Web services and with the application code to produce an executable server.

THE DPWS CORE LIBRARIES

Libraries and associated header files are the same as those listed for a simple client development.

GENERATED HANDLER AND APPLICATION FILES

In addition to the files required for a simple client development, the set of files that must be compiled and linked with the DC runtime libraries include:

- For each Web Service called **`ws`**, the generated file called *wsHandlerLib*.c. This file includes both the handler skeleton code (*wsHandler*.c) and the marshalling/unmarshalling code (*ws*C.c).
- For each Web Service called **`ws`**, the file containing the implementation of the handler functions. This file should also include the *ws.nsmap* generated file (note that the *ws.nsmap* files contains global variable declarations and should therefore be included only once in an application).
- The file(s) containing the registry configuration code and the implementation of the server main loop.

Examples of projects for both the Windows and the Linux platforms are provided in the package samples.

Warning: forgetting to include a *ws.nsmap* or *rws.nsmap* file or including it more than once is a usual source of link errors when getting started with the DC toolkit.

ADVANCED FEATURES

CUSTOMIZING CODE GENERATION

One of the challenges to be met when developing Web Services in C is the definition of an appropriate mapping between XML Schema type definitions and C structs. Because the two languages have widely different features, the default mechanisms provided by the code generation tools are not always enough to meet all application requirements. This section provides a set of guidelines that allows the developer to:

- Override the default mapping from XML Schema types to C types.
- Directly modify the gSOAP annotated header file to better control the C code generation.
- Replace for a given C type the generated marshalling/unmarshalling code with hand-written code.

MAPPING XML SCHEMA TYPES TO SPECIFIC C TYPES

The **`wsdl2h`** tool is used to transform WSDL and XML Schema documents into gSOAP annotated header files. The mapping between XML Schema types and C types appearing in the generated annotated header files is controlled by the **`typemap.dat`** configuration file, which is passed as a command-line parameter to the **`wsdl2h`** tool.

> The complete set of **`wsdl2h`** command-line options can be found in §8.2.10 of the gSOAP manual [**gSOAP Guide**].

The **`typemap.dat`** file contains mostly two types of information:

- Namespace prefixes declarations: they can be used to override the default ns1, ns2… namespace prefix generation used by the tool.
- Type definitions: they can be used to associate a specific C type to a given XML Schema type (either predefined or application-defined), identified by its QName in gSOAP notation (**`ns__localName`**).

Namespace prefixes declarations follow the form:

```
prefix = namespace URI string
```

For instance, the following line associates the **`wsa`** prefix to the WS-Addressing URI:

```
wsa = "http://schemas.xmlsoap.org/ws/2004/08/addressing"
```

Type definitions follow the form:

```
type = [declaration] | use [| pointer use]
```

where:

- Type is the name of the XML Schema type, using the gSOAP notation. Note that the prefix used in this type name must have been previously declared in the file, following the above rule.
- Declaration is an optional C type declaration (e.g. a struct, typedef or enum). It may also be an import statement for a gSOAP annotated header file containing the declaration. It may not be specified if the C type is predefined (e.g. **`char*`**) or already defined by another means.
- Use is the form under which the type will be referred to in operation prototypes and struct declarations.

- Pointer use is the form under which pointers to the type will be referred to. This form is used for optional or array fields in struct declarations. When not specified, the default for is the 'use' form followed by '*', unless 'use' is already a pointer.

The following file snippet shows examples of type definitions:

```
xsd  int = | int
xsd  string = | char* | char*
xsd__binary = #import "xop.h" | _xop__Include | _xop__Include *
wsa__EndpointReference = | endpoint_ref | endpoint_ref
```

More details about this feature can be found in the gSOAP User Manual [**gSOAP Guide**], § 8.2.11.

EDITING THE GSOAP ANNOTATED HEADER FILE

The **wsdl2h** tool provides a convenient way to generate a gSOAP annotated header file from a WSDL document. The generated file can then be used as input to the **soapcpp2** tool to generate the C code. However, directly editing the gSOAP annotated header file provides an alternative and more flexible way to specify the mapping between XML Schema types and C types. The syntax for the gSOAP annotated header file is described in details in the gSOAP User Manual [**gSOAP Guide**].

The complete set of **soapcpp2** command-line options can be found in §9.1 of the gSOAP manual [**gSOAP Guide**].

The gSOAP syntax has been extended to support some of the new features required by the WS-Addressing and WS-Eventing specifications.

When editing the gSOAP annotated header file for generating service stubs and skeletons, the following directives must be used for each operation to specify the **wsa:Action** URI associated to the input and (in case of request/response operation) output messages:

```
//gsoap ns service method-request-action: wsdlOp wsaAction
//gsoap ns service method-response-action: wsdlOp wsaAction
```

The following directives must be used to identify a given service as an event source and its output-only operations as event notifications:

```
//gsoap ns service event-source: true
//gsoap ns service method-message-exchange-pattern: wsdlOp output
```

Examples of use of the above directives can be found in gSOAP annotated header files generated by the **wsdl2h** tool.

IMPLEMENTING CUSTOM MARSHALLERS/UNMARSHALLERS

When maximum flexibility is required for mapping XML types to C types, it is possible to deactivate the **soapcpp2** code generation for a given C type and instead provide hand-written serializers and deserializers.

Code generation is deactivated by simply declaring a type as **extern**. For instance:

```
extern typedef struct wsa_endpoint_ref *endpoint_ref;
```

This declares the **endpoint_ref** type as a pointer to a **struct wsa_endpoint_ref**, and tells soapcpp2 not to generate the serializers and deserializers for the **endpoint_ref** type.

The developer must provide the following functions for each type T declared as **extern**:

```
void soap_serialize_T(struct soap *soap, const T *a)
void soap default T(struct soap *soap, T *a)
void soap out T(struct soap *soap, const char *tag, int id, const T *a,
                const char *type)
T *soap in T(struct soap *soap, const char *tag, T *a,
             const char *type)
```

More details about this feature can be found in the gSOAP User Manual [**gSOAP Guide**], § 19.5.

GENERIC INVOCATION

Sometimes, for instance for web services using very simple XML content (or on the opposite fully generic messages), one may want to get rid of code generation using the DPWSCore "generic stub & skeleton" feature.

EPX API

An XML processing API is the key feature to achieve generic invocation. DPWSCore defines one named EPX, and provides a default implementation based on the gSOAP runtime. This is a streaming XML processing API using the pull-parsing approach in a similar way than *Stax* in the Java world.

Warning:

1. This XML parsing API is currently focused and optimized for the XML subset used by SOAP 1.2. This is why for instance, processing instructions are not considered. Entities are also supposed to be resolved by the parser. As a low-level API with no validation, whitespaces are processed like any other character data.
2. EPX provides an experimental limited typed API.

The pull-parsing scheme relies on an *event* stream that is:

- read event-by-event by a parsing application (XML parsing),
- produced by an application using dedicated functions (XML serialization).

Considering the previously mentioned limitations, events related to XML content to be considered are (others, like *comments* are not supported by the current implementation):

- *EPX_EVT_START_DOCUMENT* : XML document prolog event.
- *EPX_EVT_END_DOCUMENT* : XML file end event.
- *EPX_EVT_START_ELEMENT* : Element start-tag event.
- *EPX_EVT_PREFIX_DEFINITION* : Prefix definition event (*xmlns* attributes). This event is optional and produced only if the EPX_OPT_GENERATE_PREFIX_DEFINITIONS option is turned on.
- *EPX_EVT_ATTRIBUTE* : Attribute event.
- *EPX_EVT_END_ELEMENT* : Element end-tag event.
- *EPX_EVT_CHARACTERS* : Characters events that may occur several time consecutively if the parser decides it. Coalescing is not required.

If the order of most events is obvious considering the structure of an XML document, event stream APIs may differ about the order they define for events related with the *open element* tag. The EPX order for these events is:

1. EPX_EVT_START_ELEMENT
2. EPX_EVT_PREFIX_DEFINITION

3. EPX_EVT_ATTRIBUTE (xsi:type, then xsi:nil should be first)

Let's consider the following very simple XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<ns:a xmlns:ns="http://www.example.com/dc/epx" att1="value1">
       <ns:b>Element B</ns:b>
       <c att2="value2">
</ns:a>
```

This document generates the following EPX event stream (considering whitespaces are skipped by a SOAP-oriented XML parser):

```
EPX_EVT_START_DOCUMENT
EPX_EVT_START_ELEMENT
EPX EVT PREFIX DEFINITION
EPX EVT ATTRIBUTE
EPX EVT START ELEMENT
EPX EVT CHARACTERS
EPX_EVT_END_ELEMENT
EPX_EVT_START_ELEMENT
EPX EVT ATTRIBUTE
EPX EVT END ELEMENT
EPX_EVT_END_ELEMENT
EPX_EVT_END_DOCUMENT
```

Of course, every type of event has associated EPX functions to access its specific data as illustrated later.

PARSING

First thing to do for reading an XML stream using EPX is creating an EPX parser:

```
void *pctx;
struct soap soap;
pctx = epx_new_parser(NULL, &soap);
```

Note that no *implementation* parameter is passed to the function (NULL in fact) so the default implementation on gSOAP is selected (the only one currently). The second parameter is implementation-specific and is required by gSOAP to be a runtime structure allowing XML processing. If everything went well, the call should return an opaque parsing context that will be passed to every subsequent EPX call.

Advanced use:
Other initialization APIs, such as **`epx_get_parser_option`**, **`epx_is_parser_option_settable`** allow the user to retrieve the parsing options supported by the implementation. These may be very important especially if the EPX implementation is unknown. For instance, we know here that the gSOAP implementation has the EPX_OPT_GENERATE_PREFIX_DEFINITIONS turned on so that *EPX_EVT_PREFIX_DEFINITION* events will be produced. See the API reference [**DC API**] for more details.

The **`epx_start_parsing`** function allows setting the parsing options and specifying what to parse (the *source*). For the default gSOAP implementation, this parameter is unused since it is basically made for parsing on an open socket.

```
if (epx_start_parsing(pctx, NULL, EPX_OPT_GENERATE_PREFIX_DEFINITIONS))
{
  printf("Could not initialize parsing (%d)\n",
    epx get parser error(pctx));
  exit(1);
}
```

The following code snippet shows how to process the parsing of a document, printing the parsed information on the console.

```
epx_event cev;

for (;;)
{
  switch(cev = epx next(pctx))       // sets current event
  {
    case EPX_EVT_ERROR:
      printf("Parsing error %d.\n", epx get parser error(pctx));
      return;

    case EPX_EVT_START_DOCUMENT:
      printf("Start document event.\n");
      break;

    case EPX EVT IDLE: case EPX EVT END DOCUMENT:
    case EPX_EVT_END_FRAGMENT:
      printf("Parsing ended.\n”);
      break;

    case EPX EVT START ELEMENT:
      printf("Start element event:{%s}%s\n",
          epx_get_ns_uri(pctx), epx_get_lname(pctx));
      break;

    case EPX EVT PREFIX DEFINITION:
      printf("Prefix %s defined for %s\n",
          epx get ns prefix(pctx), epx get ns uri(pctx));
      break;

    case EPX EVT ATTRIBUTE:
      printf("Attribute event:{%s}%s, value='%'\n",
          epx get ns uri(pctx), epx get lname(pctx),
          epx get characters(pctx));
      break;

    case EPX EVT END ELEMENT:
      printf("End element event:{%s}%s\n",
          epx get ns uri(pctx), epx get lname(pctx));
      break;

    case EPX EVT CHARACTERS:
      printf("Characters event: value='%s'\n",
epx get characters(pctx));
      break;
  }
}
epx_delete_parser(pctx);
```

Note the final call to **`epx_delete_parser`** that performs parsing resource cleaning.

SERIALIZATION

Serializer initialization is similar to the one of parser:

```
void *sctx;
struct soap soap;
sctx = epx new serializer(NULL, &soap);
if (epx start document(sctx, NULL, EPX OPT INDENT))
{
  printf(“Could not initialize serialization (%d)\n”,
    epx_get_serializer_error(sctx));
  exit(1);
}
```

Note the required options: we ask the serializer to perform XML indentation for human-readability. Event stream production is then quite simple and consists in calling an EPX function for each event to produce. For instance, to produce the simple XML document described earlier:

```
if (epx start element(sctx, ”http://www.example.com/dc/epx”, “a”)
    || epx_define_prefix(sctx, “ns”, ”http://www.example.com/dc/epx”)
    || epx_add_attribute(sctx, NULL, “att1”, ”value1”)
    || epx_start_element(sctx, ”http://www.example.com/dc/epx”, “b”)
    || epx_put_characters(sctx, “Element B”)
```

```
    || epx_end_element(sctx, "http://www.example.com/dc/epx", "b")
    || epx start element(sctx, NULL, "c")
    || epx add attribute(sctx, NULL, "att2", "value2")
    || epx end element(sctx, NULL, "c")
    || epx end element(sctx, "http://www.example.com/dc/epx", "a")
    || epx_end_document(sctx))
  printf("Serialization error (%d)\n", epx_get_serializer_error(sctx));
```

GENERIC STUBS

To be able to invoke a Web Service without code generation, two functions are provided: **`dpws_send`** for one-way messages, **`dpws_call`** for request-replies. The first two required parameters are the same as for generated stubs:

- A *dpws* runtime structure representing the client request context,
- A WS-Addressing endpoint reference representing the message destination.

Note that unlike generated stubs, the generic **`dpws_call`** function does not support the specification of a *replyTo* endpoint.

The third parameter is the WS-Addressing *action* associated to the operation input message.

Both generic stub functions use then two optional parameters to pass prefix definitions that could be defined at the root of the SOAP message when it is both used in headers and body, avoiding then redundancy.

The rest of the parameters are "body callbacks" for request & response messages and their associated "user data". These callbacks are user-defined functions receiving beside their *user data* an EPX context to produce or read the XML event stream for message bodies. Note that the signature of these callbacks is very simple since their only concern must be limited to XML processing.

Further information

The API reference [**DC API**] will bring you some more details about the feature, but if you want an example of use of the generic stub, please have a look at the *dyndepl_client* module code. Indeed, this is just a helper wrapping the generic stub, avoiding the user to provide, for instance, message action URIs for WS-Management (see "Client-side support" for details).

GENERIC SKELETONS

Writing a skeleton using the *generic skeleton* feature requires a little more than calling APIs and implementing body callbacks. Indeed, the user will have to write the dispatch function that is usually generated and passed to the DPWS Registry using the following pattern:

```
int myservice serve request(struct dpws *dpws)
{
  char* action = dpws->action ? dpws->action : "";
  if (!strcmp(action, "http://www.example.com/op1/request"))
    return dpws_process_request(dpws,
                "http://www.example.com/op1/response",
                "http://www.example.com/op1/fault",
                NULL, 0, op1 cbk
                );
  else if (!strcmp(action, "http://www.example.com/op2/request"))
    return dpws_process_request(dpws,
                "http://www.example.com/op2/response",
                "http://www.example.com/op2/fault",
                NULL, 0, op2 cbk
                );
  else if (!strcmp(action, "http://www.example.com/op3/request"))
    return dpws_process_request(dpws,
                NULL,
```

```
                NULL,
                NULL, 0, op3_cbk
                );
  return dpws_dpws2soap(dpws)->error = SOAP_NO_METHOD;
}
```

The **`dpws_process_request`** API is called with parameters specific to every operation among which:

- *"http://www.example.com/op1/response"* for instance is the action of the response message, showing that this operation is a request-reply contrary to "op3" (NULL supplied),
- "*http://www.example.com/op1/fault*" defines the potential fault action (for *request-reply* operations only),
- Like for the generic stub, a table with prefix definition can be supplied for response messages so that their definitions can occurs on the message root element. Not used in the previous sample,
- A callback for service implementation (*service callback*).

The *service callback* is a little different from the stub one and has the following parameters:

- *dpws* is the runtime structure containing for message processing context,
- An EPX parsing context for incoming message,
- An output parameter allows returning a callback that will produce the response message if any. The signature is then similar to the callback used to produce request messages with the generic stub.
- A second output parameter allows returning the "user data" that will be passed to the response callback once response SOAP header has been processed.

Note that contrary to other callbacks described previously, the *service callback* has a double role that is not limited EPX processing:

1. Request parsing with EPX,
2. Normal service processing (like done for a generated skeleton).

Warning: Since EPX stream production for response is done in a dedicated callback, the implementor should assume that only EPX serialization errors must be returned by this function. Any applicative fault should be detected and raised before, during the service processing (in the *service callback*).

COMPILING AND LINKING

Extra libraries are required for the EPX and generic stub & skeleton features, to be added to the required core stack libraries:

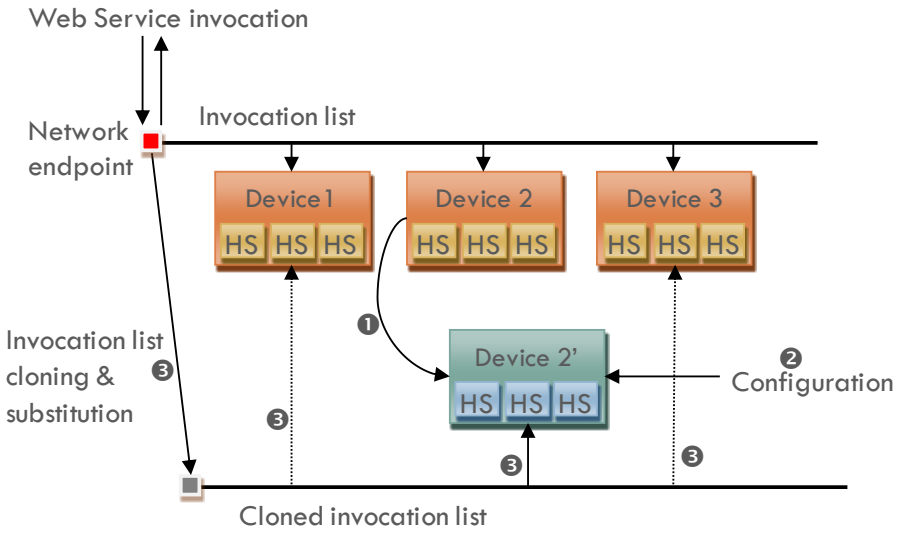
	Shared libraries	*Static libraries*
Root name	dcxml	Xmltools
Linux name	libdcxml.so	libxmltools.a
Windows name	dcxml.dll	xmltootslib.lib

It is also necessary to include the **`dc/dc_Epx.h`** and **`dc/dc_GenericInvocation.h`** headers file in the application code to respectively use the EPX API and the generic stubs and skeletons.

ADVANCED REGISTRY FEATURES

DYNAMIC REGISTRY MODIFICATION API

Device configuration can be updated while server is running and almost without service interruption. DPWSCore uses for this a clone-copy mechanism that allows free device modification while online devices remain read-only.



The figure above shows the steps for "hot configuration":

1. One of the read-only online devices (reachable through the invocation list) is cloned using the **`dpws_clone_device`**. A new handle reference is then returned for the new object.
2. The created object can be modified using **`dpws_set_XXX_att`** functions. Hosted service can even be added to the clone device.
3. When configuration is over, the clone should be put online using **`dpws_replace_device`** that creates a clone invocation list where the clone device has replaced the original device. The clone invocation will then be anchored to the network endpoint that receives web service requests in place of the original invocation list.

ADVANCED CACHE FEATURES

If a discovery listener has been started for receiving Hello & Bye messages, the cache becomes "living" and its content will asynchronously change upon message reception. Some additional features may be used to better control the "living cache" content.

CACHE CONTENT CONTROL

Cache content can be controlled in several ways:

- The number of device proxies stored in the cache can be controlled using a LRU algorithm (Least Recently Used) that keeps in cache proxies that were recently accessed through the user API,
- The device proxies can also be controlled using WS-Discovery criteria. Indeed, the user can set the same kind of filters as used for **`dpws_lookup_ex`** to keep only devices of given *types* or *scopes*.

Warning: All lookup APIs use the discovery cache and no shortcut is currently possible. As a consequence, all lookup API will be constrained by cache settings. For instance, if

10 devices are asked but cache is limited to 5 proxies, the lookup API will only retrieve 5 proxies at most.

Note that device proxies can be explicitly removed from the cache using the **dpws_invalidate_proxy** API, normally when a device has proven to be unreachable.

LIFECYCLE CALLBACKS

In addition, the user can monitor the devices added to or removed from the cache using callbacks, as shown below:

```
DPWS_SET_PTR_ATT(DC_CACHE_HANDLE, DPWS_PTR_CALLBACK_HELLO, hello_cbk);
DPWS_SET_PTR_ATT(DC_CACHE_HANDLE, DPWS_PTR_CALLBACK_BYE, bye_cbk);
```

The two callbacks are registered on the cache pseudo-object. The callback signature gives access to the *dpws* runtime structure representing the server request context and to handle reference of the device proxy being added or removed, thus allowing cache API usage.

Warning: Only "cache" API (generally named **dpws_cache_xxx**), that do not generate network messages, should be used in such callbacks since the supplied *dpws* structure is a server request context, processing a Hello or Bye message at the same time.

For example:

```
void hello_cbk(struct dpws *dpws, short href device)
{
	printf("\n<-- New device on the LAN: %s\n",
	dpws_cache_get_uuid(dpws, href_device));
}
```

XML CONFIGURATION

Toolkit configuration using an XML file is implemented by an extension that uses:

- Regular core stack configuration APIs,
- EPX and other XML utilities,
- A set of XML Schemas that define the configuration file format.

Warning: The XML configuration was introduced along with the Dynamic Deployment feature (described later in this document) and shares some of its features such as the component approach, properties and references. This is also why there is more than one XML schema defining the configuration file format.

Here is an outline of a configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<dcc:Config
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wdp="http://schemas.xmlsoap.org/ws/2006/02/devprof"
xmlns:dd="http://www.soa4d.org/dpwscore/2007/08/dyndepl"
xmlns:dcc="http://www.soa4d.org/dpwscore/2008/10/config"
xmlns:trn="http://www.soa4d.org/DPWS/Samples/DynHome"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<dcc:BootSequence>xs:unsignedInt</dcc:BootSequence>
	<dcc:PreferredLanguage>xs:string</dcc:PreferredLanguage>?
	<dcc:HTTPListener port="xs:int"/>?
	<dcc:Registry>
		<dd:ServiceClass …>
		…
		</dd:ServiceClass>*
		<dcc:Device …>
		…
```

```
            </dcc:Device>*
        </dcc:Registry>
        <dcc:Cache …/>?
        <dcc:SubscriptionManager …/>?
</dcc:Config>
```

At the top level you can find toolkit parameters (*BootSequence, PreferredLanguage*) and elements for main components of the toolkit (*HTTPListener*, *Registry*, *Cache*, *SubscriptionManager*).

For detailed information:
The complete configuration file specification is available in DPWSCore source repository or in distribution packages as annotated XML schemas. Schema annotations should describe every element or attribute or may refer to API reference for configuration parameters semantics.

REGISTRY CONFIGURATION FORMAT

The registry contains two types of root elements: service classes and devices. Other registry objects are either represented as sub-elements (*hosted service*, *service ports*) or not available (*device model* which is just a configuration helper). The *service class* element is defined as follows:

```
<dd:ServiceClass classId="xs:anyURI">
        <dd:Interface name="xs:NCName"? type="xs:QName"/>*
        <dd:Reference name="xs:NCName" type="xs:QName"
mustSupply="xs:boolean":true?/>*
        <dd:Property name="xs:NCName" type="xs:QName"
mustSupply="xs:boolean":true?
multiple="xs:boolean":false?>xs:any</dd:Property>*
        <dd:WSDLInfo location="xs:anyURI" targetNamespace="xs:anyURI"/>*
        <dd:Implementation …>
        …
        </dd:Implementation>
</dd:ServiceClass>
```

You should keep in mind that the *service class* describes the implementation of a kind of *service*. In this context:

- The *Interface* elements define the WSDL port types implemented by this kind of services,
- The *Reference* is a concept brought by the *dynamic deployment* feature that defines WSDL port types used by this service implementation. A runtime API provided by the XML configuration extension will allow the implementation to retrieve WS-Addressing EPRs bound for the reference at run-time for the service instance,
- *Property* is a concept brought by the *dynamic deployment* feature that defines a configuration parameter for this kind of service implementation. Note that the property type is theoretically free, but is currently retrieved as simple strings using the provided runtime API on the service instance. Note also that the element content can be used to provide a default value for the property.
- *WSDLInfo* contains required information to locate the service WSDLs.
- *Implementation* is the fundamental element that should contain the service implementation. The service implementation loading mechanism is made to be extensible: loaders should be provided for every type of implementation element so that the specific loader can use the custom content of the tag to return the required information for the registry. For instance a generic dynamic loader for shared libraries could be written with all library entry points defined in the *Implementation* tag. Currently, no service loader is provided by the toolkit but

writing a static loader, as demonstrated in samples, requires little effort. One static loader must be written for every *service class* as we will see later.

The *device* element contains device DPWS metadata including hosted services:

```
<dcc:Device metadataVersion="xs:unsignedInt">
       <dd:Address>xs:anyURI</dd:Address>?
       <dd:Types>list of xs:QName</dd:Types>?
       <dd:Scopes>list of xs:anyURI</dd:Scopes>?
       <wdp:ThisModel>
              <wdp:Manufacturer
xml:lang="…">xs:string</wdp:Manufacturer>
              <wdp:ManufacturerUrl>xs:anyURI</wdp:ManufacturerUrl>
              <wdp:ModelName xml:lang="…">xs:string</wdp:ModelName>
              <wdp:ModelNumber>xs:string</wdp:ModelNumber>
              <wdp:ModelUrl>xs:anyURI</wdp:ModelUrl>
              <wdp:PresentationUrl>xs:anyURI</wdp:PresentationUrl>
       </wdp:ThisModel>?
       <wdp:ThisDevice>
              <wdp:FriendlyName
xml:lang="…">xs:string</wdp:FriendlyName>
              <wdp:FirmwareVersion>xs:string</wdp:FirmwareVersion>
              <wdp:SerialNumber>xs:string</wdp:SerialNumber>
       </wdp:ThisDevice>?
       <dd:Service serviceId="xs:anyURI">
              <dd:ServiceClass classId="xs:anyURI"/>
              <dd:ServicePort>
                     <wsa:Address>xs:anyURI</wsa:Address>
              </dd:ServicePort>*
              <dd:Reference name="xs:NCName">
                     <wsa:EndpointReference>…</wsa:EndpointReference>
                     |<dd:DiscoveryHints onMultipleMatch="fail|pickOne"
bindingTime="deployment|runtime" onReferenceLost="fail|retry|ignore">
                            <dd:Hint>
                                   <dd:Types>list of
xs:QName</dd:Types>
                                   <dd:Scopes>list of
xs:anyURI</dd:Scopes>
                            </dd:Hint>*
                     </dd:DiscoveryHints>
              </dd:Reference>*
              <dd:PropertyValue
name="xs:NCName">xs:any</dd:PropertyValue>*
       </dd:Service>*
</dcc:Device>
```

Let's focus on some specific parts of the schema (for details, please refer to the XML schemas):

- *Device/@metadataVersion* should be incremented every time some data in the device tree changes. In such situations, the configuration file should be saved.
- *Types* and *Scopes* are device parameters related to [WS-Discovery],
- *ThisModel* and *ThisDevice* are device parameters related to [DPWS],
- *Service/ServiceClass/@classId* must reference the ID of the service class defined earlier in the configuration file that defines the implementation for the service.
- *Service/ServiceClass/Reference* allows binding a reference defined in the service class. Two methods are available: provide a hardcoded EPR or parameters for device discovery using WS-Discovery. The service EPR to invoke will be retrieved using a dedicated runtime API. In "discovery" mode, the service to invoke will be retrieved using the port type specified in the service class reference definition,
- *Service/ServiceClass/PropertyValue* assigns values to the properties defined in the service class.

CACHE CONFIGURATION FORMAT

The discovery cache file configuration is currently limited to the maximum number of devices proxies using the *size* attribute.

```
<dcc:Cache size="xs:unsignedShort"/>
```

SUBSCRIPTION MANAGER CONFIGURATION FORMAT

The maximum number of "running" subscriptions can be set using the *size* attribute and subscription durations can be limited using *maxDuration*.

```
<dcc:SubscriptionManager size="xs:unsignedShort"
                         maxDuration="xs:duration"/>
```

USAGE

PROGRAMMING A STATIC SERVICE CLASS LOADER

The *static service class loader* names a loader that is written to allow simple C code linking, as done for basic device programming. The first element to be provided is a specific tag for the implementation:

```
<ns:Implementation.static.pfx
         xmlns:ns="http://www.example.com/sample"/>
```

The XML configuration feature must be initialized and the class loader registered:

```
int status;
struct qname pfx_iqn = {
       "http://www.example.com/sample", "Implementation.static.pfx"};

if ((status = dpws_config_init()))
   printf("Could not initialize the XML configuration feature (%d)\n",
status);

dpws_register_loader(&pfx_iqn, pfx_load_cbk);
```

The **pfx_load_cbk** (loader callback) function must be implemented by the user using the following pattern:

```
int pfx_load_cbk(short href_sclass, void * psr_ctx, dispatch_cbk *
p_dispatch_cbk, struct scl_callbacks * p_cbks)
{
       if (epx_next(psr_ctx) != EPX_EVT_END_ELEMENT
       || QNAME NOT EQUALS WILDCARD(
              epx_get_ns_uri(psr_ctx), epx_get_lname(psr_ctx),
              "http://www.example.com/sample",
              "Implementation.static.pfx")
              )
              return DPWS_ERR_INCORRECT_IMPL_TAG;

       *p_dispatch_cbk = pfx_serve_request;
       p_cbks->new_service = pfx_new_service;
       p_cbks->free_service = pfx_free_service;
       p_cbks->serialize_impl = pfx_serialize_impl;
       return DPWS_OK;
}
```

As shown in the last code snippet, there are two things to do in a loader callback:

- Do the parsing of the implementation tag. Note that when the function is called the current EPX event is the implementation tag "start element". For a static loader, note that parsing is very simple…
- Return the callbacks for the service class:
  - The *dispatch function* is the one usually generated and used to route SOAP message that is registered on the service class,
  - *service creation*: called every time a service is instantiated for the service class. This is generally used to initialize instance runtime data.
  - *service deletion*: called every time a service instance of this class is freed. This is generally used to free instance runtime data.

- *implementation tag serialization*: The implementation tag is specific to the loader, this is why it is called to perform on-demand EPX serialization.

For further information, please have a look at samples and API reference.

FILE LOADING & BACKUP

The XML file persistency is performed through the EPX API implemented on the gSOAP runtime which uses a streaming interface bound by default on BSD sockets. In order to replace the gSOAP socket stream by a file persistence layer, several additional components are involved, as shown below:

FileX is an adapter that plugs SUN on the gSOAP stream interface. SUN is a DPWSCore persistency streaming interface. This technology stack results in the following kind of code for XML file configuration operations:

```
void * sun_stream;
sun_stream = sun_read_init(NULL, DPWS_CONFIG_DEFAULT_ID);
if (!sun_stream)
       printf("Could not open configuration file\n");
status = dpws_sun_load_config(&conf_dpws, sun_stream);
if (status)
       printf("Could not load DPWS configuration (%d)\n", status);

sun_stream = sun_write_init(NULL, DPWS_CONFIG_DEFAULT_ID, NULL);
if (!sun_stream)
       printf("Could not open configuration file for saving\n");
status = dpws_sun_save_config(&conf_dpws, sun_stream, DC_TRUE);
if (status)
       printf("Could not save DPWS registry (%d)\n", status);
```

In this code snippet, the configuration file is first loaded and immediately saved, in fact for boot sequence management. Note that:

- The configuration file uses the default name (*dpwscore.xml*) and the working directory for both loading and backup,
- Specific SUN API is used for stream opening but stream closure is performed by the **`dpws_sun_load_config`** and **`dpws_sun_save_config`**. These API also encapsulate the FileX plug-in in gSOAP.
- Note the last parameter of **`dpws_sun_save_config`** that is used to increment the boot sequence before saving configuration (which must be done once by program execution).

CLEANUP

The XML configuration/dynamic deployment feature should be cleaned when exiting the program or when not required anymore using:

```
dpws_config_shutdown();
```

COMPILING AND LINKING

In addition to the core stack libraries, the XML configuration feature requires:

	Shared libraries	*Static libraries*
Root name	dcxml, dcxmlconf	xmltools, xmlconf
Linux name	libdcxml.so, libdcxmlconf.so	libxmltools.a, libxmlconf.a
Windows name	dcxml.dll, dcxmlconf.dll	xmltootslib.lib, xmlconflib.lib

It is also necessary to include the `dc/dc_XMLConfiguration.h` and `dc/dc_Sun.h` headers file in the application code.

DYNAMIC DEPLOYMENT

Most dynamic deployment features are introduced in the XML configuration feature, especially through the XML format specification for *service classes* and *devices*. But "dynamic deployment" meaning remote operations, a subset of [WS-Management] is used for this, especially [WS-Transfer] and associated resource endpoint references. The [WS-Management] resources considered are:

- Service classes,
- Devices,
- Hosted services, which are considered both as a first-class [WS-Management] resource and a sub-element of their device resource.

The following table shows what [WS-Transfer] operations are allowed for each resource type or fragment:

Operation	*Service class resource*	*Device resource*	*Service resource*	*Types fragment*	*Scopes fragment*	*ThisModel fragment*	*ThisDevice fragment*
Get	X	X	X	X	X	X	X
Create	X	X	X				
Put		X		X	X	X	X
Delete	X	X	X				

- *Create* returns resource endpoint references that will be used for subsequent operations,
- *Get* is used for reading an existing resource or fragment,
- *Put* allows to overwrite an existing resource,
- *Delete* destroys a resource provided it is not used anymore. For instance, a *service class* cannot be deleted is it is referenced by a hosted service.

SERVER-SIDE SUPPORT

In order to use the "dynamic deployment" feature, not only the appropriate runtime version must be selected, but XML configuration must be used to configure the dynamic deployment service:

```
<dd:ServiceClass
xmlns:wst="http://schemas.xmlsoap.org/ws/2004/09/transfer"
classId="http://www.soa4d.org/dpwscore/2007/09/dyndepl/wsman">
       <dd:Interface type="wst:Resource" />
       <dd:Interface type="wst:ResourceFactory" />
       <dd:Implementation.static.dyndepl />
</dd:ServiceClass>
<dcc:Device …>
       <dd:Types>…</dd:Types>
       <wdp:ThisModel>
       …
       </wdp:ThisModel>
       <wdp:ThisDevice>
       …
       </wdp:ThisDevice>
       <dd:Service
serviceId="http://www.soa4d.org/dpwscore/2007/09/dyndepl/wsman">
```

```
                <dd:ServiceClass
classId="http://www.soa4d.org/dpwscore/2007/09/dyndepl/wsman" />
                <dd:ServicePort>
                        <wsa:Address>…</wsa:Address>
                </dd:ServicePort>
        </dd:Service>
        …
</dcc:Device>
```

Note that the template for the service class is immutable contrary to the device fragment that must contain a hosted service implementing the previous service class, the rest being totally free (the device may for instance host other services unrelated with dynamic deployment). Once declared, some initialization code must be added to handle the *implementation.static.dyndepl* tag:

```
dpws_register_dyndepl_loader();
```

In order to make configuration modifications through the dynamic deployment service persistent, some code must be added:

```
dpws_register_config_cbk(save_cbk);
```

This registers the following function called whenever the dynamic deployment service modifies the configuration.

```
void save cbk()
{
        int status;
        void * sun_stream;

        sun stream = sun write init(NULL, DPWS CONFIG DEFAULT ID, NULL);
        if (!sun stream)
                fprintf(stderr, "Could not open configuration file for
saving.\n");

        status = dpws sun save config(&conf dpws, sun stream, DC FALSE);
        if (status)
                fprintf(stderr, "Could not save DPWS registry
(err:%d).\n", status);
}
```

You can notice that the configuration backup is done here without incrementing the *boot sequence*.

Warning: The dynamic deployment feature is incompatible with manual configuration since the dedicated service considers it is the only one to modify registry contents. Performing concurrent configuration operations may lead to unexpected behavior.

COMPILING AND LINKING

In addition to the core stack libraries, the server dynamic deployment feature requires:

	Shared libraries	*Static libraries*
Root name	dcxml, dcwsman, dcdyndepl	xmltools, xmlconf,wsman,dyndepl
Linux name	libdcxml.so, libdcwsman.so, libdcdyndepl.so	libxmltools.a, libxmlconf.a, libwsman.a, libdyndepl.a
Windows name	dcxml.dll, dcwsman.dll, dcdyndepl.dll	xmltoolslib.lib, xmlconflib.lib, wsmanlib.lib, dyndepllib.lib

It is also necessary to include the **`dc/dc_XMLConfiguration.h`**, **`dc/dc_Sun.h`** and **`dc/dc_DynDepl.h`** headers file in the application code.

CLIENT-SIDE SUPPORT

A helper is supplied for dynamic deployment client development, which simply encapsulates the generic stub, providing one function for every {*resource*/*fragment*, *operation*} couple. This allows:

- Limiting invocations to the one authorized (as specified earlier in the table),
- Hiding most WS-Management stuff (headers, actions).

For instance, the following function creates a hosted service on a remote device:

```
int dyndepl_create_service(
        struct dpws *dpws,
        short href_dyndepl_service_proxy,
        char * device_uuid,
        serialize_cbk request_cbk,
        parser_cbk response_cbk,
        void * user_data
);
```

Note about the parameters:

- Every API expects the WS-Management service proxy to be supplied (*href_dyndepl_service_proxy*) which means that the user must perform the discovery of the service first using standard toolkit features,
- The service having a composite ID (selector using WS-Management terminology) made of the hosting device UUID and the service ID, the first must be supplied when the second will be part of the sent XML fragment that describes the new service,
- EPX callbacks will be required for *create* & *put* request bodies to produce the event stream for resource description,
- EPX callbacks will be required for handling *get*, *create* & *put* response bodies. Note that both *create* & *put* may return, like *get,* the resource description if it differs from the request. The *create* response will also contain the EPR that will be usable for subsequent WS-Management operations (containing especially WS-Management reference parameters):

```
<s:Body ...>
        <wxf:ResourceCreated>endpoint-reference</wxf:ResourceCreated>
        xs:any
</s:Body>
```

The *create* response *ResourceCreated* tag is not encapsulated in the APIs so its parsing should be done using the user EPX callback.

Other useful information:
Note that some other helpers allow to stream EPX into or from a file (see **`xml_file_serialize`** and **`xml_file_parse`** reference) which may be useful when processing message bodies.

COMPILING AND LINKING

In addition to the core stack libraries, the client dynamic deploiement feature requires:

	Shared libraries	*Static libraries*
Root name	dcxml, ddclient	xmltools, ddclient
Linux name	libdcxml.so, libddclient.so	libxmltools.a, libdyndepl_client.a
Windows name	dcxml.dll, ddclient.dll	xmltoolslib.lib, ddclientlib.lib

It is also necessary to include the **dc/dc_DynDeplClient.h** header file in the application code.

MULTIPLE NETWORK INTERFACES AND IP PROTOCOLS

The DPWSCore toolkit is able to manage multiple network interfaces using either IPv4 or IPv6. This means especially for WS-Discovery that:

- multicast packets may have to be sent on every interface and for every version of the protocol.
- multicast packets server reception will detect the reception interface. This allows refining responses so that only addresses for the reception interface can be published.

Note :
The reception interface detection may require only one socket or one socket by interface depending on the platform capabilities. However, IPv4 and IPv6 will always use distinct sockets.

For HTTP messages, such considerations do not exist except that a socket is created for each version of the IP protocol.

Practically, the user only has to initialize the stack using one of the **dpws_init** functions, the rest being managed by the DPWS stack. Let's consider the extensive version of the API:

```
int dpws_init_ex(
        dc_ip_filter_t * selector,
        const char * hostname,
        int versions
);
```

The following parameters are not strictly related to network interfaces:

- *hostname*, allows to force using a DNS name instead of IP addresses in published URLs,
- *versions*, is a bit flag that allow to select the [DPWS] version supported.

The only currently supported version can be selected using DPWS_DEFAULT_VERSION (or DPWS10_VERSION). For your information, DPWS11_VERSION is experimental and corresponds to the ongoing normalization process at OASIS.

The selection of the protocol & network interfaces is made through the *selector* parameter, which is in fact an IP address selector since:

- network interfaces may have several IP addresses,
- only IP addresses matter on most socket operations, because of IP protocol, but also in [DPWS] messages, network interfaces being local technical routing entities.

The chosen approach is to retain addresses that match the supplied filter. The IP address filter format is:

```
struct dc_ip_filter {
        dc_netif_filter_t * netif;
        DC_BOOL include_loopback;
        int proto;
        int nb_addrs;
        char ** addrs;
};
```

- *netif* is a network filter that only accepts IP address attached to a given network interface based on either:
  - the MAC address (recommended),
  - the index, which is defined for IPv6 and may not be significant on IPv4-only platform,
  - the name which is dependent on the platform, usable for instance on Linux (*eth0*, …) but not obvious on Windows.
- *include_loopback* is an additional filter that will select or not loopback addresses (*127.0.0.1* et *::1*),
- *proto* selects IP addresses based on the protocol version,
- if the previous criteria are inadequate, a list of IP addresses can be specified using the *nb_addrs* and *addrs* parameters.

ADVANCED STACK CUSTOMIZATION

The DC toolkit provides several advanced features available through customization of the Web Services stack. These features include:

- Operation timeouts: this feature allows a client or server to limit the time during which a blocking operation may hang.
- Connection keep-alive: this feature allows a client to reuse a connection when sending several messages to a server. Connection keep-alive must be supported both on the client and the server to work properly.
- HTTP chunked transfer encoding: this feature allows a HTTP client or server to stream a request or a response without knowing its precise length beforehand. It is only useful for keep-alive connections.
- MTOM support: this feature allows a client or server to use the standard Message Transmission Optimization Mechanism to transfer large binary content.

OPERATION TIMEOUTS

Web Services use HTTP as their main transport layer, itself built on top of TCP. This means that operations such as connecting to a server, sending or receiving data may take time and are by default subject to the standard TCP delays. The DC runtime provides timeouts to control those delays. The following four fields provided by the `struct soap` defined inside a `struct dpws` object may be used:

- `dpws.soap.accept_timeout`: this field can be used on the server side to limit the time during which the server will wait for an incoming connection request. When used, the `dpws_accept` or `dpws_accept_thr` functions will return with a timeout error upon expiration of the delay.
- `dpws.soap.recv_timeout`: this field can be used on both the server and the client side to limit the time during which the application will wait for incoming data on an existing connection.
- `dpws.soap.send_timeout`: this field can be used on both the server and the client side to limit the time during which the application will wait for outgoing data to be sent on an existing connection.
- `dpws.soap.connect_timeout`: this field can be used on the client side to limit the time during which the application will wait for a new connection to be established with the requested server.

Timeout values are integer values which can be either positive, in which case they are interpreted as the duration of the timeout in seconds, or negative, in which case they are interpreted as the opposite of the duration of the timeout in microseconds. Null values are interpreted as infinite timeouts. For instance:

```
dpws.soap.connect_timeout = 3;     // 3 seconds
dpws.soap.recv_timeout = -200000; // 200 milliseconds
dpws.soap.send_timeout = 0;        // No timeout
```

On the server side, timeouts can be set on the **struct dpws** object initialized with **dpws_server_init**, in which case they are automatically copied to the request-specific object when it is initialized (in the **dpws_accept_thr** function), or, in the case of receive or send timeouts, directly on the request-specific object before calling **dpws_serve**.

On the client side, timeouts must be set on the **struct dpws** object representing the client request context before a generated or generic stub is called. Note that the timeouts are not reset by a call to the **dpws_end** function.

CONNECTION KEEP-ALIVE

Connection keep-alive is a standard feature of HTTP 1.1, and as such may be expected by most HTTP clients. Connection keep-alive is however subject to negotiation, so a server is not required to accept client requests for keep-alive connections.

The use of connection keep-alive is strongly recommended when a client expects to send a large number of messages to the same server, as it improves performances and reduces the number of sockets that are left in **TIME_WAIT** state. It can also be useful when a device sends high-frequency event notifications to one or several subscribers (note that this use case is still considered as client-side connection keep-alive, even if it occurs on a device).

SERVER-SIDE

On the server side, connection keep-alive works by instructing the **dpws_serve** function not to return after processing a request, but rather to keep waiting for new requests on the same connection until a given number of requests have been processed or an error occurred. The **dpws_serve** function may therefore not return for a considerable amount of time when this feature is used.

The connection keep-alive feature is activated on the server side as follows:

```
dpws.soap.imode |= SOAP_IO_KEEPALIVE;
dpws.soap.omode |= SOAP_IO_KEEPALIVE;
dpws.soap.max_keep_alive = 100;
```

In the above example the **max_keep_alive** field is used to control the maximum number of requests (in this case 100) that will be processed before the **dpws_serve** returns. The above values may be set either on the **struct dpws** object initialized with **dpws_server_init**, in which case they are automatically copied to the request-specific object when it is initialized (in the **dpws_accept_thr** function), or directly on the request-specific object before calling **dpws_serve**.

Warning:
The connection keep-alive feature should always be used on the server in conjunction with a multithreaded architecture, as otherwise the server would not be able to respond to UDP messages or new HTTP connection requests.
It is also advisable to use a receive timeout in conjunction with server-side connection keep-alive, to allow the **dpws_serve** function to return after the specified inactivity delay, and thus avoid being blocked by an inactive client.

CLIENT-SIDE

On the client side, connection keep-alive is based on the use of a connection pool. The connection pool keeps open a number of TCP connections for reuse. A maximum number of active connections, along with a maximum idle time before a connection is closed, can be used to configure the connection pool. The DC toolkit provides three functions to manage and use the connection pool:

- **dpws_use_connection_pool**: this function sets a flag to configure a client request context for using or not using a connection pool. When the flag is on, all subsequent requests using the configured request context will use the connection pool, until the flag is turned off by a second call to this function.
- **dpws_init_connection_pool**: this function initializes the connection pool with the specified maximum number of connections and maximum idle time. The use of this function is optional, as the pool is initialized with 10 connections and a 10 second maximum idle time by default.
- **dpws_shutdown_connection_pool**: this function closes all connections in the pool.

COMPILING AND LINKING

Connection keep-alive does not require specific libraries to be used, beyond the standard DC libraries. It is however necessary to include the **dc/dc_ConnPool.h** header file in the client code to use the connection pool API.

HTTP CHUNKED MODE

The chunked transfer encoding is a required feature of HTTP 1.1. Unlike connection keep-alive, it is not subject to negotiation between client and server. Chunked mode is useful when using connection keep-alive as an alternative to the explicit computation of a request or response content length: when using chunked mode, the HTTP message content is sent in chunks prefixed with their length. This allows the sender to use buffers of reasonable size to stream large content without having to compute its length beforehand.

Chunked mode is activated as follows:

```
dpws.soap.omode |= SOAP_IO_CHUNK;
```

On the server side, this flag may be set either on the **struct dpws** object initialized with **dpws_server_init**, in which case it is automatically copied to the request-specific object when it is initialized (in the **dpws_accept_thr** function), or directly on the request-specific object before calling **dpws_serve**. The flag controls the transfer encoding of the response.

On the client side, this flag may be set on the **struct dpws** object representing the client request context before a generated or generic stub is called. The flag controls the transfer encoding of the request and is not reset after a call to **dpws_end**.

MTOM SUPPORT

MTOM is a mechanism used to transform a SOAP message containing elements holding base64-encoded binary data, into a MIME Multipart/Related message, in which the first part contains the transformed SOAP message where the binary contents have been replaced by **xop:include** placeholders, and the following parts contain the removed binary contents.

The DC toolkit provides support for MTOM, through two complementary mechanisms:

- The code generation must be configured to produce MTOM-compatible C structs and marshalling/unmarshalling code.
- The DC runtime must be configured to send and receive SOAP messages using MTOM.

The first step can be performed by customizing the **wsdl2h** code generation tool as explained in a previous section. The following line must be added to the type definitions section of the **typemap.dat** file:

```
xsd__binary = #import "xop.h" | _xop__Include | _xop__Include *
```

This line instructs the **wsdl2h** tool to generate a **_xop__Include** type everywhere a XSD binary type is used in XML elements.

The second step can be performed by setting the MTOM support flag in the request context used to send and receive the MTOM messages:

```
dpws.soap.imode |= SOAP_ENC_MTOM;
dpws.soap.omode |= SOAP_ENC_MTOM;
```

On the server side, this flag may be set either on the **struct dpws** object initialized with **dpws_server_init**, in which case it is automatically copied to the request-specific object when it is initialized (in the **dpws_accept_thr** function), or directly on the request-specific object before calling **dpws_serve**.

On the client side, this flag may be set on the **struct dpws** object representing the client request context before a generated or generic stub is called.

More details on the support of MTOM can be found in the gSOAP User Manual, § 16.

BASIC PROFILE 1.1 SUPPORT

The DPWS specification requires hosted services to use SOAP 1.2 and WS-Addressing. However, in some cases, to increase interoperability with existing Web services, it is useful to support the Basic Profile 1.1 (an ISO standard), both on the server and on the client side. Basic Profile 1.1 requires the use of SOAP 1.1 and does not support WS-Addressing.

In order to support services using a dual DPWS/BP 1.1 mode, both the code generation and the DC runtime must be configured in a specific way. In addition, the WSDL document describing the services must be compliant with the BP 1.1 requirements.

The **wsdl2h** code generator relies on the SOAP binding namespace present in the WSDL document to decide which version of SOAP to support in the generated code:

- When the SOAP 1.2 binding namespace is *declared* in a WSDL document, the use of the SOAP 1.2 binding is forced (even if the actual binding uses the SOAP 1.1 binding).
- When only the SOAP 1.1 binding namespace is declared and used in the WSDL document, the generated gSOAP annotated header file will allow the setup of servers that support both DPWS and BP 1.1 modes, and the setup of clients running in BP 1.1 mode on top of the DPWS stack.

The **soapcpp2** code generator must be executed on the resulting gSOAP annotated header file to produce the generated C files:

- If the '**-2**' option is used on the command line, the generated C code will only support SOAP 1.2, regardless of the input.

- If the '-2' is not used, and the SOAP1.1 binding has been used in the WSDL file, then the generated C code will allow the setup of servers that support both DPWS and BP 1.1 modes, and the setup of clients running in BP 1.1 mode on top of the DPWS stack.

On the server side, in order to be compatible with both BP 1.1 and DPWS clients, it is also necessary to set the **DPWS_BOOL_BP1_1_COMPATIBILITY** attribute on the **DC_TOOLKIT_HANDLE** pseudo-object. This is required to turn off the mandatory use of WS-Addressing headers.

```
DPWS_SET_BOOL_ATT(DC_TOOLKIT_HANDLE, DPWS_BOOL_BP1_1_COMPATIBILITY,
                  DC_TRUE);
```

On the client side, it is also necessary to clear a flag to turn off the sending of WS-Addressing headers, as follows:

```
dpws.soap.imode &= ~DPWS_HEADERS;
dpws.soap.omode &= ~DPWS_HEADERS;
```

This flag must be cleared on the **struct dpws** object representing the client request context before a generated or generic stub is called. Note that this flag must be set back on the same object before calling DC toolkit built-in functions, as WS-Discovery, WS-Transfer and WS-Eventing require the use of WS-Addressing headers.

```
dpws.soap.imode |= DPWS_HEADERS;
dpws.soap.omode |= DPWS_HEADERS;
```

HTTP GET SUPPORT

The DC toolkit provides limited support for handling HTTP GET requests. Although the DC runtime normally only handles HTTP POST requests, it is possible to define a callback function that will be called when a HTTP GET request is received. This callback must be set as follows:

```
dpws.soap.fget = get_callback;
```

On the server side, this callback should be set on the **struct dpws** object initialized with **dpws_server_init**.

The callback function takes a pointer to a **struct soap** object as parameter. This object is the one embedded in the request context used to process the incoming HTTP request. A simple implementation of a HTTP GET callback function is shown below:

```
int get_callback(struct soap* soap)
{
    char* context_path = strchr(soap->path, '/');
    // The struct dpws object associated to the request
    struct dpws * dpws = dpws_soap2dpws(soap);
    // HTTP response header with text/html
    dpws_response(dpws, SOAP_HTML);
    soap_send(soap, "<html><head><title>");
    soap_send(soap, "DC Hello page");
    soap_send(soap, "</title></head><body>Hello world</body></html>");
    return DPWS_OK;
}
```

The **dpws_response** and **soap_send** functions are part of the DC runtime internal API used by generated code. As such, they are not documented in the public API.

A more complex example of HTTP GET callback function is available in the DC toolkit source code (**dpws_http_get** in **dcDPWS_Dpws.c**). Note however that that implementation uses a lot of internal functions and should not directly be copied.

ADVANCED EVENTING FEATURES

The DC toolkit provides several mechanisms to manage subscriptions on the server side:

- Configuration of the maximum number of subscriptions and of their maximum duration.
- Detection of event notification failures and removal of "dead" subscriptions.

SUBSCRIPTION MANAGEMENT CONFIGURATION

The DC toolkit provides a few attributes that can be used to globally control the number and duration of subscriptions:

- **`DPWS_INT_MAX_SUBSC_NB`**: this attribute represents the maximum number of subscriptions that can be globally managed at any given time. It defaults to a very large value. When the number of active subscriptions exceeds this value, new subscription requests are rejected until older subscriptions are cancelled.
- **`DPWS_STR_MAX_SUBSC_DURATION`**: the maximum allowed duration for subscriptions, specified as a string compliant with the XML Schema duration type. It defaults to **`"P1D"`** (one day). This value is internally converted into a number of seconds.
- **`DPWS_INT_MAX_SUBSC_DURATION`**: the maximum allowed duration for subscriptions, specified in seconds. It defaults to 86400 seconds (one day). Subscription requests that use expiration durations greater than the maximum allowed duration are only granted this maximum duration.

These three attributes can be set by the generic DC attribute setters applied to the **`DC_SUBSC_MANAGER_HANDLE`** pseudo-object, as shown below:

```
DPWS_SET_INT_ATT(DC_SUBSC_MANAGER_HANDLE, DPWS_INT_MAX_SUBSC_NB, 100);
DPWS_SET_INT_ATT(DC_SUBSC_MANAGER_HANDLE, DPWS_INT_MAX_SUBSC_DURATION,
                 3600);
```

MONITORING EVENT DELIVERY FAILURES

A typical problem when using event notifications occurs when subscribers disappear without cancelling their subscriptions, due either to bad programming practices or to network or system failures. This leads to event notification failures, which can block the sending process for quite a long time: typical TCP connection timeouts can be up to two minutes and TCP detection of a missing peer when the connection is already established can take up to ten minutes. In addition, due to the use of a sequential algorithm to send event notifications to subscribers, failure to reach one subscriber will delay the delivery to all the following ones.

Several mechanisms are provided by the DC toolkit to alleviate this problem:

- Connect, send and receive timeouts can be set on the client request context before sending event notifications, thus limiting the delays induced by notification failures.
- A callback function can be set on the client request context before sending event notifications and be called by the runtime when a notification failure occurs.

The callback function is set on the client request context as follows:

```
dpws.fnotification_failure = notif_failed;
```

The callback takes as parameters the client request context, the handle reference for the event source and the endpoint reference responsible for the delivery failure. A typical implementation of such a callback function is shown below:

```
void notif_failed(struct dpws * dpws, short event_source,
```

```
                    struct wsa_endpoint_ref * sink)
{
    // App specific code used to check if the subscription should be
    // really removed (e.g. a counter of failures)
    dpws_remove_subscriber(event_source, sink);
}
```

The **dpws_remove_subscriber** function is an API function that can be used by the server to remove all subscriptions for a given event source and endpoint reference (it is assumed that if an endpoint reference is not reachable for one subscription, it will not be reachable for others).

External Web server integration

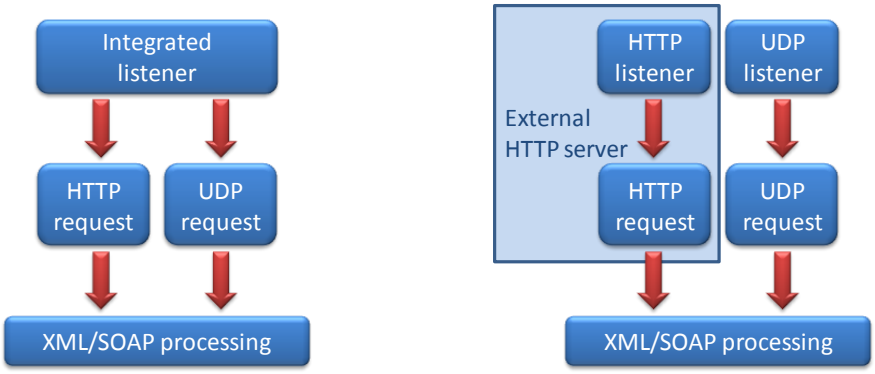
Although the DC toolkit provides an internal Web server, it sometimes required to integrate the DC Web Services stack with an existing Web server. This integration has two main impacts:

- The configuration of the device server architecture is modified, to allow the setup of the external Web server in parallel with the DC discovery listener.
- The external Web server request processing must be implemented in a way that allow incoming HTTP requests containing SOAP messages to be processed by the DC runtime.

Server configuration

The default server configuration of the DC runtime uses an integrated listener that can multiplex both incoming UDP and HTTP requests. Only a single thread is required to operate this listener, thus giving complete control of the application thread architecture to the developer.

When integrating an external Web server, it is usually not possible to have a single listener for both HTTP and UDP requests, as Web servers provide their own listener mechanisms that are generally not extensible with additional input channels.



It is therefore necessary to use a multithreaded architecture to configure and start the DC runtime environment with an external Web server:

- The DC server is configured to only start the UDP listener. This is achieved by calling the **dpws_server_init_ex** function with **DC_LISTENER_UDP** as *listeners* parameter. The usual server loop based on the **dpws_accept** and **dpws_serve** functions can then be used to process incoming UDP requests. This loop requires at least one thread to be executed.
- The external Web server must be configured and started using its specific configuration API. Depending on its implementation, the Web server execution loop may require one or several threads.

Even when using an external Web server, it remains necessary to configure the HTTP server port, and optionally its address, in the DC registry, as the server HTTP URL is used to generate the devices and hosted services transport addresses that appear in the DPWS metadata.

HTTP REQUEST PROCESSING

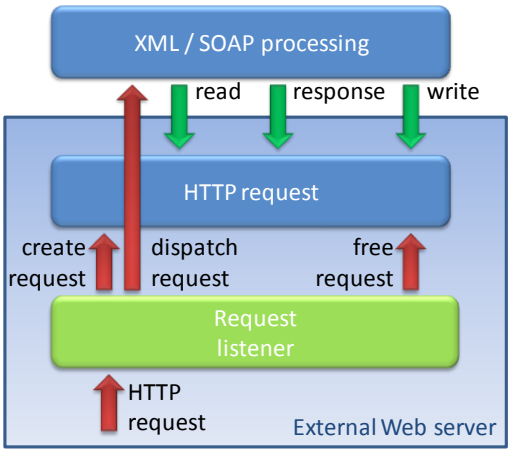
As a prerequisite, the external Web server must provide a mechanism allowing the developer to gain control of the request processing, including an API that provides access to the HTTP request and response headers and body.

The DC toolkit provides an API function (**`dpws_dispatch_request`**) that can be called to process (dispatch) a single SOAP request. In this approach:

- The Web server is responsible for all the HTTP request management: connection keep-alive, transfer encoding (chunked mode) and message buffering and delimitation.
- The DC SOAP engine controls the flow of data, i.e. pulls request data from and pushes response data to the HTTP layer. Access to the HTTP layer is done through a set of callback functions, dependant on the external Web server API, which must be provided by the application developer.

The following figure shows the three callback functions that must be provided:

- Read: this callback is used by the DC SOAP engine to read request data.
- Response: this callback is used by the DC SOAP engine to notify the HTTP layer that it is about to start sending the response.
- Write: this callback is used by the DC SOAP engine to write response data.



The **`dpws_dispatch_request`** function takes the following parameters:

- *dpws*: A pointer to a **`struct dpws`** object representing the server request context. This object must be allocated and initialized (using the dpws_client_init function) by the application developer for each request.
- *transport_data*: An opaque pointer (**`void *`**) representing the Web server-specific HTTP request. This pointer will be passed back as context to the three callback functions.
- *fns*: a pointer to a struct containing the three callback functions.
- *host:* the HTTP server host name, as found in the "Host" HTTP header.
- *path:* the request URL, as found in the HTTP request line.

- *mtype:* the request content type, built from the "Content-Type" HTTP header. It is represented by a pointer to a **`struct media_type`** object, which must be initialized and filled by the application developer.
- *action:* the contents of the optional "SOAPAction" HTTP header. It is currently not used.
- *needs_length:* a Boolean indicating to the SOAP processor that it must provide the response length.

The function returns 0 on success, an error code otherwise. Errors can include recoverable errors, such as a SOAP processing error that returns a SOAP fault to the client, and non-recoverable errors, such as transport errors.

The three callback functions take the same two first parameters:

- A pointer to a **`struct dpws`** object representing the server request context, as passed to the **`dpws_dispatch_request`** function.
- An opaque pointer (**`void *`**) representing the Web server-specific HTTP request, as passed to the **`dpws_dispatch_request`** function.

The *read* and *write* callback functions take as additional parameters a character buffer and its length, to be used for receiving or sending the request and response content. They return the number of characters received or sent, or **`-1`** when an error occurred.

The response callback function takes as additional parameters:

- *status*: an integer representing the SOAP response status, i.e. one of DC_SOAP_RESPONSE, DC_SOAP_EMPTY_RESPONSE, DC_SOAP_SENDER_FAULT or DC_SOAP_RECEIVER_FAULT. This status should be used to generate the appropriate HTTP status code (200, 202, 400 and 500 respectively).
- *mtype*: a pointer to a **`struct media_type`** object, representing the content type of the response. The content of this object can be used by the developer to build the "Content-Type" HTTP header of the response.
- *len*: the length of the response, or 0 if unknown. It should have a positive value if the *needs_length* parameter has been specified in the **`dpws_dispatch_request`** function. This value can be used to build the "Content-Length" HTTP header of the response.

COMPILING AND LINKING

The external Web server integration feature does not require specific libraries to be used, beyond the standard DC libraries. It is however necessary to include the **`dc/dc_DpwsRequest.h`** header file in the code providing the "glue" between the Web server and the DC runtime.

APPENDICES

ERROR MANAGEMENT

Most DPWSCore API functions return an error code or make it accessible through a **`dpws_get_error`** call on the *dpws* runtime structure. All components use the same error space with non-overlapping ranges, for instance:

- gSOAP use positive error codes,
- API errors use negative values (from 0 to -200),
- XML configuration / dynamic deployment errors use another range of negative values (from -200 to -300),
- XML parsing error use positive errors that do not collide with gSOAP error codes (from 100 to 200).

An error message can be retrieved using **`dpws_get_error_msg`** especially in case of gSOAP error since most API errors don't have built-in messages.

Other error code, for built-in WS-Addressing & WS-Eventing faults (negative values < -1000) may be raised automatically by the stack and won't be seen as API error codes but as faults (SOAP_FAULT) by a client. However, a user service implementation may use these codes to return automatically a built-in fault (e.g. a WS-Management implementation).