Lehrstuhl für Verteilte Informationssysteme
Fakultät für Informatik und Mathematik
Universität Passau

Dissertation

Resource Optimization of SOA-Technologies in Embedded Networks

Diplom-Informatiker Univ.
Sebastian Käbisch

Eingereicht am 22. November 2013 an der Fakultät für Informatik und Mathematik der Universität Passau.

Abstract

Embedded networks are fundamental infrastructures of many different kinds of domains, such as home or industrial automation, the automotive industry, and future smart grids. Yet they can be very heterogeneous, containing wired and wireless nodes with different kinds of resources and service capabilities, such as sensing, acting, and processing. Driven by new opportunities and business models, embedded networks will play an ever more important role in the future, interconnecting more and more devices, even from other network domains. Realizing applications for such types of networks, however, is a highly challenging task, since various aspects have to be considered, including communication between a diverse assortment of resource-constrained nodes, such as microcontrollers, as well as flexible node infrastructure. Service Oriented Architecture (SOA) with Web services would perfectly meet these unique characteristics of embedded networks and ease the development of applications. Standardized Web services, however, are based on plain-text XML, which is not suitable for microcontroller-based devices with their very limited resources due to XML's verbosity, its memory and bandwidth usage, as well as its associated significant processing overhead.

This thesis presents methods and strategies for realizing efficient XML-based Web service communication in embedded networks by means of binary XML using Efficient XML Interchange (EXI) format. We present a code generation approach to create optimized and dedicated service applications in resource-constrained embedded networks. In so doing, we demonstrate how EXI grammar can be optimally constructed and applied to the Web service and service requester context. In addition, so as to realize an optimized service interaction in embedded networks, we design and develop an optimized filter-enabled service data dissemination that takes into account the individual resource capabilities of the nodes and the connection quality within embedded networks. We show different approaches for efficiently evaluating binary XML data and applying it to resource constrained devices, such as microcontrollers. Furthermore, we will present the effectful placement of binary XML filters in embedded networks with the aim of reducing both, the computational load of constrained nodes and the network traffic. Dissimilar evaluation results of Vehicle-to-Grid (V2G) applications prove the efficiency of our approach as compared to existing solutions and they also prove the seamless and successful applicability of SOA-based technologies in the microcontroller-based environment.

Acknowledgements

I would like to take this opportunity to thank all those who contributed to this thesis. A special gratitude I give to my advisor Prof. Dr. Harald Kosch. Without his guidance and persistent support this dissertation would not have been possible. He provided me the opportunity to work in my own way and kept me motivated throughout the thesis. I would also give my thanks to Prof. Alfons Kemper, Ph. D., from the Technical University Munich (TUM) for volunteering to review my thesis.

Furthermore, I would like to acknowledge with much appreciation Dr. Jörg Heuer from the Siemens Corporate Technology in Munich. Based on my research cooperation with Siemens, he gave me a lot of support concerning my research direction and opportunities to participate in very interesting projects where I could learn a lot and involve my ideas and thesis findings. These especially include the ϵSOA and V2G project with the participation in standardization groups such as the ISO/IEC 15118, DIN 70121, and the W3C Canonical EXI. I would also like to thank Daniel Peintner as expert and co-developer of the EXI format. He gave me valuable support and feedbacks on my thesis. Furthermore, I would like to give thanks to Dr. Andreas Scholz with whom I had plenty of fruitful discussions of embedded topics. He helped me a lot to resolve difficulties in microcontroller programming. Many thanks to Dr. Richard Kuntschke for supporting me, especially in the filtering-based topics, and for all the inspiring discussions. I also want to thank to all other colleagues in team of Dr. Jörg Heuer who were always helpful and provided a pleasant working atmosphere. They include Anton Schmitt, Martin Winter, Johannes Bergmann, Martin Östreicher, Joachim Laier, Rainer Müller, Dr. Johannes Hund, Dr. Robert Nagel, Jürgen Götz, Christian Glomb, and Dr. Andreas Heinrich. Furthermore, I would like to thank my students Li Chen and Bharatesh Regoudar which supported me in implementations.

Many thanks also to Nadine Zimmerli. She spent her valuable free time and to proof read my thesis. A very special gratitude goes to my wonderful family with my wife Ulrike and my little son Elias. Without Ulrike's invaluable support, encouragement, and dedication to our child I could never have finished the thesis.

Contents

1 Introduction **1**

1.1 Embedded Network Definition 4
1.2 Application Development based on SOA with Web Services . . 6
1.3 Contributions and Chapter Outline 7

2 SOA-based Communication in Embedded Networks **11**

2.1 Introduction 11
2.2 Service-oriented Architecture 12
2.2.1 Definition and Properties 12
2.2.2 SOA in Embedded Environment 13
2.3 SOA with Standardized Web Services 15
2.3.1 Definition and Overview 15
2.3.2 XML 16
2.3.3 Schema Definition by XSD 18
2.3.4 Web Service Description Language 21
2.3.5 SOAP 27
2.4 Web Services Technologies in Embedded Environments 29
2.5 Related Work 32
2.5.1 SOA approaches for Embedded Networks 32
2.5.2 Devices Profile for Web Services 33
2.5.3 REST-based Web services with HTTP and CoAP . . . 34
2.6 Summary 36

3 Efficient and Scalable Web Service Generator **39**

3.1 XML Information Set 40
3.2 Efficient XML Interchange 41
3.2.1 Overview and Motivation 41
3.2.2 Coding Mechanism 43
3.2.3 The EXI Grammar G 44
3.2.4 The EXI IDs 47
3.2.5 XML-based Coding 48

3.3 Web Service Generation Workflow 50
3.3.1 Phase I: Schema Generation 52
3.3.2 Phase II: EXI Grammar Generation and Context-based Optimization 54
3.3.3 Phase III: Source Code Generation 60
3.3.4 Workflow Conclusion 65
3.3.5 Implementation 68
3.4 Evaluation 69
3.4.1 The Vehicle-to-Grid Dataset 69
3.4.2 Message Size 70
3.4.3 Code and Memory Footprint 71
3.4.4 Processing Speed 72
3.4.5 Evaluation Summary 74
3.5 Related Work 74
3.5.1 Binary XML 74
3.5.2 Web Services for Embedded Devices 75
3.6 Summary 77

4 Filter-Enabled Service Communication 79
4.1 Introduction 79
4.2 Querying with XPath Expressions 81
4.3 Basic Binary XML Filtering 82
4.3.1 Knuth-Morris-Pratt Algorithm, XML, and XPath . . . 83
4.3.2 XPath Normaliziation 87
4.3.3 The *BasicEXIFilter* Algorithm 91
4.3.4 Example 95
4.3.5 Conclusion 97
4.4 Optimized Binary XML Filtering 98
4.4.1 Determining Accepting and Predicate States 98
4.4.2 Determining Filter Grammar G_F 105
4.4.3 The *OptimizedEXIFilter* Algorithm 109
4.4.4 Example 110
4.4.5 Filter Code Generation 113
4.5 Experimental Evaluation 114
4.5.1 Implementation 114
4.5.2 Performance 114
4.5.3 Demo Network 117
4.6 Related Work 119
4.7 Summary and Comparison 123

5 Advanced Filter-Enabled Service Data Dissemination 125
5.1 Introduction 125
5.2 Problem Statement and Formalization 128
5.2.1 Notations and Definitions 128
5.2.2 Cost Function 129
5.2.3 Complexity 132
5.3 Filter-enabled Dissemination Algorithm 133
5.3.1 Closest Pre-Filter Node Algorithm 134
5.3.2 Dissemination Tree 137
5.3.3 Post-Filter Placement 142
5.4 Example 146
5.5 Extensions and Optimizations 150
5.6 Evaluation 151
5.7 Related Work 154
5.7.1 Data Stream Management Systems 155
5.7.2 Content-based Network Routing 156
5.8 Summary 157

6 Project Experiences 159
6.1 Introduction 159
6.2 Vehicle-to-Grid Standardization 159
6.3 Complex Charging Infrastructure 162
6.4 Related Work 163
6.5 Conclusion 164

7 Conclusion and Outlook 165

A Web Service Description 169

B XPath Queries 175

Bibliography 189

Abbreviations

ASN.1	Abstract Syntax Notation One
BFS	Breadth-first Search
BiM	Binary Format for Metadata
CoAP	Constrained Application Protocol
DCCQ	Device Class and Connection Quality
DFA	Deterministic Finite Automaton
DFS	Depth-first search
DOM	Document Object Model
DPWS	Devices Profile for Web Services
DSMS	Data Stream Management Systems
DTD	Document Type Definition
EV	Electrical Vehicle
EVCC	Electrical Vehicle Communication Controller
EVSE	Electric Vehicle Supply Equipment
EXI	Efficient XML Interchange
FI	Fast Infoset
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
ITU	International Telecommunication Union
JSON	JavaScript Object Notation
NFA	Non-deterministic Finite Automaton
OASIS	Organization for the Advancement of Structured Information Standards
PLC	Power Line Communication
REST	Representational State Transfer
RPC	Remote Procedure Calls
SAX	Simple API for XML
SECC	Supply Equipment Communication Controller
StAX	Streaming API for XML

SOA	Service-oriented Architecture
URL	Uniform Resource Locator
V2G	Vehicle-to-Grid
W3C	World Wide Web Consortium
WSDL	Web Service Description Language
WSN	Wireless Sensor Network
XML	eXtensible Markup Language
XML Infoset	XML Information Set
XPath	XML Path Language
XSD	XML Schema Definition

List of Figures

1.1 Growing microcontroller market history and forecast. Source: IC Insights [Insights, 2013] 2

1.2 (a) Sample embedded network in an abstract representation based on Defination 1.1; (b) ARM Cortex-M3 with an IEEE 802.15.4 compliant transceiver (blue blank) 4

2.1 Basic SOA model in an embedded context: 1) Service descriptions of node v_2 are published to the registry. 2) The registry instance supports identification of particular service functionalities which are required by client v_1. 3) Based on the service description discovered, v_1 is able to use and interact with the desired service. 14

2.2 Standardized Web services technologies projected to the SOA model with embedded nodes (compare Figure 2.1). 30

3.1 Web service-based interaction concepts (request/response and eventing) between two nodes in an embedded environment. . . 40

3.2 EXI grammar G of the temperature XML Schema shown in Listing 2.2. 46

3.3 Workflow of our Web Service generator for constrained embedded devices. 51

3.4 Generated $XSD_{SERVICE_MSG}$ that includes the SOAP message structure and embeds the device information (Temperature, Humidity, etc.) 53

3.5 EXI grammar representation of the example $XSD_{SERVICE_MSG}$ 55

3.6 Context-based grammar optimization of G_{ENC} and G_{DEC} for the client and Web service side. 59

3.7 Interaction between an EXI-based Web service (v_1) in a constrained embedded network and a client (v_3) that is a participating member of the Internet. Node v_2 has a bridge functionality that seamlessly renders plain-text XML messages into EXI format and vice versa, both in terms of data content and structure. To achieve this, v_2 needs only the same $XSD_{SERVICE_MSG}$ schema description as applied on v_1 to extract the EXI grammar for encoding and decoding. 67
3.8 V2G request and response messages 71
3.9 Request / Response measurement (roundtrip time) 73

4.1 Service data dissemination example: Nodes 3, 4, and 7 subscribe to service data (*value*) of node 1. 80
4.2 Navigation and search path for finding the AS for the query expression $Q_1 = /Envelope/*/status$ based on the EXI grammar G provided in Figure 3.5. 100
4.3 All accepting and predicate states in G based on Q_1, Q_2, and Q_3. 102
4.4 Route to the start state of the root grammar from the two marked AS (*status* and *value*) 107
4.5 Filter Grammar G_F 109
4.6 Filtering performance results of binary XML filter variants (*BasicEXIFiltering* and *OptimizedEXIFiltering*) and plain-text filter variants (YFilter with and without NFA) 116
4.7 Demo embedded network 117
4.8 Example of XFilter NFAs 120
4.9 Example of a YFilter NFA engine 121

5.1 Optimized service data dissemination example 127
5.2 Embedded network with 3 nodes 131
5.3 Embedded network with three processing nodes (1, 3, and 4) . 135
5.4 N_{emb}^T that spans v_{pre} (node 1) and service subscribers (nodes 4, 7, 8, and 9). An optimized service data dissemination is realized by placing post-filters at node 3 and 6. 143
5.5 Pre-filter grammar 147
5.6 $DisseminationTree$ algorithm 148
5.7 Post-filter grammar 149
5.8 Embedded network with $|V| = 50$: (a) Count of used classes in the dissemination path and number of used post-filters for each application. (b) Number of links of the dissemination path and average value of connection quality. 153

5.9 Embedded network with $|V| = 100$ 154

6.1 (a) EVSE, SECC (vertical white box within the EVSE), and EV; (b) Charging message trace in binary XML (highlighted in blue) with EXI and in plain-text XML. 160

6.2 Different instances of a standardized charging infrastructure based on 4 EVSEs: Arbitrary EVs are able to connect and communicate (by the EVCC) with one of the EVSE (represented as dotted lines). The communication unit of the EVSE, the SECC, is outsourced. The EVSE routes the messages to one of the SECCs (depending on resource load). Depending on the charging session and costumer demands, the SECCs are able to communicate with arbitrary backend services (represented as dotted lines). 163

List of Tables

3.1 EXI ID assignment of local names based on the XSD shown in Listing 2.2. 47
3.2 EXI ID assigment of URIs (initial entries) 47
3.3 Content analysis of the input WSDL 52
3.4 EXI ID tuple (local name ID and namespace ID) of the sample $XSD_{SERVICE_MSG}$ shown in Figure 3.4. IDs of the type declarations are not listed. 60
3.5 Code footprint (in kBytes). *Includes the Contiki OS with 6LoWPAN stack. 72
3.6 Static RAM usage (in kBytes). *Includes the Contiki OS with 6LoWPAN stack. 72

4.1 Abbreviated syntax for XPath expressions 82
4.2 Memory usage (in bytes) of different filter scenarios (2 XPath queries, 4 XPath queries, and 8 XPath queries) compiled for the ARM Cortex-M3 microcontroller. 118
4.3 Comparision between BasicEXIFiltering, OptimizedEXIFiltering, and YFilter 123

5.1 Ressources capability of nodes in Figure 5.1(a) 146
5.2 Network configurations for evaluation 152

List of Algorithms

4.1 $DeterminePrefixFunction(P)$ 85
4.2 $KMPStringMatcher(T, P)$ 86
4.3 $XPathNormalizer(Q, XSD)$ 90
4.4 $BasicEXIFilter(M^{EXI}, \{Q_1, Q_2, ...Q_n\})$ 93
4.5 $PredicateTest(id, v, Q^N, i, Q, q)$ 94
4.6 $DetermineASPS(G, Q)$ 103
4.7 $CheckASPS(s, Q^N, i)$ 104
4.8 $FilterGrammar(G, Q)$ 110
5.1 $FilterEnabledDissemination(N_{emb}, v_s, C, Q, XSD, \alpha)$ 133
5.2 $ClosestPreFilterNode(N_{emb}, v_s, C, G_F, \alpha)$ 136
5.3 $BestDCCQRoute(N_{emb}, v_s, v_d, \alpha)$ 137
5.4 $DisseminationTree(N_{emb}, v_{pre}, C, \alpha)$ 140
5.5 $BestDCCQTree(\{N_{emb}, N^C\}, v_s, \alpha)$ 141
5.6 $PostFilterPlacement(N^T_{emb}, Q, v_{pre}, G_F, XSD)$ 145

Chapter 1

Introduction

In recent years, embedded networks (a.k.a. sensor actuator networks) have increasingly become the topic of academic research and have had a growing impact on different applications and business areas, such as home/building and industry automation, the automotive industry, healthcare, and entertainment. In modern homes and buildings, embedded networks are typically used to control and manage mechanical and electric systems such as heating, air-conditioning, and lighting. Driven by ongoing research and standardizations in the domains of energy savings, smart grids, and comfortable home/office environments, these automation systems will be extended to new applications, which will consider more and more interconnected, heterogeneous devices, including those of other domains (e.g., refrigerators, washing machines, inverters, smartphones, etc.). The basic idea involves setting up an intelligent infrastructure of devices and services that provide accessible data to create additional values. One context is optimized energy usage in households; for example, an energy consumer such as a washing machine adjusts the washing time in terms of the energy price requested, e.g., from the Internet, or in terms of energy availability provided locally by solar energy plants and in relation to the weather forecast.

A similar perspective on intelligent infrastructure is true for the automotive industry: today's vehicles contain up to 80 networked and embedded control units, e.g., for driver assistance and hazard identification. In the near future, the intelligence of these networks will be optimized even more by introducing the possibility to have a car interact with vehicles close by in order to warn drivers of obstacles, such as accidents, and to avoid traffic congestion. In addition, electric vehicles will interact with local power infrastructure via power supply equipment and smart grid as well as with different stakeholders such as energy traders and clearing houses. A recharging process can be aligned with the predetermined time when the vehicle is

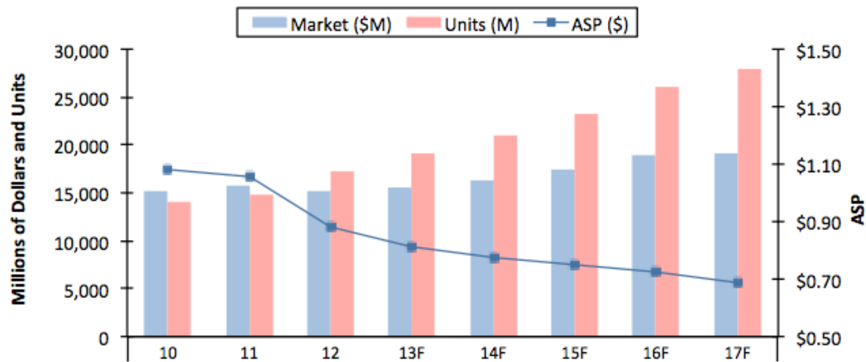


Figure 1.1: Growing microcontroller market history and forecast. Source: IC Insights [Insights, 2013]

required again, locally available (green) energy, and energy price. Users will be able to interact via a smartphone with their electric vehicles to request, for instance, the state of charge or to adjusting the required vehicle time that in turn may affect a rescheduling of the charging process.

These scenarios point toward a new communication paradigm in the future of the embedded domain. Traditional single and isolated embedded solutions which can be found in today's home automation or in the automotive industry, for example, are opening up to their environments and interacting with other embedded networks consisting of heterogeneous devices from other domains. More and more devices will become connected to each other, and together they will create new synergies and applications with new and different types of stakeholders. A major shift will also occur in the hardware used in the embedded domain: typical field devices, which are traditionally represented by sensors and actuators (a.k.a. actors), are becoming smart through the usage of programmable microcontrollers, which are able to run dedicated small applications and come with communication facilities allowing them to interact directly with other network nodes and even other field devices. Microcontroller platforms are low-powered, small computer systems that typically contain a small amount of ROM and RAM, CPU with a low clock rate, and communication interface with limited bandwidth. Ongoing reductions in price, low-energy consumption, and flexible use further the adaptation of such hardware systems in embedded networks. A study (see Figure 1.1) from IC Insights supports this assertion. It charts the historical trend and forecasts the total number of microcontrollers sold (in US Dollars and total number of units) and their average selling prices (ASPs) per unit [Insights, 2013]. A comparison of the years 2010 and 2017 reveals that units expected to be sold will have almost doubled whereas the

ASP per microcontroller will be approximately 40% less.

Given the growing impact of embedded networks, questions arise as to how to increase interoperability in terms of communication protocols to realize the cross-domain applications described above. Traditionally, protocol converter-based gateways have always been used between domains or company-specific networks, which process a set or subset of disjointed protocols (e.g., BACnet [ISO, 2003] and DALI [IEC, 2007]). However, the protocol mapping functionality of gateways is too costly in terms of installation and maintenance, and applications incur a processing overhead if gateways are used between networks or domains. Common end-to-end protocols would overcome gateways and would ease the development of applications for engineers, since they would not have to face multiple domain-specific protocols. Established IT technologies, such as the ones applied within the very successful Internet, provide a very interesting approach for increasing interoperability. IT technologies consist of well-known open standards and documentations, have a rich set of programming libraries and tools for development and managing, and, in general, enjoy high acceptance by developers. Currently, standardization consortiums such as the IETF and W3C are working on solutions to render established, seamless IT technologies feasible for the embedded domain. In recent years, approaches such as IETF 6LoWPAN [Bormann and Mulligan, 2009] were successfully developed to make IP feasible for constrained devices such as microcontrollers. This enables a seamless and an end-to-end IP connectivity to and from IT-based environments such as the Internet. In addition, it also creates the opportunity of adapting other well-known IT technologies above the network layer, such as UDP and TCP.

These new perspectives and possibilities for creating cross-domain applications, as described at the beginning, the increasing number of interactive, heterogeneous devices associated with this development, and the ongoing trend to adopt IT technologies for direct communication with embedded devices, including even microcontrollers, will change the architecture of traditionally hierarchic- and centrally-based embedded domains, such as described in [Scholz, 2011] for the automation systems. There will be embedded networks consisting of programmable nodes with varying kinds of hardware resources including memory, communication media, bandwidth, and processing capabilities, and the applications will be executed in a distributed way. Before we consider the challenges involved in developing of applications in more detail, we are going to introduce our definition of embedded networks, which takes into account the device resources mentioned above. Furthermore, we will associate the embedded network components with exemplary embedded hardware.

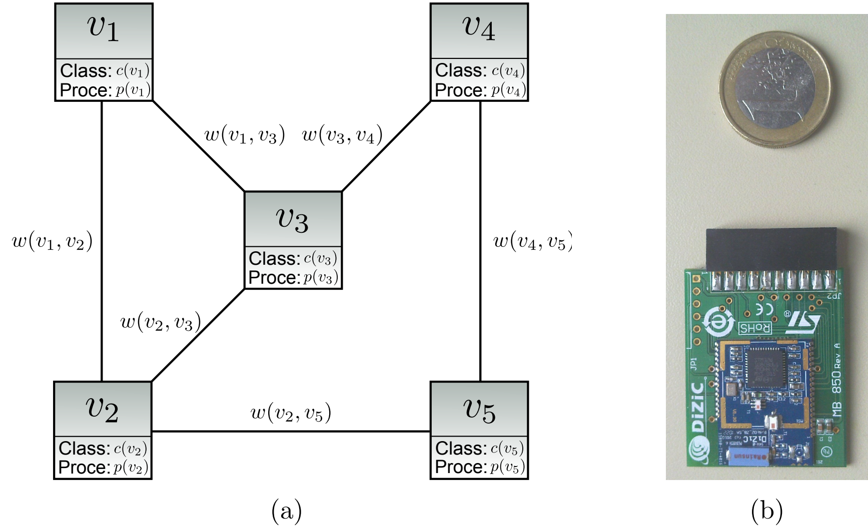


Figure 1.2: (a) Sample embedded network in an abstract representation based on Defination 1.1; (b) ARM Cortex-M3 with an IEEE 802.15.4 compliant transceiver (blue blank)

1.1 Embedded Network Definition

Embedded networks can consist of different kinds of hardware nodes that have varying kinds of computational capabilities, ranging from powerful PCs, to smartphones and routers, to small constrained devices that contain a microcontroller with very constrained resources. Relatedly, we can classify nodes that have particular resources available and have particular physical connections to neighboring nodes. In this context we define an embedded network as follows:

Definition 1.1 (Embedded Network)
An embedded network is defined by an undirected graph $N_{emb} = (V, E, w, c, p)$. V describes the set of vertexes or nodes in which each node represents a physical device. Set E describes the set of edges or physical relations provided by an underlying communication medium between two nodes. Such a relationship is described by the weight function w with $w : E \rightarrow \mathbb{R}_{[0,1]}$. The class of a node device is characterized by the device class function c with $c : V \rightarrow \mathbb{N}_{>0}$. An indicator of whether a device node is processable is given by the processable function $p : V \rightarrow \{0, 1\}$.

In this thesis we are not going to distinguish between the terms *network* and *graph*. In an abstract manner Figure 1.2(a) shows a sample embed-

ded network with 5 nodes, including their physical relations to each other. Definition 1.1 enables us to define a set of different kinds of device classes, which is used (or expected) in the network by the c function. Hence, in this thesis, we assign the highest number to nodes with the most constrained resources in terms of amount of memory and processing capability. Figure 1.2(b) shows a sample microcontroller-based platform from STMicroelectronics[1] that embeds an ARM Cortex-M3 and the popular low-power wireless IEEE 802.15.4 [IEEE, 2011] compliant transceiver. There exist different kind of Cortex-M3 configurations. In this example and as a microcontroller used as a reference platform in this thesis, we used a configuration that has 256 kBytes of programmable ROM, 16 kBytes of data RAM, up to 24 MHz of clock rate, and a wireless communication bandwidth of 250 kBit/s (2.4GHz IEEE 802.15.4). To clarify our class assignment principle based on our embedded network definition, we will now assume that we are using three different types of hardware classes in the network shown in Figure 1.2(a): v_1, v_2, and v_3 use an ARM Cortex-M3 each, v_4 uses a hardware system with a resource complexity found in home network routers, and v_5 represents a consumer PC. Thus, we would assign node v_5 the class complexity of 1 ($c(v_5) = 1$), node v_4 the complexity of 2 ($c(v_4) = 2$), and the nodes v_1, v_2, and v_3 the class value of 3 ($c(v_1) = c(v_2) = c(v_3) = 3$).

The weight function w abstracts the connection quality between two nodes that have a physical connection; this can be a wireless one, such as the low-power variants IEEE 802.15.4 and Bluetooth, or a wired one, such as Power Line Communication (PLC) or bus-based connections (e.g., RS-485 [TIA/EIA, 1998]). The connection quality can represent, e.g., the quality of the physical link, package loss ratio, current delay, or bandwidth. A combination of these would also be possible. We will assign a number that approximates the value of 1 to a superior connection quality. In contrast, a bad connection quality approaches the value of 0. An embedded network that only consists of microcontroller-based nodes for sensing purposes (e.g., temperature/humidity measurement or fire detection) and that only uses wireless connections, via the IEEE 802.15.4, for example, is typically called a Wireless Sensor Network (WSN). Typically, use cases of WSNs also have to take into account the limited energy resources since sensor nodes are battery powered [Dargie and Poellabauer, 2010].

The boolean function p will provide us with feedback if an embedded node has resources available for a potential additional application. This indicator does not only reflect the memory available, it can also associate the current processing load. We will assign the value 1 to nodes if they are able to

[1] http://www.st.com

install a potential application. A value of 0 will signal that no resources are available.

1.2 Application Development based on SOA with Web Services

The development, for example, of cross-domain applications in embedded networks will remain a highly complex task for developers in the future. They have to take into account a decentralized infrastructure consisting of heterogeneous devices that combine diverse resource capabilities, including hard boundaries from microcontrollers, for example, and different communication media with varying kinds of physical connection qualities to neighboring nodes.

In the IT domain, it has been proven that the Service-oriented Architecture (SOA) paradigm is a very suitable design principle for increasing interoperability in a heterogeneous system landscape. SOA would perfectly meet the unique characteristics of embedded networks, which mainly see everything as a *service*. In particular, we can project this concept onto the embedded domain for nodes providing services such as sensing, acting, and/or processing capabilities. These kinds of services can be requested or subscribed to by a service requester or client, respectively. Thus, an application in an embedded network would consist of a set of interacting services and clients hosted on distributed nodes.

When it comes to the technical realization of the SOA approach, Web services, which are standardized by the World Wide Web Consortium (W3C), are the most prominent and well established solution. Web services provide a rich set of application-relevant facets, including event-driven interaction, overlay routing, and security. As a fundamental concept, Web services clearly differentiate between the open interface used for interacting and using a service's functionality, and the application logic behind it all. This eases the development of applications, since developers only have to deal with the platform-independent interface for integration. Web services and their conventional implementations, however, are typically limited to powerful devices; constrained embedded devices such as microcontrollers (e.g., ARM Cortex-M3) are mostly out of their scope. This is explained by the fact that Web services operate on plain-text XML, which is a very verbose and redundant format. Processing XML to retrieve the actual data consumes a great deal of memory and processing time. Furthermore, data values are represented in an untyped manner, and additional datatype conversations have to be performed

for a subsequent processing of the values in the application. Transmitting service-based XML data would also negatively affect the bandwidth, increase network traffic, and reduce throughput in constrained embedded networks. All these drawbacks explain why standardized Web services have not yet been adopted for the embedded domain.

In this thesis, we will focus on this issue and wrestle with the challenge of bringing the SOA paradigm with standardized Web service technologies into the embedded environment, even when constrained devices, such as the ARMCortex-M3 with an IEEE 802.15.4-compliant transceiver presented above, are used. We will also concentrate on realizing event-based and filter-enabled Web service interactions and investigate how service data can be efficiently disseminated in constrained embedded networks. This optimizes the usage of resources in embedded networks due to the reduction of both computational load on nodes and network traffic and harmonizes the service interaction in embedded networks.

1.3 Contributions and Chapter Outline

This thesis presents methods and strategies for developing and realizing optimized Web service communication in embedded networks. Our approaches are scalable to resource-constrained embedded networks that consist of microcontrollers with limited memory, processing, and bandwidth. We focus on a model-based approach to develop an application based on standardized Web services as defined by the W3C. Furthermore, by using our pre-knowledge of the service requesters, we are going to design and adapt an optimized filter-enabled service data dissemination within applications to ensure efficient resource usage of embedded networks. For this kind of notification mechanism, we take into account the different kinds of device classes, the connection quality, and the processability of the nodes based on Definition 1.1 into account. Dissimilar evaluation results of future Vehicle-to-Grid (V2G) applications prove the efficiency of our approach as compared to existing solutions, and they prove the seamless and successful applicability of SOA technologies in the microcontroller-based environment. As a reference point in terms of embedded hardware we are using the ARM Cortex M3, which was presented in the previous subsection and is shown in Figure 1.2(b).

The chapter outline and detailed contributions are as follows:

Chapter 2 presents the concept of the SOA paradigm in more detail. This chapter focuses on introducing standardized Web services - the well-known technical implementation of SOA - and underlying W3C technologies and

protocols such as XML, XSD, WSDL, and SOAP. These technologies will form the basis of this thesis. Discussions and examples will show, however, that a direct adaptation of Web service protocols is not feasible for constrained embedded devices.

Chapter 3 introduces our optimized and scalable Web service code generation tool. Before explaining the model-based generation workflow in detail, we discuss the abstractions of XML data, followed by the XML Information Set (XML Infoset) and the possibility of efficiently encoding it in a binary XML representation by means of EXI. EXI is a grammar-driven approach and we will show how EXI can be optimized for our code generation purposes. Sample Web service instances, as taken from the V2G domain, for example, provide proof for our concept and its applicability in constrained embedded networks.

Chapter 4 describes concepts and approaches for filtering binary XML data to achieve efficient service communication within applications in embedded networks. We will present two mechanisms for evaluating binary XML data by relevance through given XPath expressions. We introduce *BasicEXIFiltering* that runs on top of an EXI grammar. Then, we introduce the *OptimizedEXIFiltering* that involves XPath expression within an EXI grammar. Lastly, we consider a number of XPath queries to evaluate both approaches.

Chapter 5 addresses the issue of efficient filter-enabled service data dissemination in constrained embedded networks. We will discuss the challenge involved in finding an optimized filter placement that takes the resources of embedded networks into account, such as connection quality, device class, and processability. A cost function will be derived to evaluate a service data dissemination variant based on the current resource status of an embedded network and participating Web service actors. The core of the chapter is devoted to our filter-enabled dissemination algorithm, which provides a resource-aware dissemination tree with dedicated nodes for pre- and post-filtering. Finally, in a simulated environment we will evaluate our filter-enabled dissemination approach by considering varying network and resource constellations.

Chapter 6 shares some of our experiences obtained from projects and active participation in the V2G standardization, the ISO/IEC 15118, which

decided to adopt efficient Web service-based message interaction for the embedded charging environment prompted by our developed approaches in this thesis.

Chapter 7 concludes this thesis and provides an overview of ongoing and future work.

Chapter 2

SOA-based Communication in Embedded Networks

2.1 Introduction

When it comes to the development of applications for embedded networks, special characteristics such as heterogeneity and resource restrictions of embedded nodes have to be taken into account. A very suitable design pattern for developing applications in a distributed environment is the SOA paradigm, which revolves around the idea of the orchestration of and communication with services, independent of the underlying hardware system.

This chapter gives an introduction about the SOA with its communication principle (Section 2.2) and discusses its significance for the embedded domain (Section 2.2.2). Web services are the very well-known technical implementation of SOA. An overview is given in Section 2.3. Basic techniques particularly used in standardized Web services include the markup language XML (Section 2.3.2) and its schema definition language XML Schema (Section 2.3.3). Section 2.3.4 introduces the Web Service Description Language (WSDL) protocol, which defines the interface of a Web service for interacting with a client. The last protocol presented in the context of Web services is SOAP, which is used as a message framework format to transport data between the Web service and a client (Section 2.3.5). Some of the examples provided in this chapter will be fundamental to later chapters, which will refer back to them. Section 2.4 will sketch the previously introduced technologies in the embedded environment and will explain their impact there. Section 2.5 on related works will provide an overview of SOA and Web service activities that also focus on the embedded domain.

2.2 Service-oriented Architecture

2.2.1 Definition and Properties

In the recent years, Service-oriented Architecture or SOA has become a very popular acronym. Definitions and descriptions vary depending on the literature used. This is due to the different points of view brought to the topic by enterprise architects, business executives, project managers, quality assurance engineers, and software developers [Lublinsky, 2007]. In this thesis our definition of SOA is based on the architecture design principle provided by the W3C [Haas and Brown, 2004]:

> *SOA is a set of components which can be invoked, and whose interface descriptions can be published and discovered.*

The term '*component*' is characterized by the W3C as a *software object, meant to interact with other components, encapsulating certain functionality or a set of functionalities.* We will associate the term '*component*' with the term '*service*' which is typically found in similar SOA definitions, e.g. of the Organization for the Advancement of Structured Information Standards (OASIS) [Brown and Hamilton, 2006] or in literature such as [Melzer, 2007] or [Rosen et al., 2008].

SOA does not provide a particular technology, but instead it provides an abstract concept of a software architecture for the publication, discovery, and usage of services. The user of a service is typically called *client* or *service requester*. Both terms will be used in this thesis. Below, we will explain the major properties that form the SOA paradigm. A similar overview can be found in [Melzer, 2007], among others.

loose coupling: One of the main abstraction properties of the Service-oriented Architecture is the *loose coupling* of individual services. On demand, applications can dynamically discover and use a particular service. More precisely, such dynamic service binding is processed at run-time and one is unaware at compile-time of the services that are bound [Melzer, 2007]. In practice, however, such scenarios are not left to chance by the application, and typically a developer defines a set of services or service domains that can be trusted and really provides the expected functionality.

reusability: In a SOA environment, an application is based on a number of services, and the services serve individual functions. An application is designed by chaining or orchestration of these services that reflect the desired

functionality. Based on the *reusability* principle of SOA, a service may be participate in one ore more individual applications.

registry: The instance where all services can be registered/published is called *registry* (or *directory*). The registry is used to search for individual functions of a service that are required for a client or an application.

standards: Once the desired functionality of a service has been discovered, the next step invovles the binding and usage of this function within the application. However, such straight harmonization is only possible if there is a clearly defined interface that is based on well-known *standards*. In general, the fundamental idea of the SOA paradigm can only be a success if the mechanism of providing, searching, and using of the service is based on open requirements that each SOA-based participant can follow and understand. Proprietary solutions inhibit the dynamic and loose coupling of services and would mean a return to inflexible and expensive gateway solutions.

development separation: The SOA paradigm clearly distinguishes between an interface for service/client interaction and the actual implementation behind. Based on this concept, applications may interact with a number of services that are based on different programming languages, operating systems, and hardware platforms. A developer only has to deal with a service's standardized interface and does not have to invest time and money on its specific implementation.

Depending on the literature used, additional characteristics such as *security* and *semantic* can exist [Melzer, 2007]. These topics, however, are not in the scope of this thesis; we will only concentrate on the fundamental properties of SOA discussed in this subsection. The next subsection will explain how the SOA design principle is applied to an embedded environment.

2.2.2 SOA in Embedded Environment

Based on the properties of SOA presented above, we are going to discuss the influence of this design paradigm if applied to embedded networks.

As the name SOA suggests, *service* is the main part of the architecture. In the embedded context, the functional capabilities of a service can be based on hardware units, such as sensors and actuators, or on data processing functionality [Scholz, 2011], e.g., comparing values against a threshold or scale conversion. Hence, an application in an embedded network is based on multiple services that are bound to an underlying physical hardware, such

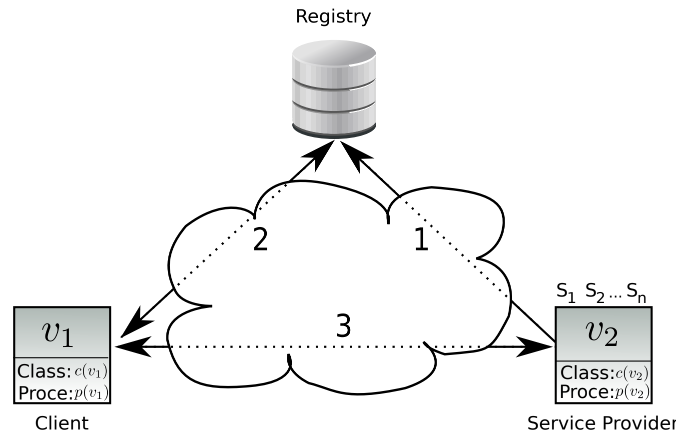


Figure 2.1: Basic SOA model in an embedded context: 1) Service descriptions of node v_2 are published to the registry. 2) The registry instance supports identification of particular service functionalities which are required by client v_1. 3) Based on the service description discovered, v_1 is able to use and interact with the desired service.

as sensors or actuators, and/or on services which could run independently of the hardware used and might also be located beyond the embedded networks, such as in the Internet [Scholz, 2011]. Participating embedded nodes can run several different kinds of services, with some but not necessarily all bound to the underlying physical hardware.

The process of how an orchestration based on the SOA principle can be applied is discussed with the help of abstract embedded network shown in Figure 2.1. A node v_2 provides one or more services (S_n). So as to make the services public on an embedded network, it will provide and register the interface description of the services at the registry. Please note that it is up to the node and use case as to which services are provided publically. There may be services which are only intended to be available locally and not publically. The registry instance itself can be an actively participating node in the embedded network and is able to gather service description any time, or it can be an instance node that is only online when it has to configure new applications in the network, for example. The latter variant is based on the concept of having a configuration phase (registry is online) and a run time phase (registry is offline). This way, the registry can be seen as a configuration computer that manages and sets up the application in the embedded network.

A client such as v_1 which is interested (or requires) a particular service, can fall back to the registry instance to identify the desired service functionality. Lets assume one of the services which is provided by v_2 meets the

requested interface by v_1. Based on the knowledge of the interface service description and the location where the service is hosted the client can start to interact or request the desired information. Thereby, the interaction can be performed in different kind of ways such as request/response or one way interaction.

2.3 SOA with Standardized Web Services

SOA is not a particular technology; instead, it describes an abstract design concept for developing software applications in a distributed environment. Web service is one of the mostprominent implementations of the SOA paradigm. This section introduces this technology and discusses its usage in detail.

2.3.1 Definition and Overview

As with SOA, there are several different approaches when it comes to defining Web services; any interpretation depends on the context of where it is applied [Melzer, 2007]. Whenever we invoke Web services in this thesis, we are referring to the definition that is provided by the W3C. Its definition is [Haas and Brown, 2004]:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

W3C describes a Web service as a technology that enables a *machine-to-machine interaction over a network.* For realizing applications using the Web service approach, different protocols are proposed: as service interface description that declares the served functionality and data model, the WSDL [Christensen et al., 2001] as standardized by the W3C shall be used. As communication framework for the interaction between two nodes in a network, the standardized W3C SOAP protocol [Lafon and Mitra, 2007] shall be applied.

Before we go into more detail on the WSDL and SOAP protocols, we will introduce the well-known markup language format, the eXtensible Markup Language (XML), and its schema language, the XML Schema Definition

(XSD), both of which form the basis of the Web services protocols in terms of interface description and for the Web service communication.

2.3.2 XML

The eXtensible Markup Language (XML) is a markup language and was standardized by the W3C in 2008 [Bray et al., 2008]. XML is a very popularly data format that is human-readable as well as machine-readable. It is applied in many application domains, including to describe user interfaces and graphics (e.g., Scalable Vector Graphics (SVG) [Ferraiolo et al., 2003], Extensible Application Markup Language (XAML)[1], and Android XML[2]) or office content (e.g., Office Open XML [ECMA, 2006] and OpenDocument [ISO, 2006]). Furthermore, XML has been adopted for many communication applications and standards when it comes to exchanging data between computer systems (e.g., Google AdWords[3] and Smart Energy Profile 2.0 (SEP 2.0)[ZigBee, 2013]). Thereby, the usage of XML for data interaction is typically defined in a Web service context.

Specifications of this markup language define different kinds of rules used for encoding documents or messages, respectively, in a XML manner. Listing 2.1 shows a simple XML document that carries temperature information. An XML document is in plain-text format and is mainly structured into two parts: the first part is called *prolog* or *XML declaration* and appears in the first line of each XML document. It contains information relevant to an XML preprocessor, namely the XML version used and character set of encoding in the document. The sample document in Listing 2.1 would tell a preprocessor that it is XML-structured using version 1.0 and that it uses the UTF-8 character style.

The second part is called a *root element* or *document element*. It builds the actually XML structure and embeds the actual data. Here, the root element is represented by the <Temperature> start tag, its content, and by the </Temperature> end tag. The content can once more be an element (also called nested element), simple text, or a mixture of both. In the example, there is a nested element <value> used, which in turns contains a text value '24.5'. Elements can also have attributes. An attribute can be seen within the <Temperature> element, with the attribute name *scale* and the value 'Celsius'.

[1] http://msdn.microsoft.com
[2] http://developer.android.com
[3] adwords.google.com

Listing 2.1: XML example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Temperature scale="Celsius">
   <value>24.5</value>
</Temperature>
```

An XML document can only be processed and parsed properly if the content is *well-formed* [Bray et al., 2008]. Such a document is well formed if, e.g., it only has one root element, each element has an end tag, and attribute values are quoted. The example in Listing 2.1 meets the demands and hence, it is well formed.

XML processors or parsers are used to gather information from an XML document. Mainly, two established variants exist: Document Object Model (DOM) [DOM, 1998] and Simple API for XML (SAX) [Megginson, 2000]. A DOM parser builds a tree data structure from an XML document. The tree structure can be accessed randomly or traversed in a way that extracts the needed data at the particular tree node/element position. For realizing this, the complete DOM tree has to be kept into the (runtime) memory. In contrast, the SAX parser is an event-based parser and does not require the complete XML document to be part of memory. It parses the XML document into a series of events. More precisely, each XML construct has a corresponding SAX event: *startDocument, endDocument, startElement, endElement*, and *characters*. Attribute-based information is typically provided within the *startElement* event. The SAX parser invokes (or *pushes*) the callback functions including the related events that occur while going over the XML document.

Besides the two known approaches of DOM and SAX, the Java programming language community (including, e.g., BEA Systems, Oracle, and Sun Microsystems) designed an alternative parsing approach, which is regarded as a median between DOM and SAX, called Streaming API for XML (StAX) [Benson, 2004]. StAX is a *pull*-based parser that uses a *cursor* that represents a particular position in an input XML document. The application triggers the cursor to determine the next content of the document. To identify the XML construct, different event types are used which are related to SAX events discussed above. Thereby, information of attributes are decoupled and represented by a separate event *attribute*.

In this thesis, we always refer to StAX when we talk about an XML-related parser. StAX gives us the flexibility to trigger each event separately, which enables intermedia proceedings and less memory usage.

2.3.3 Schema Definition by XSD

When using XML for data exchange, applications typically expect to process only XML messages that have certain elements and attributes used in a particular way. For example, an embedded device that only processes temperature messages structured as shown in Listing 2.1 would fail to retrieve the desired information if there is a derivation of data location, naming, and/or value typing. Schema languages can be used to provide an interface definition for which kind of XML data can be processed and understood.

The W3C XML specification comes with its own schema language description, called Document Type Definition (DTD) [Bray et al., 2008]. The rules provided enable the deceleration of the XML structure, such as the order and nesting of elements. Even though DTD is widespread and applied in many different markup standards such as the W3C XML Signature [Bartel et al., 2008], DTD has a very limited definition rule with no flexibility for extensions and data type variations [Meinel and Sack, 2003]. Data types are mainly restricted to a string-based type. Furthermore, DTD is not defined in XML style, which engendered efforts to study a new syntax.

Based on DTD's drawbacks, the W3C developed a new schema language, called XML Schema Definition (XSD). XSD's main strength is its support of basic type definitions, which are also called *simple types*: boolean, byte, short, int, float, string, etc.. In addition, restrictions can be assigned to simple types, such as a particular range of value intervals or default values listed by enumerations. Aside from simple type declarations, the XSD specification provides the opportunity to define complex data types, also called *complex type*. A complex type provides the rules to define the order of elements and occurrences as well as the presence of attributes within an element start tag of XML instances. In other words, complex types model the structure of XML.

To get a better understanding of how an XML Schema is used, Listing 2.2 shows an example that would provide the structure and datatype declarations of the temperature XML instance that is shown in Listing 2.1. Each XSD document begins with the root element <xs:schema>. Typically, the schema root element contains at least one attribute definition: The fragment

xmlns:xs="http://www.w3.org/2001/XMLSchema"

references that all elements and data types used in the schema come from the *"http://www.w3.org/2001/XMLSchema"* namespace. The association is done by the usage of the *prefix xs*. Other attributes are typically *targetNamespace* and *default* namespace definitions, which indicate that each element and attribute are defined by a particular schema context. For the sake

Listing 2.2: XSD example

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="Temperature">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="value" type="xs:float" />
   </xs:sequence>
   <xs:attribute name="scale" type="tempScaleType" />
  </xs:complexType>
 </xs:element>

 <xs:simpleType name="tempScaleType">
  <xs:restriction base="xs:string">
   <xs:enumeration value="Celsius" />
   <xs:enumeration value="Fahrenheit" />
  </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

of clarity and simplification, it is not part of our usage here.

Declarations which are nested as children of the *schema* root element are defined and visible as global. In our case, we employ two global definitions, namely a global element *Temperature* and a simple type *tempScaleType*. Global declarations can be reused by referencing them from other parts within the schema and/or from other XSD files. Consequently, a change in this global definition would automatically affect parts that refer to the global definition. In contrast, locally defined elements, attributes, or types are only present at the level where they are defined.

Considering the *Temperature* element declaration, it can be seen that element is defined as an anonymous complex type which contains other elements/attributes declarations (*value* and *scale*). The *sequence* group definition declares that the sub-elements must appear in a defined order. Since only one element is defined in our example, this requirement has no effect. In general, there are two further group definitions, namely *choice* (only one child element can occur) and *all* (child elements can appear in any order). The selection of the group is based on the desired restrictions or flexibility of parts within the XML instances. Please note that attributes are never included within the group definition (as can be seen at the *scale* attribute declaration). The XML Schema only provides the mechanism to define attributes but it does not provide any order guidelines. Thus, an XML parser can provide the defined attributes in any arbitrary order, which would nonetheless always be

valid.

Elements and groups in a complex declaration can define their own number of occurrences using *minOccurs* and *maxOccurs* attributes: minOccurs defines the minimum and maxOccurs the maximum number of times an element can occur. The default value for *minOccurs* and *maxOccurs* is 1. This value is also assumed when both attributes are not explicitly used within the element declaration. Thus, the *value* element has to occur exactly one time within the temperature element.

If an element or group is assigned *minOccurs="0"* (also called *optional declaration*), then this element or group with its nested sub-structure can be skipped in an XML instance. In contrast, an unlimited number of occurrences of an element or group is defined by *maxOccurs="unbounded"*. In order to define an upper bound, a fixed maximum number can be assigned. Again, attributes are excluded from this occurrence definition. XSD only provides the opportunity to declare an attribute as required or not. By default, attributes are always optional. In order to make an attribute as a mandatory part, the *use* attribute can be declared and set to be *required* within the attribute tag. For instance, so as to define the *scale* attribute as required, the definition has to be changed to

<xs:attribute name="value" use="required" type="tempScaleType"/ >

As we mentioned earlier, the strength of XSD is its broad support of basic type definitions. In our example, we are able to declare the *value* element directly as a float type to represent type aware temperature values. In addition, we can make some restrictions on the base type, for instance that the scale attribute shall only take the following values: *Celsius* and *Fahrenheit*. The globally defined *tempScaleType* simple type shows how this restriction on the string base type can be realized. There, the desired values are simply declared as *enumeration* within the *restriction* declaration. XML instances are only able to assign these two values. Otherwise, the instance is not valid within the context of the underlying XML Schema.

Before we end this section, we should mention that XML Schema specification provides a huge set of further declaration rules for refining XML instances in terms of structure and type intervals. In this section we introduced the most commonly used XSD declaration constructs, which also align with all XSD examples in this thesis. As a guideline, in this thesis we use upper-case characters for all elements that are type complex. Elements employing lower-case characters are typed simple. For additional reading about XML Schema and all of its opportunities, we recommend the W3C specifica-

tion [XML Schema, 2001] as well as related literature such as [van der Vlist, 2002].

2.3.4 Web Service Description Language

The Web Service Description Language (WSDL) is a W3C standard that defines a service interface on an abstract model in XML format [Christensen et al., 2001]. Currently, there two versions exist, namely 1.1 and 2.0 [Chinnici et al., 2007]. Even though WSDL 2.0 is the latest version, it is not as widespread and has not been as widely adopted in real systems as the 1.1 version [Melzer, 2007]. Hence, we always refer to version 1.1 in this thesis. WSDL itself provides a broad flexibility for defining a Web service. In this section, we will follow the Web service profile guideline as provided by the OASIS Web Services-Interoperability Organization (WS-I) [Chumbley et al., 2010]. The WS-I organizes the crowded space of interoperability specifications by providing guidelines on how to implement services that comply with standards and how to use combinations of Web service specifications to achieve interoperability [Nezhad et al., 2006]. Well-known development tools for Web services such as Apache Axis [Perera et al., 2006] or gSOAP [van Engelen and Gallivan, 2002] usually claims to comply with the WS-I profile.

Typically, a Web service serves different kinds of operations or Remote Procedure Calls (RPC) respectively. To define what the Web service offers, several main components are used in the WSDL definition shown in Listing 2.3.

Listing 2.3: Main components and structure of a WSDL definition

```
<definitions>

 <types>
  <!-- XSD definition of the messages -->
 </types>

 <message>
  <!-- abstract definition of the transported data -->
 </message>

 <portType>
  <!-- list the operations and which messages are
       involved -->
 </portType>

 <binding>
  <!-- defines the data format and protocol -->
```

```
 </binding>

</definitions>
```

An WSDL distinguishes between abstract and concrete definitions. An abstraction enables the opportunity of reusability[4]. This is the case for the *type* and *message* component, which defines the data being exchanged, and the *port type*, which provides one or more operation definitions. The concrete definitions instruct the protocol and data format, which is set in the *binding* container.

Below we will explain each component in detail. As an illustration, we are going to develop a simple Web service that shall run on an embedded device and shall serve information such as *temperature, humidity*, and generic *status* information (e.g., low battery, error, etc). Information about temperature and humidity shall be provided in two ways: an instance (the client) that is interested in this information can request it by remote procedure call, or subscribe to it. In subscription variant, the service only sends the information when there is any change, for example, in the humidity value. Assuming the service uses the C programming language for implementation, the RPC may have the following signature:

- *float temperature(uint8_t subscribe);*
- *uint8_t humidity(uint8_t subscribe);*

For subscription, a client needed to have sent at least one request message with the information of subscription (*subscribe*=1). To aviod clutter, we will not always show the full definition for each component. However, Listing A.1 in Annex A provides the full WSDL document that serves as the reference point for almost all further examples in the thesis.

types: This first element embeds schema declarations for the data used by the Web service. Thereby, the WSDL uses the standardized XSD syntax (see Subsection 2.3.3) to very precisely define the data model. In Listing 2.4 you can see the 5 root element definitions of our desired *(get)Temperature, (get)Humidity*, and *status*. Particular type declarations are defined below and follow the design principle presented in Section 2.3.3.

message: Message declarations provide service data, including the actual message, being exchanged between the Web service and the client. Thus,

[4]This is comparable to the global element and type definition in XML Schema (see the previous subsection)

Listing 2.4: XSD defination within the WSDL

```
<xs:schema targetNamespace="embedded:device:data" <!-- ... --> >

	<!-- Request and response root definition -->
	<xs:element name="getTemperature" type="GetType" />
	<xs:element name="Temperature" type="TemperatureType" />
	<xs:element name="getHumidity" type="GetType" />
	<xs:element name="Humidity" type="HumidityType" />

	<!-- Generic device information for the SOAP Header-->
	<xs:element name="status" type="statusType" />

	<!-- Complex and simple types -->
	<xs:complexType name="GetType">
		<xs:sequence>
			<xs:element name="subscribe" type="xs:boolean" />
		</xs:sequence>
	</xs:complexType>

	<xs:complexType name="HumidityType">
		<xs:sequence>
			<xs:element name="value" type="xs:byte" />
		</xs:sequence>
		<xs:attribute name="scale" type="humScaleType"
			use="required" />
	</xs:complexType>

	<xs:complexType name="TemperatureType">
	<!-- ... (see Listing A.1) -->

	<xs:simpleType name="humScaleType">
		<xs:restriction base="xs:string">
			<xs:enumeration value="Absolute" />
			<xs:enumeration value="Relative" />
		</xs:restriction>
	</xs:simpleType>

	<xs:simpleType name="statusType">
		<xs:restriction base="xs:string">
			<xs:enumeration value="Standby" />
			<xs:enumeration value="OK" />
			<xs:enumeration value="Error" />
		</xs:restriction>
	</xs:simpleType>
</xs:schema>
```

Listing 2.5: The message component

```
<!-- Header definations -->
<message name="SOAPHeader">
   <part name="header" element="edd:status"/>
</message>

<!-- Body definitions -->
<!-- getTemperature message -->
<message name="GetTemperatureMsg">
   <part name="parameters" element="edd:GetTemperature"/>
</message>

<!-- Temperature message -->
<message name="TemperatureMsg">
   <part name="parameters" element="edd:Temperature"/>
</message>

<!-- getHumidity message -->
   <!-- ... -->

<!-- Humidity message -->
   <!-- ... -->
```

there may be one or multiple message definitions in the WSDL, depending on how many operations (e.g., *temperature* and *humidity*) are provided.

We define four kinds of messages (see Listing 2.5): two that reflect the request messages (*getTemperature* and *getHumidity*) and two that reflect the result message (*Temperature* and *Humidity*). Each message contains a part element definition. This is comparable to the parameter of a function call. Since a function can also take more than one parameter, the message element can consist of more than one part. The part parameter type is associated with a concrete type defined in the type component above.

portType: This component defines the relation of operations and decides which messages are involved. It also provides the mechanism for declaring what kind of message pattern variant is used. In general, different design patterns are possible: request-response (the Web service receives a request message from a client and will return a response message), one-way (only a message is received by the Web service), solicit-response (the Web service sends a request message to the client and waits for a response message), notification (the Web service only sends a (notification) message to the client).

Listing 2.6 shows two portType definitions: *embeddedDeviceReqResInterface* and *embeddedDeviceEventingInterface*. The first portType mentioned

Listing 2.6: The portType component

```
<!-- Port defination for request/response -->
<portType name="embeddedDeviceReqResInterface">

   <!-- Access Operation -->
   <operation name="Temperature">
      <input message="edws:getTemperatureMsg"/>
      <output message="edws:TemperatureMsg"/>
   </operation>

   <!-- Humidity Operation -->
   <operation name="Humidity">
      <input message="edws:getHumidityMsg"/>
      <output message="edws:HumidityMsg"/>
   </operation>
</portType>

<!-- Port defination for eventing -->
<portType name="embeddedDeviceEventingInterface">
   <!-- ... -->
```

declares the involved messages of our two operations (*Temperature* and *Humidity*) in request-response style. The second portType defines the operations in notification style. This will come into play when clients subscribe to temperature or humidity data events. The Web service will independently send a one-way message to the client subscribers.

binding: This last component defines the format of the message and the protocol that shall be used for interactions between the Web service and a client. The WSDL standard does not specify details for such a binding [Christensen et al., 2001]. The WSDL 1.2 Bindings specification [Moreau and Schlimmer, 2003] offers different kinds of bindings for SOAP [Gudgin et al., 2003], HTTP [Fielding et al., 1999], and MIME [Freed and Borenstein, 1996], however, these are options, not requirements. In this thesis, we are going to concentrate on the SOAP-based binding, which is one of the most used variants in real applications and frequently referenced in the literature on Web service. The next subsection introduces the SOAP framework in detail.

Listing 2.7 shows usage of the SOAP binding as applied to our operations defined above. The binding element provides a name identifier and a type attribute, which references the portType defined above. By using the soap:binding element, the SOAP binding is enabled. This way, we can set the style *RPC* or *document* as well as the application transport protocol to be

Listing 2.7: The binding component

```
<binding name="embeddedDeviceWSReqRes"
   type="edws:embeddedDeviceReqResInterface">
   <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>

   <!-- Temperature operation -->
   <operation name="Temperature">
      <input>
         <soap:body use="literal"/>
      </input>
      <output>
         <soap:header message="edws:SOAPHeader"
                part="header" use="literal"/>
         <soap:body use="literal"/>
      </output>
   </operation>


   <!-- Humidity operation -->
   <operation name="Humidity">
      <!-- ... -->

</binding>

<binding name="embeddedDeviceWSEventing"
   type="edws:embeddedDeviceEventingInterface">
   <!-- ... -->
```

used when the SOAP is transmitted[5]. In our case, we selected the *document* style and the HTTP namespace to transport SOAP via HTTP since this is typically WS-I compliant. The soap:operation declares the binding of a operation and the soapAction attribute defines that the SOAPAction within the HTTP header can be used to identifying the service or the operation respectively. Finally, the soap:body and the soap:header elements indicate to which part of the SOAP framework (based on *Envelope, Header*, and *Body*; also see Section 2.3.5) the data is transported. In our case, the generic status information element will be only be transported within the SOAP Header of response messages. Meanwhile, all other data is embedded within the body part. The *use* attribute provides the data encoding, which is selected by the WS-I compliant *literal* here.

As mentioned at the beginning of this section, WSDL makes available many facets and variants for defining Web services. As with the XML Schema introduction, we have limited the declarations to those the thesis proposes. Additional information can be found in [Christensen et al., 2001], [Chumbley et al., 2010], and [Melzer, 2007].

2.3.5 SOAP

SOAP is a message format for exchanging data and executing RPCs, among others, on a Web services. This XML-based protocol is a standard of W3C and its latest version is 1.2 [Lafon and Mitra, 2007]. Originally, SOAP stood for *Simple Object Access Protocol*, however, this acronym was dropped in Version 1.2 of the standard. This is due to the fact that SOAP is not only suitable for accessing objects but also for exchanging, e.g., text documents.

SOAP has a very simple message framework: the root element is called *Envelope* and within this element an optional *Header* and mandatory *Body* elements are nested. The *Header* is typically used to transport generic information. This information can be categorized as *nice to have* but may be ignored. In contrast, information transported within the Header is essential to some use cases, e.g., for accessing purpose (e.g, with WS-Policy [Hondo et al., 2007]), trusting (e.g., with XML Signature [Eastlake et al., 2002]), or addressing (e.g., with WS-Addressing [Gudgin et al., 2007]).

The *Body* contains the actual payload. Typically, the embedded data are related to a RPC interaction. That means it is either request-based or response-based content. Based on our WSDL definition presented in the

[5]The style variants used do not indicate which programming model is used; instead they only pretend how to build a SOAP message based on the defined WSDL binding. See http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/ for a realted discussion.

previous section, Listing 2.9 shows an example in which the temperature information is requested by transporting the *getTemperature* element and its *subscribe* parameter.

Listing 2.8: SOAP temperature request message

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
 xmlns:es="embedded:device:data">
	<SOAP-ENV:Body>
		<es:getTemperature>
			<es:subscribe>false</es:subscribe>
		</es:getTemperature>
	</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In general, in the context of RPCs the first element within the *Body* element of a request message always signals to the Web service interpreter which operating method is intends to call. For example, in the case of the *getTemperature* element the *temperature* method is called as well as the *subscribe* parameter with the value *false* is passed. Please note that this relationship is defined in the portType component of a WSDL definition (see portType in Listing 2.6) as well as the corresponding data model is declared by the message and type component respectively (see Listing 2.4).

The corresponding response message contains the result of the RPC by embedding this information as a child element within the *Body* element. Listing 2.9 shows an example response message to the temperature request. This time, it also uses the *Header* element to provide the status information about the embedded device node (=*'LowBattery'*).

Listing 2.9: SOAP temperature response message

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
 xmlns:es="embedded:device:data">
	<SOAP-ENV:Header>
		<es:status>LowBattery<es:status/>
	</SOAP-ENV:Header>
	<SOAP-ENV:Body>
		<es:Temperature scale="Celsius">
			<es:value>21.1</es:value>
		</es:Temperature>
	</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In the case of a client subscription to humidity information, Listing 2.10 shows such a sample notification message. Please note that in the subscrip-

tion context, the notification message is not embedded within the header element.

Listing 2.10: SOAP humidity notification message

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
 xmlns:es="embedded:device:data">
    <SOAP-ENV:Body>
        <es:Humidity es:scale="Relative">
            <es:value>64</es:value>
        </es:Humidity>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Typically, SOAP is used in combination with or via the Hypertext Transfer Protocol (HTTP) [Fielding et al., 1997]; however, it is not restricted to it. SOAP is an independent transfer protocol to enable adaptation for different kinds of distributed applications and networks, especially if HTTP cannot be applied due to resource reasons, among others.

2.4 Web Services Technologies in Embedded Environments

To discuss the impact of Web service technologies in the embedded domain we will first directly project them onto the (embedded) SOA model introduced in Figure 2.1. In Figure 2.2, once again we find v_1 representing a service user (the client), the Web service provider v_2, and the registry. To provide one or more services installed on v_2 to the network, we use interface descriptions based on WSDLs, which are made known to the registry. Service requesters such as v_1 are able to obtain a suitable Web service via the registry. Thereby, the WSDL is provided that contains the detailed information about the functions sets (RPCs), data models, and where the service is located in the network. Based on this background, v_1 can then interact with the desired Web service running on v_2 using the SOAP message protocol.

This technology projection is, however, not really possible in the embedded domain. Specifically, constrained embedded devices are not able to keep a service description such as a WSDL in their memory or to process SOAP messages efficiently. This is due to the fact that WSDL and SOAP are XML based technologies, which are used in plain-text manner and claim a lot of memory. E.g., the WSDL that we defined in this thesis is approximately 7 kBytes. We would waste too much memory for the interface description alone if we were to keep this information on a microcontroller such as the

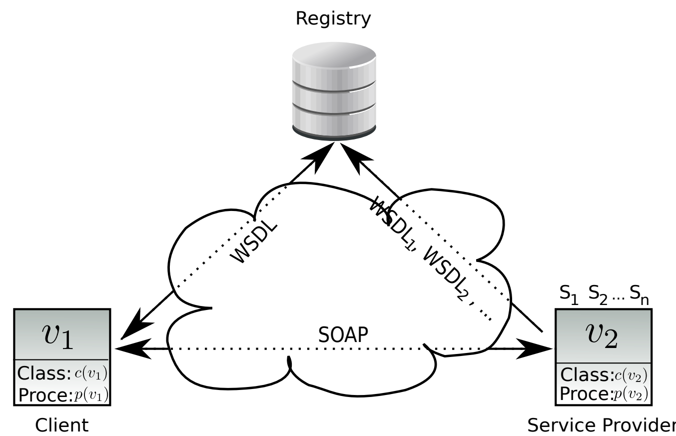


Figure 2.2: Standardized Web services technologies projected to the SOA model with embedded nodes (compare Figure 2.1).

ARM Cortex-M3, especially in relation to the required OS and actual programming logic. An alternative would involve *outsourcing* the service description to a more powerful instance node. An embedded node, such as v_2 may only provide information such as a URL where the registry can find the WSDL description of the service. Such discussions, however, are outside of the scope of this thesis. Please see related PhD work [Scholz, 2011] that addresses this issue and also introduces an alternative compact service description, the embedded service description language (eSDL).

In this thesis, we are going to focus more on the processing and interpretation of the XML-based data that comes into play through interactions between clients and services such as between node v_1 and v_2. SOAP messages like the ones presented in Section 2.3.5 also pose a challenge to constrained microcontrollers such as the ARM Cortex-M3 for several reasons: Let us consider the SOAP response message seen in Listing 2.9. A client that mainly requires the value of the temperature element has to parse previous elements, including *Envelope, Header*, and *status* in the message. After identifying the value element, the value itself has to be extracted (21.1); this, however, is also represented in a text/string format. For internal usage of the actually float-based value (e.g., comparing them against a threshold), we have to invest the processing overhead and convert the string value to float on the embedded node. In general, the parsing process is not only about identifying the element name that is desired, but structure and namespaces also have to be considered within the message context. E.g., Listing 2.9 and Listing 2.10 each provide a value element; however, they do so in different kinds of physical contexts (onc in the temperature and the other in the humidity context). XML parsing/processor tools taking this into account, however, come with

the corresponding memory usage which is not feasible for constrained microcontrollers [Peintner, 2014].

The size of SOAP-based messages is also an issue in constrained embedded networks in terms of bandwidth and throughput. Plain-text XML are very verbose and redundant, as can be seen, e.g., in Listing 2.9. The advantage of having human readability results in high network traffic and increased transmission time. Wireless low power communication or power line communication carries the risk of high package loss. This, in turn, may result in resubmissions, which, however, would additionally stress node resources. In general, the risk of losing a package increases with message size, since messages, which do not fit in one package, have to be parted in several packages. E.g., let us assume we have a wireless embedded network that uses IP-based communication with 6LoWPAN via IEEE 802.15.4. 6LoWPAN provides 108 bytes of payload (depending on compression settings) and the size of Listing 2.9 is 364 bytes. Consequently, 4 packages have to be used to transmit these messages. It would be desirable to use only one package, which, however, should consist of all the equivalent information in Listing 2.9.

After considering the existing development tools for Web service implementation, we have to face the issue that relevant approaches are not suitable for constrained embedded devices. The choice of development tools is already restricted by the fact that we only use C as a typical programming language for microcontroller devices. Identifying such frameworks that also meet the proper XML passing process as mentioned above, we can relate, e.g., to Apache Axis2/C[6] and gSOAP[7] [van Engelen and Gallivan, 2002]. Both are based on the stub (for the client side implementation) and skeleton (for the Web service side implementation) principles that are generated on a given WSDL. The files that are generated are used by developers to embed them in their applications. Based on the individual generation principle, only the functions (the RPCs) and its functions signature as defined within the WSDL are provided to the developers. This simplifies development and increases the adaptation of setting up Web service functionality. Although there is a code generation part that only provides the specific interfaces to the application, nested processes typical fall back on different kinds of generic libraries such as generic XML parser or databinder. For example., Axis2/C works with the standard GNU LibXML2[8] library, which is, however, very demanding in terms of memory [Peintner, 2014]. gSOAP comes with its own generic XML databinder mechanism and claims to have a memory footprint of under 150

[6]http://axis.apache.org/axis2/c/core/
[7]http://www.cs.fsu.edu/~engelen/soap.html
[8]http://xmlsoft.org/downloads.html

kBytes for same application scenarios. Applying our (small) Web service example definition as introduced in this chapter requires around 180 kBytes of memory as is. Consequently, a richer Web service in terms of RPC and data model would increase memory usage and would overwhelm constrained embedded devices[9].

2.5 Related Work

In this section, we will concentrate on related works that respond to SOA and Web service approaches that also relate to embedded topics. We will start by discussing different SOA projects in this domain. Then, we will present a Web service profile that is specified for devices, the Devices Profile for Web Services (DPWS). Finally, we will discuss an alternative to the SOAP-based Web services that is based on the Representational State Transfer (REST) approach.

2.5.1 SOA approaches for Embedded Networks

A number of projects address the SOA paradigm for the embedded domain, including OASiS [Kushwaha et al., 2007], SIRENA [Jammes and Smit, 2005], SOCRADES [De Souza et al., 2008], and ϵSOA [Sommer et al., 2009, Scholz et al., 2009, Scholz, 2011]. OASiS provides a programming framework to realize service-oriented applications. As an SOA implementation, however, OASiS do not invest in the usage of standardized Web service communication techniques. Instead, it uses a (proprietary) byte-sequence message structure for service interaction. For communication with a standardized Web service that is located, for example, in the Internet, an inefficient and expensive gateway solution [Amundson et al., 2006] is used, which is also not in our interest (see Chapter 1).

In contrast, the SIRENA project developed a framework which is based on standardized Web services with a set of Web service extensions that are listed in the Devices Profile for Web Services (DPWS) specification (see Section 2.5.2). SIRENA proved the SOA concept for different application domains, such as home and industry automation, telecommunication, and automotive. These demonstrators, however, consider more powerful embedded devices, which are able to run the developed DPWS stack with gSOAP. Thus, this approach is not applicable to our constrained microcontroller plat-

[9]The evaluation part of Chapter 3, in which we tested a more complex service application, will introduce further numbers of gSOAP.

forms with limited memory, processing, and bandwidth that we consider in this thesis.

The SOCRADES projects are based on the concept of SIRENA, but consider additional topics such as device life cycle and framework for device supervision. However, the SOCRADES infrastructure also proposes to be DPWS-enabled and considers more powerful device classes, such as the Sun Microsystem's (now Oracle) Sun SPOT[10] (4MB of flash, 512kBytes of RAM, 180MHz). Again, the DPWS-based messages that are used in plain-text manner would be not feasible to our considered microcontrollers.

A SOA approach for embedded networks that focus on constrained microcontrollers is the ϵSOA project. The ϵSOA platform combines 3 different design principles, which include SOA, model driven development, and a stream-based execution model [Scholz, 2011]. Features such as the service orchestration by application patterns or the optimization mechanism of best service position in constrained embedded networks ease the development of applications. In this thesis, we follow the ϵSOA interpretation of services and application: an application is interpreted as a set of data providers (sensors), data processors (application logic), and data consumers (actuators) [Scholz et al., 2009]. Services within the embedded network (also called *ϵService*) use compact XML-based messages[11] for payload transportation. The data context or the specific RPC is identified in the transport protocol by assigning a numerical ID. This approach has similarities to REST-based Web services that use always use the *HTTP Post* method (see Section 2.5.3). In this thesis, however, we will consider standardized SOAP-based Web services independently of the underlying transport protocol. SOAP is able and can be used platform independent to transport the operation context (the RPC) within the body along with either the parameter or result data. Our work, however, can be regarded as a supplement to the ϵSOA project that may use SOAP Web service capabilities for SOA-based application development in constrained embedded networks.

2.5.2 Devices Profile for Web Services

Originally, the Devices Profile for Web Services (DPWS) was a proposed standard from Microsoft for the usage of Web services on embedded devices. In 2008, DPWS was submitted for standardization to OASIS and the latest standardized version is 1.1 [Driscoll and Mensch, 2009]. The profile combines existing specifications associated with Web services[12]. Thereby, the

[10] http://www.sunspotworld.com/

[11] More details about the message representation are given in Section 3.5.2

[12] In the litereature, such Web service specification are quite often labeled with WS-*.

Web service specifications are selected in that scope that they *enable secure messaging, dynamic discovery, description, and eventing* [Driscoll and Mensch, 2009]. Thus, this includes WS-Addressing [Gudgin et al., 2007], WS-Discovery [Kemp and Modi, 2009], WS-Eventing [Malhotra et al., 2009], WS-MetadataExchange [Davis et al., 2011], and security related specifications such as WS-Policy [Vedamuthu et al., 2007]. DPWS pretend some requirements which parts of the specification must at least be implemented to enable interoperability between DPWS-based Web services and clients. Microsoft integrated DPWS into Windows Vista to control peripheral equipment.

Despite the defined profile being announced to be applicable in the embedded context, we believe this to be impossible when considering microcontrollers with very low memory, limited bandwidth capability, and restricted processing ability. E.g., Web service specifications such as WS-MetadataExchange may generate inflated SOAP messages that would increase network traffic within constrained embedded networks. This also leads to a complex parsing on the constrained embedded devices, which consumes a lot of processing time, memory, and reduces throughput. Motivated by these drawbacks, there is research being done on topics to make DPWS applicable on constrained devices, for example by defining a new device type with restrictions on the DPWS usages [Moritz et al., 2009] or defining a new message format in a tag length value (TLV) manner [Moritz et al., 2010]. At the time of writing, however, there is no known technical DPWS approach that meets all of our conditions such as the realizing efficient processing, producing, and transmission of XML-based messages within resource constrained embedded networks (also see Section 3.5.2).

2.5.3 REST-based Web services with HTTP and CoAP

Using the combination of WSDL and SOAP, Web services are also called SOAP-based or standardized Web services. REST-based Web services provide an interesting alternative.

REST stands for *Representational State Transfer* and was introduced by Roy Thomas Fielding in his PhD thesis [Fielding, 2000]. REST is not a protocol such as SOAP, but rather an architectural style. REST provides a set of architectural constraints that, *when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems* [Fielding, 2000]. Applications that rely on the principle of REST are also called *RESTful*.

A RESTful Web service manages a set of *resources* (the information of

which is provided) each of which shall be accessible via a Uniform Resource Locator (URL). The HTTP provides four basic methods for the four most common operations that can be applied on the resources [Richardson and Ruby, 2007]:

GET: Retrieve a representation of a resource

PUT: Create a new or modify an existing resource

POST: Create subordinate resources of an existing resource

DELETE: Delete an existing resource

Listing 2.11 shows an example REST call using the GET method to request the temperature *ressource* of a device with the id 10.

Listing 2.11: REST call using the GET method

```
GET /device/10/Temperature?subscribe=false HTTP/1.1
Host: embeddedNetwork.com
```

Unlike SOAP, which comes with a strict, predefined XML message framework (see sample Listing 2.8), only a URL is used and transmitted to the server. Besides addressing information of the requested resource, a list of parameters can be added by stating this with the '?' character. In our example, we send the *subscribe* parameter which is assigned with the value *false*. Additional parameters may be added by separating them with the '&' character.

An example response of the Web service is shown in Listing 2.12.

Listing 2.12: REST response message

```
HTTP/1.1 200 OK
Last-Modified: Tue, 22 Feb 2013 10:18:20 GMT
Content-Type: text/xml

<Temperature scale="Celsius">
   <value>24.5</value>
</Temperature>
```

The prolog contains some HTTP information, for example whether the request was a success (*OK*), how up-to date the resource is (*Last-Modified*), and which format the payload with the requested information is used (*Content-Type*). In general, REST is independent of the data format. Besides XML, JavaScript Object Notation (JSON) [Crockford, 2006], HTML, or simple text content are common alternatives.

In recent years, REST has become very popular. This is justified by the simplicity of requesting resources (only using a URL) and the close relation of the WWW. Resources can be addressed as URL, similar to links on Web pages. Responses can contain new *links* to other resources, which can be followed by the client.

Despite the simplicity of the REST concept, a direct adaptation into constrained embedded networks in an efficient manner is not possible. This is due to the HTTP used in plain-text manner. Similar to plain-text XML, microcontrollers have to parse characters to identify, e.g., the HTTP methods, and they have to do the processing overhead such as type conversion (e.g., parameter *subscribe* in the GET method in Listing 2.11). In the 6LoWPAN context, a HTTP header such as the response message would usally require a package of its own, before the actual payload information even starts.

To overcome this issue the Constrained RESTful environments (CoRE)[13] working group from the Internet Engineering Task Force (IETF) is currently developing an alternative HTTP protocol which is specialized for the constrained embedded environment: the Constrained Application Protocol (CoAP) [Shelby et al., 2013]. CoAP relies on HTTP and makes use of the GET, PUT, POST, and DELETE methods by reflecting them in a compact number representation that can be easily parsed. E.g., the GET method is assigned with the *CoAP Method Code* 1. In addition, CoAP also offers features such as built-in discovery, multicast support, asynchronous message exchanges, and an UDP [Postel, 1980] binding with optional reliability support [Shelby et al., 2013].

The CoAP sounds very promising, since all the features are developed in the context of constrained embedded devices, which we also focus on in this thesis. At time of writing, CoAP was in draft status. In this thesis, we are concentrating on established, standardized SOAP Web services with XML-based messaging, which are independent of underlying application transfer protocols such as HTTP or CoAP. Furthermore, to support interoperability and ease application development, we prefer the decision of using only one data format in constrained embedded networks.

2.6 Summary

In this chapter, we presented the idea of the SOA paradigm, introduced the standardized Web service technology, which is the well-known technological implementation of SOA, and projected them onto the embedded context.

[13]http://tools.ietf.org/wg/core/

One of the key advantages is the separation of the (public) Web service interface and the actual (private) logic implementation behind it. The interface is defined in a standardized way by using WSDL and the interaction to and from a Web service instance is realized via the SOAP message protocol. Developers only have to concentrate on this open standards and not on the individual implementation behind it (which may be proprietary specific).

Primarily, Web services were mainly introduced to overcome the interoperability issue in the Internet and put aside aspects of performance and resources. To a certain degree, Web services can be applied to the embedded environment; however, if microcontrollers come into the play, which we see participating more and more in the future of embedded networks, then XML-based protocols in plain-text manner are not really applicable because of their memory and bandwidth usage as well as processing overhead.

So as to keep all the advantages of the SOA paradigm with Web services, such as interoperability, reusability and flexibility, but also the broad support of development tools, it would be desirable to find a Web service development approach that is also scalable to microcontrollers and does not disregard W3C standardization requirements.

Chapter 3

Efficient and Scalable Web Service Generator

Using standardized Web service technologies would ease the realization of applications in embedded networks. Web services adaptation, however, did not reach the embedded environment. This is due to the fact that Web service protocols such as SOAP are primarily built on plain-text XML syntax. Consequently, constrained embedded devices such as the ARM Cortex M3 would be overtaxed when it comes to handling XML information in terms of memory and processing. In addition, an high-frequency message interaction would harm the network traffic as well as delay the execution of service applications in microcontroller-based embedded networks.

In this chapter, we are going to present a technique for optimized source code generation that enables the development of XML-based and standardized Web services that is also applicable in the constrained embedded domain. The goal is to provide a model-driven approach to set up SOA-based applications within Web services with the support of the usual communication pattern concepts, such as request / response and eventing between nodes (Figure 3.1). In addition, Web services instances that are generated shall fulfill the following requirements:

- *Compact messaging*: Message interactions between nodes should consist solely of compact information that leads to the usage of low message packages to decrease the risk of resubmissions (*reliability*) and would supports low network traffic.

- *Minimal code footprint and RAM usage*: The compiled source code of the generated Web service shall consist only of that functionality

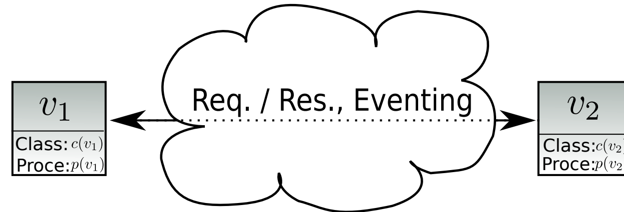


Figure 3.1: Web service-based interaction concepts (request/response and eventing) between two nodes in an embedded environment.

and use of memory which is really required to reflect the modeled Web service.

- *Efficient processing*: To realize a high node responsibility and throughput, Web service messages shall be processed and interpreted in a direct way without any processing overheads.
- *Compatibility*: Web service instances shall be based on the standardized W3C Web services protocols (see Section 2.3). This simplifies the integration and reuse of existing Web service applications and avoids the usage of expensive implementation of gateways for protocol mappings.

The first subsection of this chapter will address and discuss the abstraction of XML-based documents by means of XML Infoset (Section 3.1). Section 3.2 introduces the EXI format as an encoding format of the XML Infoset, which is a fundamental component of our Web service generator presented in detail in Section 3.3. As proof of concept of our innovated approach, we will present evaluation results from the V2G domain (Section 3.4). Before offering concluding observations, we will discuss related works in Section 3.5.

3.1 XML Information Set

Typically, a well-formed XML document or message use plain-text syntax based on start and end tags to structure the data (see Section 2.3.2). However, this representation is only one method of encoding in the context of the XML Information Set (XML Infoset). The XML Infoset is a W3C specification [Cowan and Tobin, 2004] that formalizes an XML document as an abstract data model by a set of *information items*. An XML document's information set can consist of eleven different items: *Document Information, Element Information, Attribute Information, Processing Instruc-*

tion Information, Unexpanded Entity Reference Information, Character Information, Comment Information, Document Type Declaration Information, Unparsed Entity Information, Notation Information, and Namespace Information. This way, the *Document Information Item* is always present in each XML's information set. Each information item contains a subset of additional properties. For example, the *element information item* contains, among others, *namespace name, local name*, *attribute*, *children*, and *parent* as properties to provide an element's namespace and local name information, the used attributes, which nested children it contains (which may be another list of *element information items*), and its superior element.

The information set of the XML document shown in Listing 2.1 would consist of the following items and properties:

- A document information item with a child[ren] *Temperature*
- An element information item with the local part *Temperature*, an attribute *scale*, and a child[ren] *value*
- An attribute information item with the local part *scale* and the (normalized) value *Celsius*
- An element information item with the local part *value*
- Character information items for the character data (*24.5*).

Closing items such as end tags or end documents do not exist in the XML Infoset representation. Structure information is given by the properties, such as *children* or *parent* in the information items. In conclusion, we are able to define and structure XML-based information in the same manner as we do with the start and end meta tag syntax that uses plain-text characters.

In the thesis, we consider the *XML* in the abstract manner in terms of XML Infoset. The plain-text representation will be seen as an opportunity to physically represent XML and will be noted as plain-text XML. An alternative and relativly new encoding variant of the XML Infoset is introduced in the next subsection.

3.2 Efficient XML Interchange

3.2.1 Overview and Motivation

Since the beginning of the XML format, there have been attempts to fit the plain-text data representation into a more compact and simple process-

able format. This was motivated by the aspect that XML documents have a tendency to be redundant and verbose, especially in terms of data ratio transported and the meta tags structuring the actual value data. Consequently, if XML is used as a transportation format, there is an increase in network traffic and a parsing overhead to retrieve the actual information out of the XML instance. This has also been the main reason that prevented adoption of XML as a viable transportation format in the embedded domain in the past.

A simple approach involves applying ZIP-based compression algorithms such as *DEFLATE* (combination of the LZ77 algorithm and Huffman coding) [Deutsch, 1996] to the plain-text XML messages. ZIP can have very good compression results [Bournez, 2009a] and would positively affect network traffic; however, we would increase the processing of XML messages on the device side. This is due to the fact that we are not able to work on the ZIP level to gather information from messages, since XML structure information gets lost when encoding with ZIP. Hence, we have to unzip before we are able to determine the data of the XML message. This additional processing overhead does not benefit of memory usage and computational processing.

In the context of XML compression, ZIP is characterized as a structureless compression method since it does not take structure rules of XML into account. A prominent variant which provides a structure-based coding is Binary Format for Metadata (BiM). The BiM format was originally designed for being used in conjunction with the MPEG-7 metadata format of the Moving Picture Experts Group (MPEG), which is used to represent audiovisual content [Avaro and Salembier, 2001, BiM, 2005]. BiM turned out to be very generic, since it is able to handle most XML-structured data, provided the data is valid to an underlying XML Schema [Sutter et al., 2005]. BiM achieves compression ratios comparable to ZIP [Niedermeier et al., 2002]. On top of that, BiM supports direct parsing on the bit stream level without prior decoding into plain-text XML. This is based on the fact that even in encoded binary format the structure of an XML document is still represented.

Even if binary XML with BiM made it possible to process XML-based messages quite efficiently and to achieve a small compression size as well, it did not become very established in the XML domain. There are many reasons for this, mostly involving expensive licensing issues and missing coding flexibility, such as schema-less or schema-deviated coding. Such flexibility may be required by some applications, such as customer extensions on protocols that can be found in the ISO/IEC 15118 [ISO, 2010].

The W3C, the inventor and standardizer of XML, faced this issue and

created a working group called XML Binary Characterization (XBC) [Goldman and Lenkov, 2005] to analyze the condition of a binary XML format that should also harmonize with the standardized XML format as well as with the XML Infoset. The outcome was the start of the W3C Efficient XML Interchange (EXI) format, which gained recommendation status at the time of writing this thesis (the beginning of 2011) [Schneider and Kamiya, 2011]. The W3C introduces EXI as follows:

> *The Efficient XML Interchange (EXI) format is a very compact, high performance XML representation that was designed to work well for a broad range of applications. It simultaneously improves performance and significantly reduces bandwidth requirements without compromising efficient use of other resources such as battery life, code size, processing power, and memory.*

The EXI group faced the facts about the disadvantages of plain XML mentioned above and developed a new representation for XML-based data, namely XML in an efficient binary representation with more flexibility than former formats. The different coding mechanism of EXI, discussed below, explains this flexibility.

3.2.2 Coding Mechanism

In general, EXI is a grammar-driven approach that achieves very efficient encodings for a broad range of use cases [Peintner et al., 2009, Bournez, 2009a]. Different coding rudiments exist, which depend on the usage of the binary XML format. The most important cases are listed and explained below:

schema-less Using this mode, there is no schema knowledge required of the XML-based data. EXI uses a specified grammar set to encode and decode the data. Element names and namespace URIs as well as the values of attributes and elements will be transmitted once. Internal string- and value-tables track all identifiers and values that occur. If an identifier or value occurs again, a number ID is encoded that is associated to a table entry.

This mode is recommended if no XML Schema knowledge is present or if a very frequent change is expected within data model. Even if there is a compression improvement, using *schema-less* coding is not recommended for the embedded environment. This is justified by the fact that there is still a high ratio of string as well as non-typed data representation in the stream that leads to processing overhead. Furthermore, this mechanism cannot be

limited in terms of memory usage since the grammar is extended at runtime whenever a new and unknown XML-based structure is present.

default, schema-informed This mode requires schema information of the data model used in the application. Based on this schema knowledge, EXI transforms all information included into equivalent EXI grammars. For a coding process, this grammar will be used and processed. This mode also enables features that allow encoding and decoding of information not defined by the underlying XML Schema. EXI is calling this case schema deviation coding and falls back to the basic grammar set also used in the *schema-less* mode.

This mode is recommended when the core data is known in an application and the underlying XML schema definition is present. In addition, this mode possesses the required flexibility to send arbitrary and non-schema defined data at any point. All core data will be encoded in an efficient way. Schema-deviated data will be signalized in an EXI stream and the data itself will be encoded based on the same mechanism as the *schema-less* variant.

strict, schema-informed In this mode, schema information is required of the underlying data model. Similar to the *default mode, schema-informed* mode, EXI grammars are created based on the given XML Schema. However, the mechanism for schema derivation is not permitted. XML-based messages which deviate the underlying XML schema, cannot be encoded into EXI representation.

This mode is especially recommended if the used data is known and fixed. Compared to the variants of *schema-less* and *default mode, schema-informed*, the *strict, schema-informed* mode produces the best encoding and processing results. This is based on the fact that no additional signalization, such as *is there a schema deviation* or a new identifier (e.g., attribute/element names), has to be encoded into the binary stream. Consequently, this mode is proposed for embedded devices with very restricted resources. Therefore, we will concentrate on this coding mechanism in this thesis. In the next section we will explain how to construct such a grammar for encoding and decoding XML-based data.

3.2.3 The EXI Grammar G

The EXI format uses a grammar-driven approach. Let us first define what we understand by an EXI grammar in the context of the W3C EXI specification and as used in this thesis:

Definition 3.1 (EXI Grammar G)
An EXI Grammar is a set of deterministic finite automata (DFA) where each automaton represents a complex type and its content (attributes, elements, and their occurrences) of a given XML schema. Each DFA contains one start state and one end state, which reflect the beginning and the end, respectively, of a complex type. The transition-state construct represents the sequential order of elements and/or the alphabetical order of attributes within a complex type. Optional definitions (e.g., *choice*, $minOccurs = 0$, etc.) are reflected by multi transitions and assigned an event code (EV).

In other words, G is based on regular grammar (the DFAs) derived from schema constraints such as element order and its occurrences. This is true, since schema rules are declared in a deterministic way. Declarations that would lead to ambiguous variants are not permitted and would lead to XML Schema validation errors. The EXI specification defines a predefined process of how schema information is to be transformed into DFA representations [Schneider and Kamiya, 2011]. In general, such a transformation step is justified by the aspect automatons are much simpler to process than (redundant) XML Schema information. In conclusion, the EXI grammar contains the same information as the XML Schema (including datatypes); however, it simplifies the concept of the XML structure by providing only those opportunities that logically occur next in the structure.

Figure 3.2 shows an example of what such an EXI grammar can look like for our temperature XML Schema which was provided in Listing 2.2. As can be seen, the grammar G is based on two DFAs: the *Root* and *Temperature* grammar. The *Root* grammar (a.k.a. document grammar) is a fixed predefined grammar that occurs in each EXI grammar representation of arbitrary XML Schemas. It contains all entry points of all root elements in the schema plus an entry point for a generic element. We will not focus on the possibility of accepting a generic element as root element here, since this is comparable to schema-less coding (see subsection 3.2.2), which is not within the scope of this thesis. For our schema case, we only define one root element, namely the *Temperature* one. Thus, the start state of the root grammar will contains two transitions: one that leads to the defined *Temperature* element state and one that leads to the generic element state which is indicated only by the dashed arrow here.

In general, the grammar indicates states distinguished by multiple transitions through an n-bit unsigned integer event code. Since EXI features the characteristics of entropy coding [Shannon and Weaver, 1949], the number of bits ($= n$) to represent the event codes of a state with m transitions is

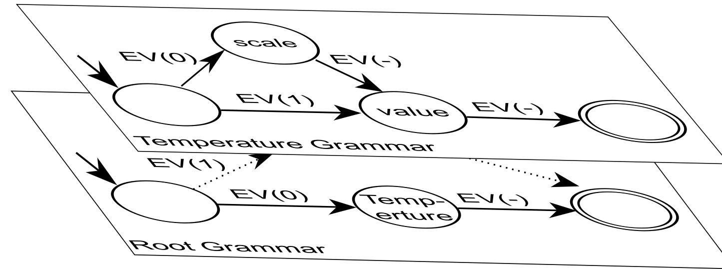


Figure 3.2: EXI grammar G of the temperature XML Schema shown in Listing 2.2.

determined by

$$n = \lceil log_2 m \rceil \tag{3.1}$$

The number of the event code itself depends on the alphabetical order of the element/attribute name that is indicated by the transitions. In this thesis, we will use the label EV for event codes that also directly shows the n-bit representation. E.g., $EV(01)$ is a valid event code for a state that has three or four transitions. A state that only has a single transition will be labeled with $EV(-)$.

Coming back to the EXI grammar example in Figure 3.2, the two transitions from the start state in the root grammar are assigned with event code $EV(0)$ for the *Temperature* transition and $EV(1)$ for the generic element transition (the generic element will always be seen as the item last ordered [Schneider and Kamiya, 2011]). Since the *Temperature* state is typed as complex (see Listing 2.2), it refers to the *Temperature* grammar, which represents the content of the type. Now, let us take a look at the details: The (local) complex type defines an optional attribute *scale* and a mandatory element *value*. Thus, we are once again given the choice of whether to embed the attribute *scale* within the *Temperature* element or not. Transforming this into the EXI DFA grammar representation, there are two transitions from the start state, with one ($EV(0)$) leading to the attribute state *scale* and the other leading directly to the *value* state ($EV(1)$). As this example also shows, the mandatory *value* element cannot be skipped in any way since the successor transition of the *scale* state is a single transition to the *value* state. If all information is transformed logically into corresponding states and transitions, the DFA will be closed by the end state as it is the last state that can be visited in the grammar.

After we have seen the idea of how an EXI grammar is constructed based on a given XML Schema, we will explain how such a grammar can be used to encode and decode XML-based messages. Before doing this, though, we

will introduce the EXI IDs.

3.2.4 The EXI IDs

The advantage of binary XML is the smaller and more compact message size when compared to plain-text XML [Bournez, 2009b]. This is achieved by avoiding transporting the string-based element and attribute names as well as the corresponding namespaces. Based on the event codes and the EXI grammar, the corresponding element or attribute can be identified. In order to interact efficiently with the EXI encoder (e.g., which element/attribute shall be encoded) and decoder (e.g., which element/attribute is present in the stream) EXI provides a compact and unique ID identifier of all named declarations in an XML Schema such as elements, attributes, complex and simple types (*Local-Name Partitions*). The EXI standard assigns IDs based on the alphabetical order of the *names* used in the XML Schemas. As an example, Table 3.1 shows the ID assignments for the XSD example presented in Listing 2.2.

ID	Local Name
0	Temperature
1	scale
2	tempScaleType
3	value

Table 3.1: EXI ID assignment of local names based on the XSD shown in Listing 2.2.

ID	URI Name
0	"" *(empty string)*
1	http://www.w3.org/XML/1998/namespace
2	http://www.w3.org/2001/XMLSchema-instance

Table 3.2: EXI ID assigment of URIs (initial entries)

The local name ID set, such as shown in Table 3.1, is always considered in a particular namespace context. Thus, a second ID is used for an efficient representation of the namespaces used (*Uri Partition*) in the schema. In that way, the first three IDs are reserved for the empty string and the default namespaces of the *Namespace* and *XML Schema Instance* (see Table 3.2). Additional namespaces will be mapped starting with the 4th entry

in the URI table. From there, the alphabetical order will be considered again.

Based on the EXI IDs, we are able to associate elements and attributes of an XML message in a very efficient manner. E.g., the content of the XML message that is shown in Listing 2.1 can be associated with a local name and URI ID tuple such as (0, 0) for the Temperature element, (1,0) for the *scale* attribute, and (3,0) for the *value* element. Note: since no namespace is assigned in this example, the namespace URI ID will always be 0 for the *empty string* entry. Through using such ID combinations, implementations avoid the overhead of the inefficient string comparison of element/attribute names and namespaces. This becomes especially apparent when name and namespaces are relatively long.

3.2.5 XML-based Coding

EXI uses a set of DFAs to encode or decode XML-based data. The data itself can be represented in different ways. Typically, we have plain-text XML representation. Alternatively, the XML-based information can be represented by a DOM or as the data structure of a programming language (e.g. Java Architecture for XML Binding (JAXB) [Fialli and Vajjhala, 2005]). The latter provides the benefit of having and working on typed-based data. Overhead processing of type casts can be avoided and we are able to retrieve the data directly from the EXI stream. We will further investigate this topic later in this chapter when discussing the generation of data structures for a programming language (Section 3.3.3).

The following example explains how the temperature-based XML message seen in Listing 2.1 is encoded to the binary XML representation with EXI. The XML Schema definition was used (see Listing 2.2) to determine the EXI grammar G seen in Figure 3.2. We will start with the root element *Temperature* and apply this to the start state in the *root* grammar. At this point, we have to check which transition leads to the *Temperature*. This is fulfilled by the transition labeled with $EV(0)$. The encoder would write the bit '0' to the EXI stream to signalize the decision which transition is to be followed and which state is to be visited next. Since the *Temperature* state is a complexType we know that it has a grammar representation of its own, and hence, we will continue with the start state in the *Temperature* grammar. There, we have to make another choice and we have to check what information is present in the XML message at this decision point. The *scale* attribute is used and consequently, we follow the transition with the event code $EV(0)$. This bit is also written into the EXI stream. Since the *scale* attribute is typed simple (as string/enumeration) the encoder would write

the enumeration value 0 to the EXI stream. The single successor transition leads to the *value* state, where we now have to encode the *value* element. Please note that no event code information has to be written to the stream for this transition. Again, the encoder would encode the value to the stream since the *value* is a simple type (float). In that particular case, EXI would part the float value to a mantissa and exponent representation, each of which are encoded as integer values. EXI supports integers of arbitrary magnitude, however, they are represented as a sequence of octets terminated by an octet with its most significant bit set to 0 [Schneider and Kamiya, 2011]. In other words, only this number of bytes is used to represent the desired value.

To complete our example, we run into the end state of the *Temperature* grammar and the encoder would continues with the *root* grammar. As can be seen, there is only a single transition left after the *Temperature* state, which also leads to the end state of this grammar, which here means the end of the encoding process. The outcome is summarized by the complete bit stream:

$$EXI_Stream := 10000000\ 00001111\ 01010000\ 00011000\ 00000000$$

The green bits represent the event codes, the red ones the scale value, and the blue ones the float value. As a prologue within the EXI stream, the first byte is assigned the mandatory and default EXI header value. The actual data is encoded starting with the second byte block. This example of a stream also shows that EXI is a bit-based coder that uses each and every possible bit within a byte. If there are free bits left in a byte block and no further information needs to be encoded, the byte will be padded with zeros. This case can be seen in the last byte of the stream.

Compared to the 112 bytes of plain-text in Listing 2.1, this XML-based representation is much more compact, only needing 5 bytes. All text-based and redundant tag information has been removed and the logical XML structure is represented by the two 1-bit event codes. Furthermore, the data values ($Celsius$ and 24.5) are type-aware encoded and not string-based represented as was done in Listing 2.1.

In general, the compression ratio depends on the the XML Schema and instances provided. There are use cases in which EXI representation is said to be over 100 times smaller than XML [Bournez, 2009b]. Works that provide approaches to further increase compression rates for some particular data types, such as multimedia content, also exist [Peintner et al., 2009]. In this thesis, we will present compression size of sample service messages that can be reached by applying the standard W3C EXI specification [Schneider and Kamiya, 2011].

The decoding process works in an almost similar way. The EXI stream is applied on G as input and the event codes in the stream navigates through the automata. The start state in the root grammar would read the first bit in the stream to decide which of the two possibilities has to be traversed. Since 0 signals the *Temperature* state, the *Temperature* tag can be initialized. In the *Temperature* grammar we would read the next bit at the start state to decide whether the *scale* attribute is present or not. Since this appearance is provided in our case, attribute information can be embedded within the *Temperature* with the decoded scale value ($Celsius$). Based on the grammar structure no event code has to be read to signal the *value* element after the *scale* state. Consequently, the *value* element tag can be initialized and its float value (24.5) can be decoded subsequently. Based on the end states, which are passed as the final steps, the end tags can be set and the outcome would be equivalent to Listing 2.1.

In the implementation context, it makes sense to distinguish grammar G between G_{ENC} for encoding purpose and G_{DEC} for decoding purpose even though both contain the identical DFA structure. This is justified by the fact that a grammar which is only used for encoding purpose should only keep the functions for encoding event codes and data types, for example. Vice versa, environments that only need the mechanism for decoding the EXI stream, G_{DEC} should solely serve the corresponding decoding functionality. We will focus more on this kind of grammar specialization in the next subsection.

3.3 Web Service Generation Workflow

The automatized workflow of our proposed Web Service generator for embedded devices with constrained resources, such as microcontrollers, is mainly divided into three phases:

I: Based on a given service description S_D, the service concept is analyzed and schema information is generated that describes all possible messages that can be understood by the service.

II: EXI grammar is generated and optimized.

III: Source codes are generated which contain the *EXI Processor*, RPC skeletons, message dispatcher, and the service stubs for the client.

Figure 3.3 provides an abstract overview of each step. The next subsection will narrate the details of each workflow phase. For a good step-by-step understanding we will apply the Web service example that was introduced and defined by the WSDL in Section 2.3.4.

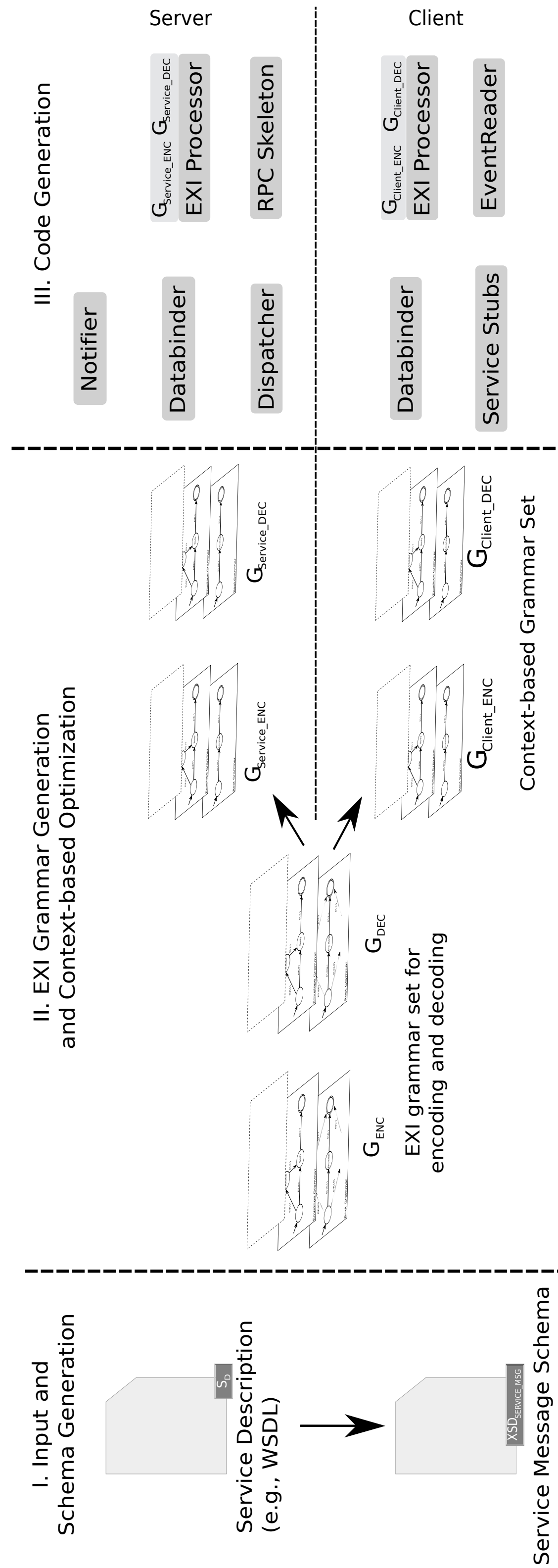


Figure 3.3: Workflow of our Web Service generator for constrained embedded devices.

3.3.1 Phase I: Schema Generation

This first step involves taking a service description S_D, such as a WSDL, as input. Reading the S_D, the content will be then analyzed in terms of which service procedure calls (RPCs) are served, which corresponding message patterns are used, and which particular data are exchanged within the service messages. Table 3.3 shows the result of this analyze step when the WSDL that was introduced in Section 2.3.4 is applied.

Operation	Req Message / Res Message	Req Data / Res Data	Cl^S	Cl^R	Se^S	Se^R
Temperatur	GetTemperature	subscribe	x	-	-	x
	Temperature	status, scale, value	-	x	x	-
Humidity	GetHumidity	subscribe	x	-	-	x
	Humidity	status, scale, value	-	x	x	-

Table 3.3: Content analysis of the input WSDL

The table shows which operations are served by the Web service and which request and response messages are involved. The data column shows which data elements and attributes are used within the corresponding message. The last four columns show the sending and receiving rolls of a client and the Web service. A marked entry in Cl^S points our which messages and data are involved when the client sends a message to the Web service. Entries in Cl^R highlight which messages and data a client can receive in particular. The same kind of information is also provided for the service side, namely by Se^S for the sending and Se^R for the receiving context.

The motivation underlying such an analysis table is quick identification of which kind of data is involved in general. E.g., it can be identified that the client will only embed the *subscribe* element in messages that are sent (Cl^S) and received (Se^R) by the Web service.

The next fundamental preparatory step in this phase involves the generation of an XML Schema $XSD_{SERVICE_MSG}$ that defines all possible XML-based messages with which the service can interact. For WSDL with SOAP binding, the generated $XSD_{SERVICE_MSG}$ will inherit the standardized *Envelope*, *Header* and the *Body* elements of the SOAP namespace (labeled XSD_{SOAP}). Furthermore, the particular service data that is defined in S_D is referenced and declared in the $XSD_{SERVICE_MSG}$.

Figure 3.4 shows the outcome of our sample WSDL. As can be seen, the SOAP framework, *Envelope*, $Header$, and *Body* make up the message framework. The $Header$ embeds the *status* element and the *Body* refers to

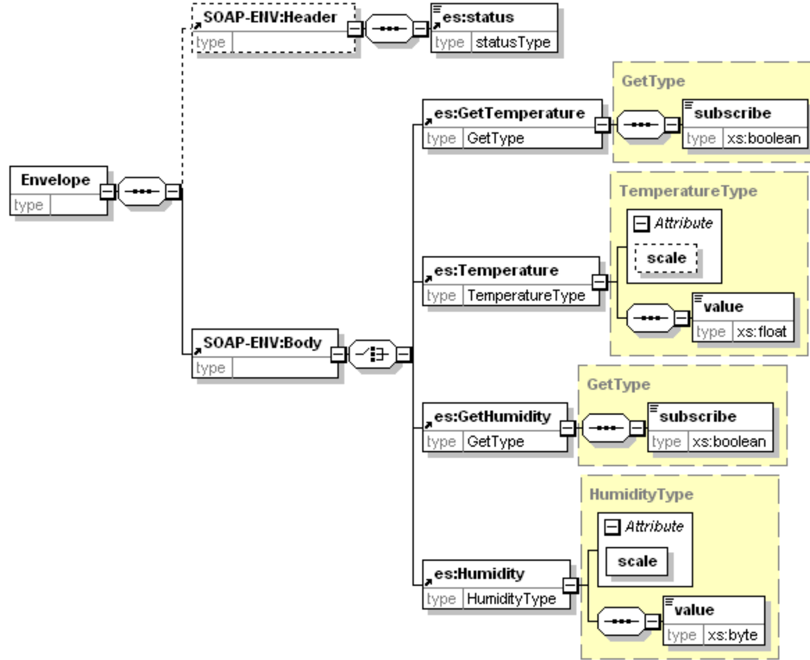


Figure 3.4: Generated $XSD_{SERVICE_MSG}$ that includes the SOAP message structure and embeds the device information (Temperature, Humidity, etc.)

the request and response-based *Temperature* and *Humidity* elements with their data type definition.

To formalize this process for SOAP messages, one can say that $XSD_{SERVICE_MSG}$ is a specialization of XSD_{SOAP} or

$$XSD_{SERVICE_MSG} \subseteq XSD_{SOAP} \quad .$$

Thus, all instances that can be built by $XSD_{SERVICE_MSG}$ are valid for XSD_{SOAP} and of course for $XSD_{SERVICE_MSG}$. More precisely: Let $I(XSD_{SOAP})$ be the set of all instances based on XSD_{SOAP} and let $I(XSD_{SERVICE_MSG})$ be the set of all instances based on $XSD_{SERVICE_MSG}$. For all message instances $i_{SERVICE_MSG} \in I(XSD_{SERVICE_MSG})$ is $i_{SERVICE_MSG} \in I(XSD_{SOAP})$ and valid for $XSD_{SERVICE_MSG}$ and XSD_{SOAP}. With other words

$$i_{SERVICE_MSG} \in I(XSD_{SERVICE_MSG}) \cap I(XSD_{SOAP}) \quad .$$

The generation of the $XSD_{SERVICE_MSG}$ will concludes this first working phase and the next phase will commence.

3.3.2 Phase II: EXI Grammar Generation and Context-based Optimization

Based on the $XSD_{SERVICE_MSG}$ generated in the first phase, the EXI grammar G based on Definition 3.1 is generated. To do so, we are following the predefined process of the EXI specification [Schneider and Kamiya, 2011] on how the schema information is to be transformed into the corresponding grammar representation (see also section 3.2.3).

Figure 3.5 shows the result of the transformation process. Mainly, 6 DFAs are constructed:

Root: This basic DFA always enables access to all existing root elements in the XML schema. In $XSD_{SERVICE_MSG}$ multiple root elements are accessible, namely *Envelope*, $Temperature$, *Get*$Temperature$, *Humidity*, and $GetHumidity$. In a SOAP context it is clear that we will always follow the *Envelope* root element. For reasons of clarity, we will only show this transition; all other root elements are indicated by dotted arrows. Since 5 root elements are defined, a unique event code - a digit with 3 bits - will be assigned to each transition. Based on alphabetical order, the Envelope transition is assigned the bits 000.

Envelope: The DFA *Envelope* reflects the SOAP framework with an optional $Header$ and mandatory *Body* element. If a transition with the event code 1 is followed from the start state, the $Header$ element is present. Otherwise, transitions with an event code 0 omit the $Header$.

$Header$: This grammar presents the $Header$'s content, which is only the simple typed *status* element. Due to its uniqueness, no event code is needed here.

Body: Based on the input WSDL, we know that there are several kinds of messages with which a client and the Web service interact. The *Body* grammar reflects the message context by addressing each of these messages as their own transitions from the start state. For example, the request message *Get*$Temperature$ is assigned with the event code 01 transition. Meanwhile, the response message $Temperature$ is reflected by the 11 transition.

$Temperature$: The attribute *scale* is defined as optional; consequently, the $Temperature$ DFA provides the transition depending on whether *scale* is present (transition with event code 0) or not (transition with event code 1 to the *value* state).

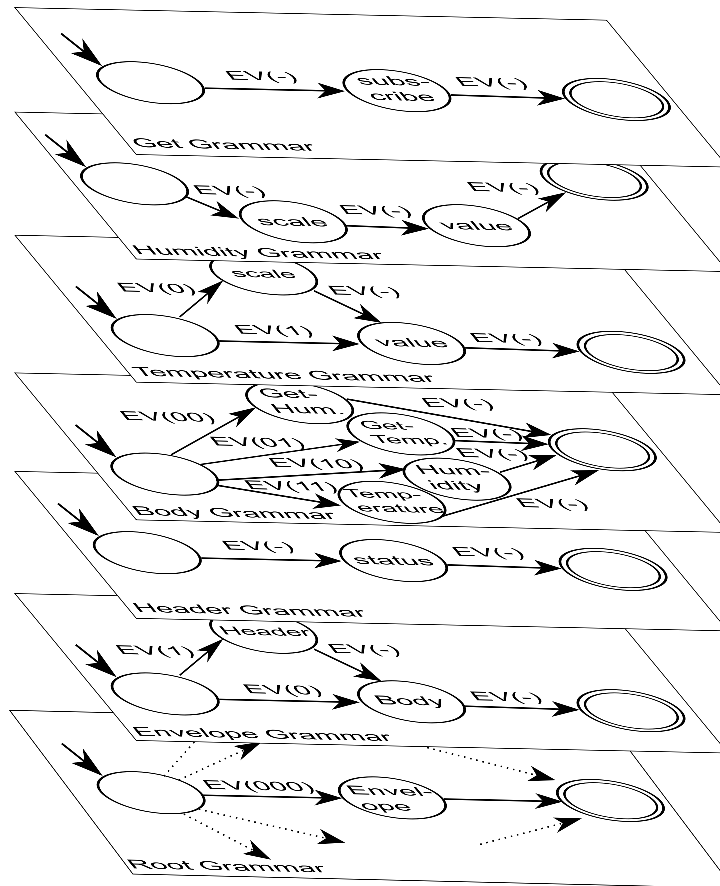


Figure 3.5: EXI grammar representation of the example $XSD_{SERVICE_MSG}$

Humidity: Unlike the *Temperature* definition, the *scale* attribute is mandatory. Hence, no event code is required in this DFA.

Get: This last DFA reflects the *Get* complex type that consists of one element: *subscribe*. Since this element is also mandatory, no event code has to be used for this automaton.

As discussed in Section 3.2.5, it is useful, in terms of implementation, to distinguish grammar G between G_{ENC} for encoding and G_{DEC} for decoding purposes. At this point, we could say that our process is complete. Grammars G_{ENC} and G_{DEC} applied to each side - client and service - would successfully encode and decode all kind of messages defined in the context of Web service description. However, such implementation incurs grammar overheads on each side. The following aspects explain this:

1. XML Schema descriptions such as the standardized XSD_{SOAP} include

many other global element declarations besides the *Envelope* declaration. From the point of view of modeling, this has some benefits in terms of reusability and avoiding redundancies of elements. In contrast, EXI has to take all of this into account when it comes to the EXI grammar building process. A Web Service, however, would never send a *Header* element without without the SOAP *Envelop* construct, for example.

In the case of our specified $XSD_{SERVICE_MSG}$ above, a number of global elements addressable from the root are also defined (e.g., *GetTemperature* and *Temperature*). The *Root* grammar which can be seen in Figure 3.5 sketches this aspect in the EXI grammar representation by the multiple transitions from the start state. To set up a valid Web service message, transition beyond the *Envelope* with event code 000 would never occur for encoding as well as for decoding. Event code patterns beyond this event code would not be seen as valid and messages would be rejected.

2. Considering the client side and service side separately, we would always discover the following aspect: A client sends only request-based messages to a service and receives only response-based messages from a service. That means EXI only encodes request-based messages and decodes only response-based messages. Thus, there is no need to provide the full mechanisms for encoding response-based messages or decoding request-based messages. This holds true for the service side as well: Services only receive request-based messages and send response-based messages. Supporting encoding of request-based messages and decoding of response-based massages is obsolete as well.

   Considering the example above, where we have a Web service that serves temperature and humidity information, only request messages such as *GetTemperature* and *GetHumidity* are decoded. Meanwhile, *Temperature*- and *Humidity*-based messages are encoded. Consequently, there is no need to provide a grammar for the Web service which encodes request message such as *GetTemperature* or decodes response messages such as *Temperature.* In turn, the client side needs this grammar structure to encode all request messages and to decode all response messages. Instead, there is no need to include the encoding grammar for response messages such as *Humidity* or the decoding mechanism for request messages such as *GetHumidity.*

Based on these two findings, we are optimizing the EXI Grammars G_{ENC} and G_{DEC} for the Web service and client-side usage. We also call this step

context-based optimization due to prior knowledge of the generic message framework and message handling on the client / service side. The result is a smaller encoding and decoding grammar set for the client side and for the service side: G_{CLIENT_ENC}, G_{CLIENT_DEC}, $G_{SERVICE_ENC}$, and $G_{SERVICE_DEC}$. In other words this grammar includes all states and transitions that are relevant to the client and service for encoding and decoding. Unnecessary states and transitions were removed. Event codes that are defined in G_{ENC} and G_{DEC} are kept in the remaining fragment parts of the smaller grammar set. Doing so guarantees that encoding and decoding of service-based messages provides the same result as G_{ENC} and G_{DEC} would.

To formalize this process, one can say

$$G_{ENC} = G_{CLIENT_ENC} \cup G_{SERVICE_ENC}$$

and

$$G_{DEC} = G_{CLIENT_DEC} \cup G_{SERVICE_DEC}$$

Let us consider an instance i_{REQ_MSG} of a request message in terms of $i_{REQ_MSG} \in I(XSD_{SERVICE_MSG})$. i_{REQ_MSG} can be encoded by G_{CLIENT_ENC} or G_{ENC} to an EXI representation $i^{EXI}_{REQ_MSG}$. To decode $i^{EXI}_{REQ_MSG}$ the grammar of G_{DEC} or $G_{SERVICE_DEC}$ can be used. Conversely, a i_{RES_MSG} that is a response message in terms of $i_{RES_MSG} \in I(XSD_{SERVICE_MSG})$ can be encoded by $G_{SERVICE_ENC}$ or G_{ENC} to an EXI representation $i^{EXI}_{RES_MSG}$. To decode $i^{EXI}_{RES_MSG}$ the grammar of G_{DEC} or G_{CLIENT_DEC} can be applied. For reasons of symmetry, it is also valid that

$$G_{CLIENT_ENC} = G_{SERVICE_DEC}$$

and

$$G_{CLIENT_DEC} = G_{SERVICE_ENC}$$

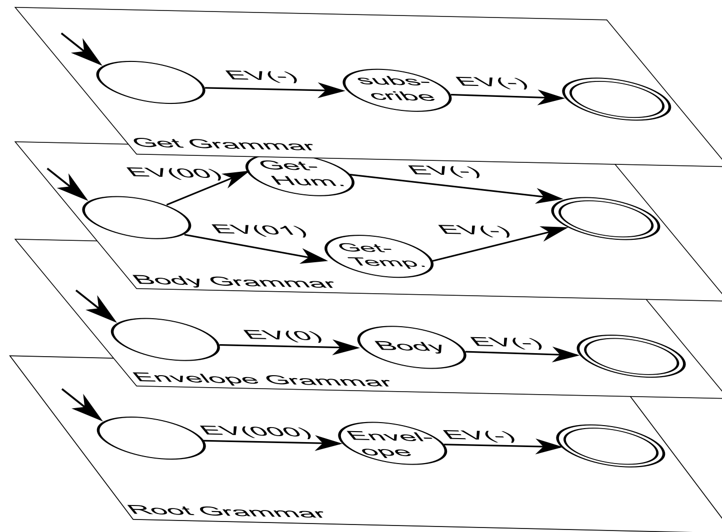
In other words, you will find the same DFA structure of G_{CLIENT_ENC} in $G_{SERVICE_DEC}$ since the information that is encoded on the client side is also decoded on the service side. Similarly, it is true that the same DFA structure of G_{CLIENT_DEC} can be found in $G_{SERVICE_ENC}$. Response messages encoded by the Web service also have to be decoded on the client side. Thus, the same DFA structure must be present.

To identify which messages and data are required for each side, especially which data information is needed for purposes of request or response, we are using the content analysis table that was generated in Phase I. For instance, Table 3.3 provides the answer in column Cl^S as to which messages and data need to be encoded on the client side based on our ongoing example: *GetTemperature*, *GetHumidity* and *subscribe*. Consequently, grammar G_{CLIENT_ENC}

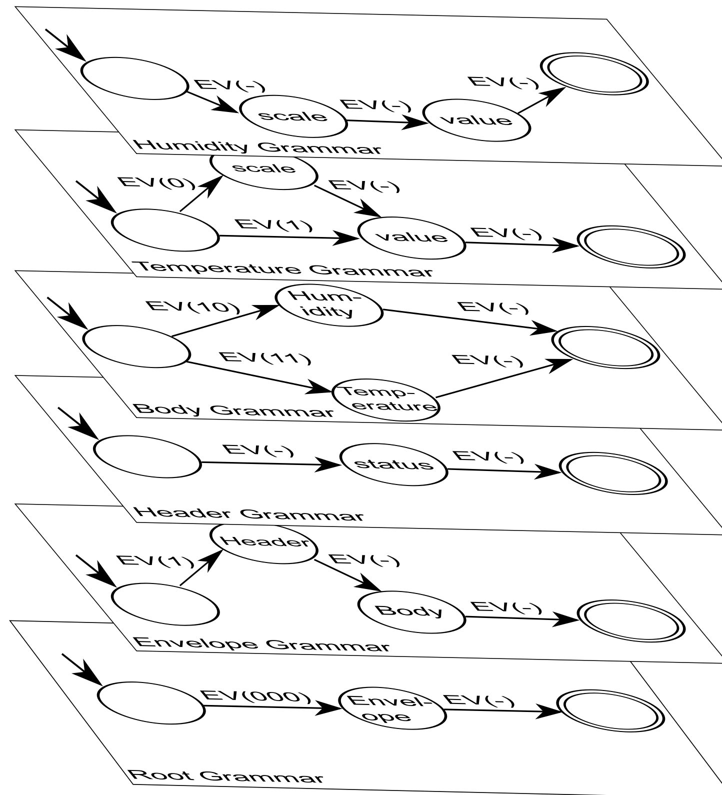
only contains these DFAs with those particular states and transitions needed to encode the EXI stream for the SOAP message that embeds the *GetTemperature* or *GetHumidity* requests. Based on the symmetry above, which is also underlined by Table 3.3 in columns Se^R to Cl^S we would get the same context-based optimization for $G_{SERVICE_DEC}$. Figure 3.6(a) shows the optimization for both grammars: G_{CLIENT_ENC} and $G_{SERVICE_DEC}$. All transitions and states that do not lead to *GetTemperature* or *GetHumidity* are removed. Furthermore, we limited the selection of the transitions of the root elements/states in the *Root* grammar to one. It must be noted that all event codes on the transitions are the same as were used in origin G. The compatibility requirement justifies this, since an instance shall have an identical EXI representation with G_{CLIENT_ENC} as with G_{ENC} and thus, we are able to decode analogously with $G_{SERVICE_DEC}$ or G_{DEC}.

Figure 3.6(b) shows context-based optimization for G_{CLIENT_DEC} and $G_{SERVICE_ENC}$ based on the columns Cl^R and Se^S of the WSDL analysis table. Here, the reduction is applied to multiple root elements in the *Root* grammar, the *Body* transition from start state in the *Envelope* grammar (all response messages contain the *status* value), the request elements (*GetTemperature* and *GetHumidity*) in the *Body* grammar, and its *Get* grammar representation. All DFA contain the structures to encode and decode the non-skippable *status* value within the $Header$ as well as the $Temperature$ and $Humidity$ information.

The technique of how we determine the states and transitions that are to be retained and the ones that are to be removed is not explained in this chapter. We are coming back to this issue when we are introducing the filtering mechanism in the next chapter.



(a) Optimized grammar G_{CLIENT_ENC} and $G_{SERVICE_DEC}$.



(b) Optimized grammar G_{CLIENT_DEC} and $G_{SERVICE_ENC}$

Figure 3.6: Context-based grammar optimization of G_{ENC} and G_{DEC} for the client and Web service side.

Before completing this workflow and moving on to the next phase, we are going to determine all EXI IDs of the local names and namespaces (see Section 3.2.4) defined in the $XSD_{SERVICE_MSG}$ (Table 3.4). Typically, EXI would also provide the ID representations of the simple and type names. Since these IDs do not affect our next step, we will skip over these unique numbers and only provide the IDs of the elements and attributes.

#	EXI ID	Local Name - Namespace
1	(0, 4)	GetHumidity - embedded:device:data
2	(1, 4)	GetTemperature - embedded:device:data
3	(3, 4)	Humidity - embedded:device:data
4	(5, 4)	Temperature - embedded:device:data
5	(8, 4)	scale - embedded:device:data
6	(9, 4)	status - embedded:device:data
7	(11, 4)	subscribe - embedded:device:data
8	(13, 4)	value - embedded:device:data
9	(0, 5)	Body - http://www.w3.org/2003/05/soap-envelope
10	(1, 5)	Envelope - http://www.w3.org/2003/05/soap-envelope
11	(2, 5)	Header - http://www.w3.org/2003/05/soap-envelope

Table 3.4: EXI ID tuple (local name ID and namespace ID) of the sample $XSD_{SERVICE_MSG}$ shown in Figure 3.4. IDs of the type declarations are not listed.

3.3.3 Phase III: Source Code Generation

This phase produces the Web service source code, which can be used by developers for their applications on the server and/or client side. We support the code in C, C++, and Java. Below, we are going to present sample listings of the code generation step in the C programming language; this is also the preferred language for the constrained embedded domain.

Data structure: Based on the XML Schema generated in phase *I* (the $XSD_{SERVICE_MSG}$), data structures are generated that map the identical hierarchy structure and equivalent type representation as defined in the schema. More precisely, complex types are represented as structs (C) or classes (C++ and Java) and simple types as primitives datatypes such as char, int, and float. E.g., an element that is typed as an unsigned short in an XSD document would be mapped to uint16_t datatype of the C programming language.

Listing 3.1 shows the data structure concept as a snippet based on our temperature/humidity $XSD_{SERVICE_MSG}$ example above.

Listing 3.1: C data structure of the temperature / humidity example

```
struct EXIDocumentType
{
  struct EnvelopeType Envelope;
};

struct EnvelopeType
{
  struct HeaderType* Header;
  struct BodyType Body;
};

struct HeaderType
{
  enum statusType status;
};

enum statusType
{
  OK, LowBattery, ERROR
};

struct BodyType
{
  struct TemperatureType* Temperature;
  ...
```

The *EXIDocumentType* structure will be generated as a basis structure for any kind of schema input. In general, this structure is used to map the root elements in the schema. However, in the Web service context we already know that only one root element is used (see Section 3.3.2). In the case of a SOAP binding it would be the *Envelope* element. Consequently, the *EXI-DocumentType* struct consists of this single entry. As we know, the *Envelope* element itself is defined as a local complex type within the $XSD_{SERVICE_MSG}$ and hence, its definition with its content is transformed into the *EnvelopeType* struct. Since the *Header* element is defined as optional, the *Header* variable is defined as pointer. Generally, selections by option or choice declarations of elements or attributes are transformed to pointers. Information always transported in each message, such as the *Body* element/variable, is defined statically. The data structure generator also takes into account defined restrictions on simple types. E.g., the *status* takes the string as a basis type. However, the Web service only accepts 3 status values (*OK, LowBattery,* and *ERROR*), which are predetermined as restrictions by using enumerations in the schema definition (see Listing 2.4). Consequently, we map this enumeration to a C enum construct which contains only these valid values (see *enum*

statusType) and avoid character representation in the programming language.

EXI Processor: To obtain the facility to process EXI data, the code of the *EXI Processor* is generated. The *EXI Processor* is a dedicated processor that is only able to process XML-based data that can be instantiated by the XML Schema generated in phase I. More precisely, the code only provides the mechanism required to encode and decode the EXI stream in the corresponding client and service contexts. This is done by applying the optimized and context-based EXI grammar of Phase II. Thus, the code generated for the client side only uses this mechanism to encode request messages and to decode response messages ($G_{Client_{ENC}}$ and $G_{Client_{DEC}}$). Correspondingly, the server side gets the code for decoding request messages and encoding response messages ($G_{Service_{ENC}}$ and $G_{Service_{DEC}}$). As interface the *EXI Processor* provides a StAX-like interface (see Section 2.3.2) to encode or decode XML-based events (startDocument, startElement, etc.). A detailed description of how such an EXI Processor is efficiently generated in terms of memory footprint and processing speed which, however, includes all required EXI functionality, is given in the PhD thesis of Daniel Peintner [Peintner, 2014].

Databinder: The databinder component that is generated contains the mechanism to serialize the data structure to the EXI stream representation and, conversely, to deserialize the binary XML stream to the corresponding data structure. It uses the StAX interfaces of the *EXI Processor* to obtain a valid XML-based stream or to read the values into the structure at runtime. To realize this, the corresponding StAX events and corresponding functions are generated in appropriate order, and only those are generated which are really required at this particular point. E.g., two complex typed based elements (*Header* and *Body*) are embedded in the *Envelope* element. Consequently, at this point and in the encoding context, we have to check first of all whether the *Header* is present. If so, the StAX *start element* event will be called passing the *Header* EXI ID (=(2,5); see Table 3.4). Otherwise, the body element is signalized (=(0,5)). It is continued recursively with encoding the content of *Header* or *Body*, respectively, and finally closed the *Envelope* complex type by the StAX *end element* event function. Listing 3.2 clarifies this process as a code snippet.

RPC Skeleton: The operations defined in the service description S_D are transformed into skeleton methods which developers later will have to fill in with the corresponding logic to react on the request data provided. In general, the RPC skeletons are only generated for the service side and the RPC

Listing 3.2: Serializing mechanism of the databinder

```
void serialize_Envelope(struct EnvelopeType* Envelope)
{
  if(Envelope->Header!=NULL)
  {
    encodeEXIStartElement(2,5);
    serialize_Header(Envelope->Header);
  }
  else
  {
    encodeEXIStartElement(0,5);
    serialize_Body(&(Envelope->Body));
  }
  encodeEXIEndElement();
}
```

function listed carries a signature that also corresponds to service description S_D.

Listing 3.3 shows the header C file generated, which contains all operations (*Temperature* and *Humidity*). The *param* variables contain the data of the request message. The result variable contains the information which is sent back to the service requester. In cases for which additional information is required, such as what is carried within the Header part, this struct is also provided as a parameter in the skeleton functions.

Listing 3.3: RPCs signature

```
void Temperature(struct HeaderType header, struct getTemperatureType
    param,struct TemperatureType result);

void Humidity(struct HeaderType header, struct getHumidityType param,
    struct HumidityType result);
```

Dispatcher: The dispatcher method generated builds the core of the Web service and is only used on the server side. The dispatcher takes the XML-based EXI stream and, using the databinder mechanism, starts to deserialize the stream. After all required data has been parsed the corresponding RPC with the received parameters is invoked. The return parameters provided are serialized as EXI streams once again. Summing up, the complete service message is interpreted and constructed automatically at runtime.

Listing 3.4 sketches the idea of the Web service dispatcher and shows the interaction of the different code fragments generated: Firstly the passed

Listing 3.4: Web service dispatcher

```
void dispatcher(uint8_t* inStream, uint8_t* outStream)
{
  struct EnvelopeType Envelope;
  deserialize_EXIStream(inStream, &Envelope);

  if(Envelope.Body.getTemperature!=NULL)
  {
    Temperature(Envelope.Header, &Envelope.Body.getTemperature, &
        Envelope.Body.Temperature);
  }
  else if(Envelope.Body.getHumidity!=NULL)
  {
    Humidity(Envelope.Header, &Envelope.Body.getHumidity, &Envelope.Body
        .Humidity);
  }
  serialize_Envelope(outStream, &Envelope);
}
```

byte stream *inStream* (dispatched from the transportation stack), which represents the request message, is decoded (*deserialize_EXIStream*) and mapped to an Envelope data structure instance *Envelope*. After that, it is determined which RPC has to be called: in case of the initialization of the GetTemperature field within the Envelope data structure, the generated *Temperature* method (by prior implementation) with the corresponding parameters is called to determine the temperature information (*scale* and *value*). Since the Header structure is also passed, the device *status* can be set within the *Temperature* function. As a final step, the response EXI stream is prepared by passing the *Envelope* data structure to the serializing method (*serialize_Envelope*).

Notifier and EventReader: The *Notifier* enables the developer to set up event messages for one or more clients. Compared to the dispatcher, the notifier gets along without request messages and thus, it only provides the mechanism for serialization. On the opposite side, the *EventReader* enables the client to interpret an event message of a Web service. In that case, only the interfaces for deserialization are realized.

Service Stubs: The client is able to either call remote methods (the RPCs) or request data from the Web service by using the service stubs generated. The request with its data is encoded as an EXI message stream, transported to the service, accepts the response message and provides it in a data struc-

ture. Since the service stubs work almost like the service dispatcher (in reverse sequence), a code snippet is not shown here. In general, the signatures of the service stubs are equivalent to the RPC skeletons. Thus, our example would work with the functions seen in Listing 3.3.

3.3.4 Workflow Conclusion

Based on a defined service description, which can be designed by well-known modeling tools, the generator produces a source code containing the mechanism to process and interpret XML-based messages. Developers do not need to investigate request/response mechanisms or efficient message representations and their processing by means of EXI. The same interfaces and concepts are provided that are typically used for Web service and hence, the adaptation into existing or involving new applications is straightforward. Based on our optimized code generation approach, the code footprint of the Web services are very small and feasible for many kinds of microcontroller platforms such as the ARM Cortex-M3. E.g., the code footprint of our temperature/humidity Web service compiled with the Contiki OS [Dunkels et al., 2004] is approximately 50 kBytes (10 kBytes for the Web service itself and 40 kBytes for the OS). In addition, our Web service generator approach avoids the disadvantage of Web service solutions that are based on plain-text XML, which negatively affect network traffic and message processing, especially in constrained environments. E.g., the messages size of the temperature/humidity Web service example generated ranges from 2 bytes (request messages) to a maximum of about 10 bytes (response messages). Working with EXI, we remain compatible with XML at the XML Information Set [Cowan and Tobin, 2004] level, rather than at the XML syntax level [Schneider and Kamiya, 2011]. If it is desired to get the plain-text XML representation, e.g. for debugging reasons, the EXI stream can simply be decoded into the plain-text XML syntax by utilizing generic EXI processor libraries and providing the XML schema that underlies the XML-based messages.

Our approach is also suitable for participating in existing applications which (partially) use Web service or client-side implementation that process plain XML messages. To realize this, an intermediate translator that can transform plain-text XML into an EXI representation and vice versa has to be involved in communications on the application level. Figure 3.7 shows a concept for such an intermediate translator, as applied on a separate node (v_2). Node v_1 is a member of the constrained embedded network and serves an EXI-based Web service with different RPCs. A client v_3 from the Internet requests the Web service of v_3 by using messages in plain-text XML such as SOAP. Node v_2 has a bridge functionality that has access to both

networks. If this node receives a request message from a client addressed to v_1, the message is transformed into the EXI representation. In the same manner, the corresponding response message is transformed back to plain XML and forwarded to v_3. All these transformations are only valid, if the corresponding EXI grammar is present which is also used at v_1. This is realized by the knowledge of the corresponding service message XSD $XSD_{SERVICE_MSG}$ extracted of a service description S_D such as a WSDL (see Phase I) or by a direct representation of the EXI grammar without the prior grammar construction process [Peintner, 2014].

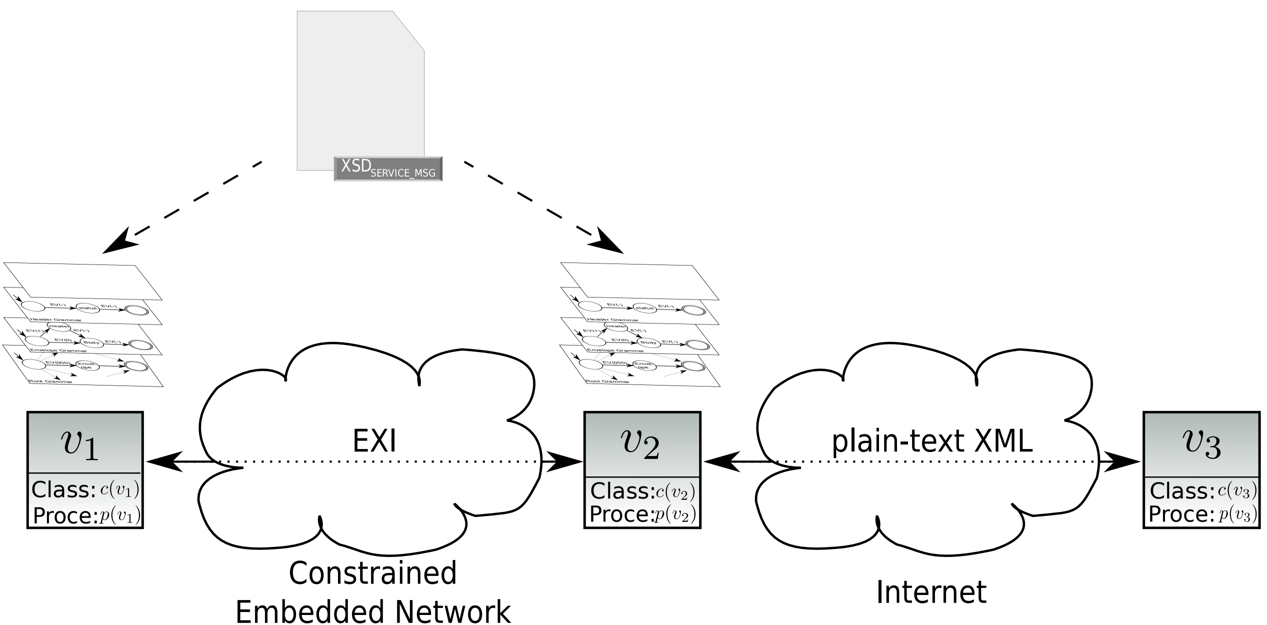


Figure 3.7: Interaction between an EXI-based Web service (v_1) in a constrained embedded network and a client (v_3) that is a participating member of the Internet. Node v_2 has a bridge functionality that seamlessly renders plain-text XML messages into EXI format and vice versa, both in terms of data content and structure. To achieve this, v_2 needs only the same $XSD_{SERVICE_MSG}$ schema description as applied on v_1 to extract the EXI grammar for encoding and decoding.

3.3.5 Implementation

The implementation of our efficient and scalable XML-based Web service generator is written in the Java programming language. The structure of our implementation is synchronized to the three phases of the workflow presented here:

I To read a service description S_D we developed an interface that has to be implemented for a desired description format to retrieve essential information such as which RPCs are served and which messages and data are involved. For the WSDL implementation, we used the JDOM [Hunter, 2001] library to parse the XML-based content. The information determined is then used to build up our analysis table and to construct the $XSD_{SERVICE_MSG}$ file. To build such a valid XSD we also use the JDOM library. To get an abstraction of the $XSD_{SERVICE_MSG}$ we use the XML Schema API provided by the Xerces library [Apache, 2010].

II To construct the EXI grammar and to determine the EXI IDs of all local names and namespaces involved in the $XSD_{SERVICE_MSG}$ we use the de-facto reference implementation of EXI and is called EXIficient [Peintner, 2012]. Our approach extended the implementation with the context-based grammar optimization to later provide the individual automata fragments for the Web service and client side.

III At this point, we provide the developer the opportunity to generate the code in the C, C++, or Java programming language. Dependent on the service description S_D, we only generate the required code components reflecting the service. E.g., if a S_D only defines eventing interactions, only the notifier, databinder, and the EXIProcessor containing the mechanism for that purpose are generated. Our code generator technique uses the facilities of the Eclipse Modeling Framework (EMF). In the case of the EXIProcessor we use the efficient solution that comes from the EXIdizer, which was developed in [Peintner, 2014]. By using this, we only have to pass our context-optimized EXI grammar for the server and client side that restricts the type-based encoding and decoding mechanism .

Service instances generated by us were successfully tested on microcontroller platforms such as the Contiki OS [Dunkels et al., 2004] and Java Micro Edition Connected Limited Device Configuration (CLDC) 1.1 [Oracle, 2009] environment [Käbisch et al., 2010a, Käbisch et al., 2011].

3.4 Evaluation

In Section 3.3.4, we already presented some results from our temperature/humidity Web service example in terms of message size and memory footprint. In this section we are going to evaluate the applicability of our service generator approach for a more complex Web service definition and consider message and memory size used as well as performance. Our main focus is the usage of a generated code in the embedded domain as compared with other known solutions. Therefore, we are going to analyze the performance of our EXI-based and a plain-text XML-based service implementation. For the plain-text variant we are using the well-known gSOAP tool kit (see Section 2.4). As a dataset reference we are using the latest XML-based message specification draft of ISO/IEC 15118 Vehicle-to-Grid Communication Interface (V2G CI) which will shortly introduced here.

3.4.1 The Vehicle-to-Grid Dataset

The ISO/IEC 15118 V2G working group [ISO/IEC, 2012] specifies an XML-based service communication between an *Electrical Vehicle* (EV) and an *Electric Vehicle Supply Equipment* (EVSE) for a charging process. At the time of writing and setting up the evaluation, the V2G standard was in *Draft International Standard* (DIS) status. The ISO/IEC 15118 covers AC-based and DC-based charging. Several data information, such as power parameters, status, and metering information, has to be exchanged. The V2G message structure is used in a similar way as SOAP: the *Header* element carries generic information such as *sessionID* and *notifications*, and a *Body* element that transports the actual message content. Below, we will consider which message patterns are required for a DC-related charging. Every message is requested by the EV (client side), while the EVSE (service side) responds. Those messages are: *SessionSetup, ServiceDiscovery, ServicePaymentSelection, ChargeParameterDiscovery, PowerDelivery, ChargingStatus, CurrentDemand, CableCheck, WeldingDetection, SessionStop,* and *ContractAuthentication*. Besides these messages listed here, further message patterns are defined, which, e.g., correspond to topics such as value added services (VAS) and security. Since these messages are not directly related to a DC charging process, they will not be considered in the following evaluation.

Overall, about 100 different elements are exchanged between EV and EVSE to realize a proper charging process. Thus, there is a need to realize such a communication in a very efficient way, since constrained device units are used in particular on the EV side as well as on the EVSE side. Chapter 6 will further discuss this aspect and provide further details on the V2G

protocol.

3.4.2 Message Size

Figure 3.8 shows the size (in bytes) of the request and response instances of the DC V2G messages for plain-text XML and binary XML format with EXI. The EXI service generated sends messages which are on average around 75 times smaller than the equivalent XML messages. This is one of the key strength of EXI that leads to less network traffic. Furthermore, the opportunity arises to pack a complete message into one data package provided by a network protocol. E.g., the IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [Bormann and Mulligan, 2009] protocol provides, depending on configuration, only a payload size of between 53 (without header compression) and 108 bytes (with header compression). If this is the case, plain-text XML messages have to be distributed in several packages. In wireless sensor networks the risk of losing data packages is usually significant, especially when the number of hop-count is high [Liang et al., 2010]. Then, the likelihood of distorted messages increases proportionally to the number of packages used for transmitting a single message. Even if a reliable data transport is used such as TCP, the increased risk of distorted messages leads to an increase of resubmission efforts.

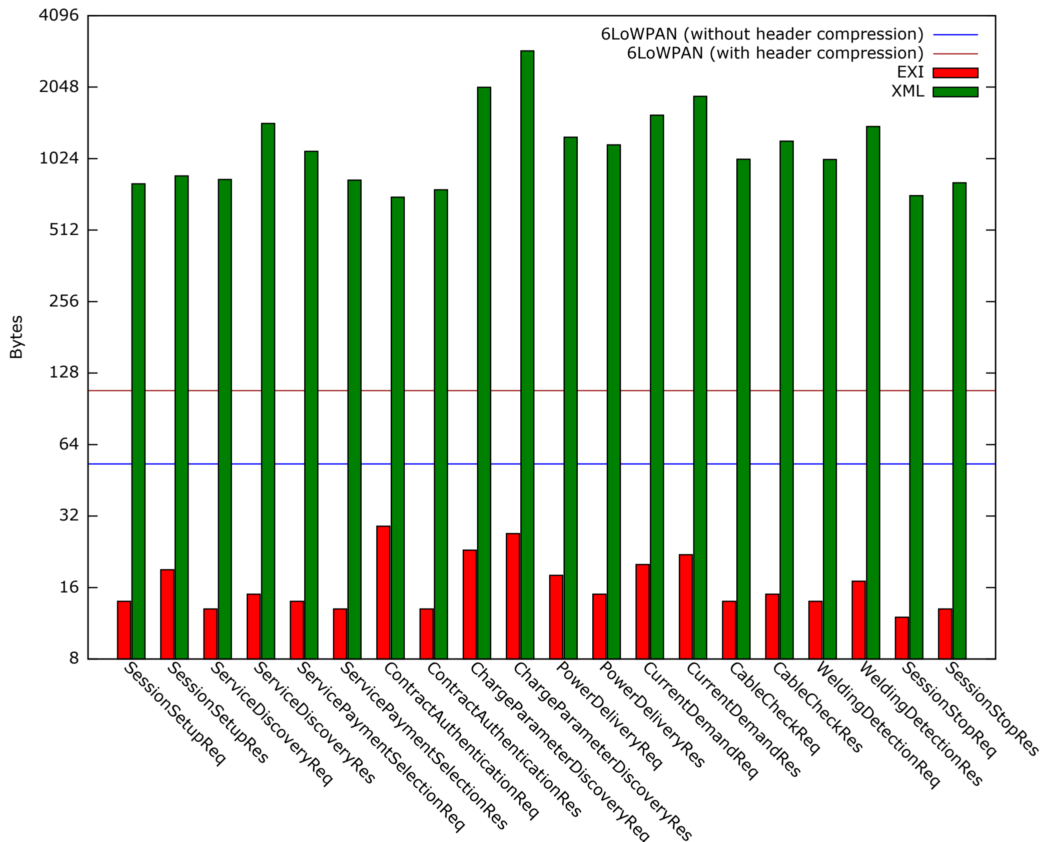


Figure 3.8: V2G request and response messages

3.4.3 Code and Memory Footprint

The code footprint is an important requirement for successfully creating an XML-based service implementation on constrained systems such as microcontrollers. For evaluating the code footprint we compiled the V2G client and service for two different platforms: the aforementioned ARM Cortex-M3 microcontroller (see Section 1.1) with the Contiki OS, and, as reference, for a standard x86 computer system running Linux. Table 3.5 shows the resulting size of the footprint if it is compiled with the gcc with the optimization flag. The table also shows the size of the EXI Web service implementation that does not use the context-bases optimization and operates the full DFA set on each side (EXI WS Full). In addition, as an equivalent plain-text XML Web service variant, we took the gSOAP implementation into account.

It can be seen that the EXI-based Web services are up to 3.5 times smaller in code size than the gSOAP implementations. Furthermore, we are also able to run the code on the ARM Cortex-M3 boards. The size provided here is

the complete image uploaded on the microcontroller. Even if these images contain the Contiki OS with the 6LoWPAN stack (affects around 38kB of flash) and EXI WS V2G implementation, there is still enough space for further program logic. Due to size, gSOAP is completely out of scope on this evaluation board.

Platform	EXI WS		EXI WS Full		gSOAP	
	Client	Server	Client	Server	Client	Server
ARM M3 / contiki	114*	110*	142*	137*	-	-
x86 / linux	175	177	197	229	607	601

Table 3.5: Code footprint (in kBytes). *Includes the Contiki OS with 6LoWPAN stack.

Table 3.6 shows the static RAM usage of the three Web service variants. The results of gSOAP look promising. However, gSOAP allocates memory at runtime, which leads to non-deterministic behavior and estimation in terms of maximum memory usage. In system-critical environments, like the automotive sector, this is not a reasonable implementation. In contrast, *worst-case* memory usage of the EXI Web services is represented in the table, since our implementations works on static memory allocation.

Platform	EXI WS		EXI WS Full		gSOAP	
	Client	Server	Client	Server	Client	Server
ARM M3 / contiki	11.7*	11.8*	11.7*	11.8*	-	-
x86 / linux	11.7	11.8	11.7	11.8	8.0	12.0

Table 3.6: Static RAM usage (in kBytes). *Includes the Contiki OS with 6LoWPAN stack.

As with ROM usage, the Web service generated by our approach leaves enough memory for the actually programming logic.

3.4.4 Processing Speed

To evaluate the performance of the V2G EXI service generated, for comparison, we also used the service implementation based on the gSOAP tool kit. In general, gSOAP is known for generating fast Web services in C and C++ with very low code footprint. As a hardware reference for the performance test we used a more powerful embedded board that comes with an ARMv5 processor that is able to run an embedded Linux distribution

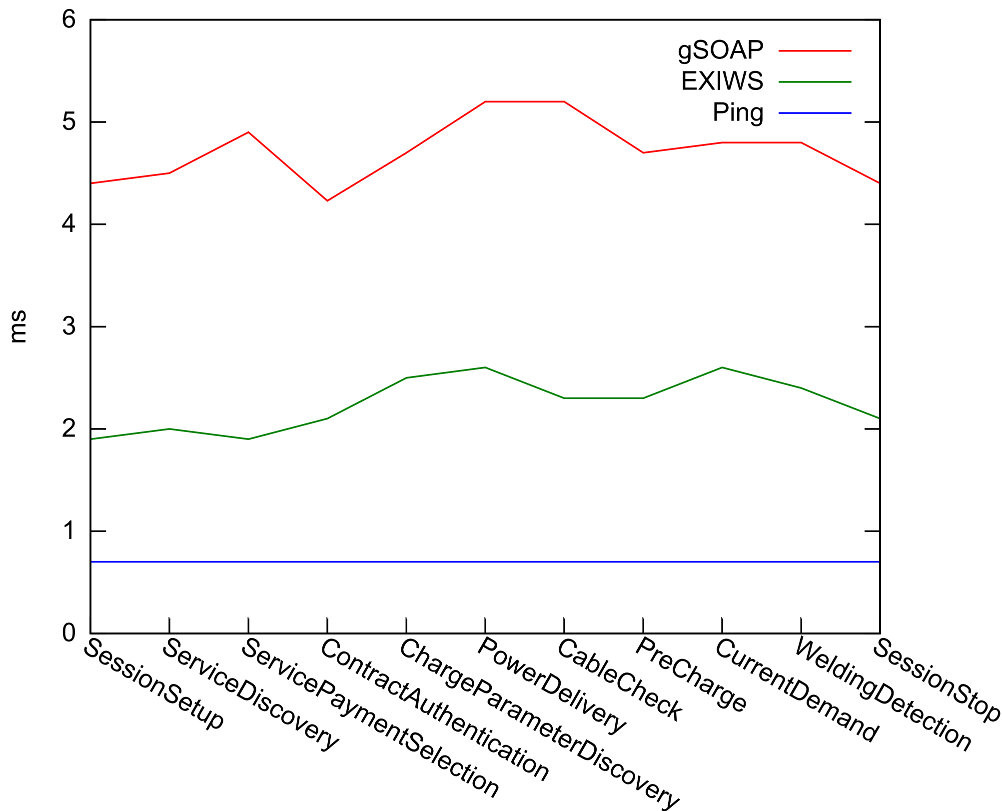


Figure 3.9: Request / Response measurement (roundtrip time)

This board was connected to a consumer PC by an Ethernet Lan cable. To create the same conditions, we used a service description based on WSDL that describes the V2G service and its messages. In addition, we dropped the usage of HTTP and sent the message directly via IP/TCP.

Figure 3.9 shows the result of the request-response time measurement. Each message pattern was sent 100 times: we determined the average time by using the V2G EXI service and gSOAP implementation. As a reference measurement, we determined the average time of a 100 time ping count between the two boards. It can be seen that EXI service performs almost up to 2.5 times faster than the plain XML variant with gSOAP. These numbers show that our Web service approach is profitable for the direct operation of the binary XML level without the need for any type conversion. The plaintext XML approach and its processing overhead is not only affected in the application layer; e.g., the IP/TCP layer has to divide the messages into several packages, which is additionally time consuming. This tends to be reinforced when packages get lost during transmission.

3.4.5 Evaluation Summary

The evaluation in this chapter shows that the usage of XML-based Web services is feasible in a very efficient manner in the embedded domain with constrained devices such as microcontrollers. EXI enables fast processing of service messages compared to the plain-text XML variant and reduces network traffic immensely. Even in complex services such as V2G, the code footprints are very small for a client and server side implementation.

3.5 Related Work

Before we conclude this chapter, we are going to discuss related works in terms of binary XML and Web service focused on embedded devices.

3.5.1 Binary XML

In the context of XML compression, we already introduced two former coder alternatives to EXI earlier in this chapter; these can be distinguished between structure-less or generic (e.g. ZIP) and structure-based compression (e.g. BiM) (see Section 3.2.1). We also want to mention two other important variants which are also often associated with each other in the context of binary XML, namely Abstract Syntax Notation One (*ASN.1*) as a schema-informed variant and *Fast Infoset* as a variant able to encode schema-less.

Abstract Syntax Notation One (ASN.1) is standardized by the International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), and International Telecommunication Union (ITU) [ITU, 2002]. This standard provides language to describe data types in an abstract manner, which can then be applied to different *encoding rules* to represent the data on bit-level. In the case of the ASN.1 *Basic Encoding Rules* (BER) [International Telecommunication Union, 2002a], the data is encoded by starting with a type identifier, followed by a length description, and then the actual data value, (a.k.a. type-length-value (TLV) transformation). Meanwhile, ASN.1 *Packed Encoding Rules* (PER) [International Telecommunication Union, 2002b] provide a much more compact and binary encoding with a set of data type support. For XML, ASN.1 provides two kinds of encoding rules: The ASN.1 *XML Encoding Rules* (XER) [International Telecommunication Union, 2001] standardizes conversion between ASN.1 and XML, and the *Mappings from XML schemas to ASN.1* [International Telecommunication Union, 2004] defines a mapping from XML Schema declarations to the ASN.1 notation. The latter sounds very promising, since

having the ASN.1 notation present enables us to use, e.g., the PER mechanism to bring the XML-based data in binary representation. These mapping rules to the ASN.1 notation are more or less comparable to the rules governing EXI grammar construction of an XML Schema. However, the rich set of declaration varieties provided by XML Schema, such as data type restrictions, are not fully covered or possible by ASN.1. In general, ASN.1 PEV provides good compression results [Scholz, 2011] but there are also cases in which it led to the same or larger document size as the plain-text XML document [Bournez, 2009b]. To provide full flexibility to developers in terms of data modeling in their Web services and to avoid the side effects as presented in [Bournez, 2009b], ASN.1 encoding rules are not suitable solutions for use in standardized Web services.

The Fast Infoset (FI) [ITU, 2005] is also a standard of ITU that relies on the syntax notation of ASN.1 and provides an alternative encoding format for the XML Infoset (see Section 3.2.2). Basically, a vocabulary table is used to index strings like element/attribute names. When first occurring in the document, the full name or value is present in the FI stream. Redundant elements/attributes and values are represented by the vocabulary table index. Consequently, names and values only appear once. Highly redundant XML messages in terms of element/attribute names and values will achieve a better compression ratio than messages that contain unique information at any point. As an alternative, the FI standard describes the possibility of providing references in the FI header to pre-calculated tables. This avoids the construction overhead of the vocabulary tables at runtime and would further reduce the binary stream since only indices numbers are exclusively used.

In general, the basic philosophy of FI is comparable to EXI schema-less coding (see Section 3.2.2). However, not taking the structure knowledge into account, which is provided by an underlying XML Schema, leads to worse compression results when compared to EXI [Scholz, 2011] and would have negative effects in terms of network traffic and type conversion overheads.

3.5.2 Web Services for Embedded Devices

A large number of Web service toolkits are available for different programming languages and platforms. Based on the existing implementations and available research we can distinguish between highly configurable solutions that are targeting more (multi-core) powerful systems such as Apache Axis [Perera et al., 2006] for C/C++ and Java, and those for embedded environments

which we will mainly discuss in this section.

A prominent generation toolkit is gSOAP [van Engelen and Gallivan, 2002], which was also used in our evaluation. A few gSOAP-based code generation techniques for Web services for embedded devices can be found in [van Engelen, 2004]. Based on our evaluation, however, constrained device classes such as the ARM Cortex-M3 microcontroller cannot deal with the code footprint of gSOAP or even additional libraries such as proposed in [van Engelen, 2004]. In addition, using plain-text XML would harm network traffic and would increase the risk of package loss, especially when it comes to usage in constrained embedded networks with wireless low-power communication.

In [Lerche et al., 2011] a Web Service prototype implementation is presented that reflects a subset of DPWS (see Section 2.5.2) and is called *uDPWS*. The number presented are very promising: 10kBytes of RAM and 3kBytes of RAM compiled for a TI MSP430 microcontroller. However, this solution is also based on the usage of plain-text XML, which has to handle message sizes of about 700 Bytes, for instance. This negatively affects the efficient usage of 6LoWPAN in terms of message fragmentation, as discussed in Section 3.4.2, and processing and transmitting time, which was also noticed in [Lerche et al., 2011]. In addition, it is unclear whether there is a generic solution beyond DPWS and toolkit support for code generation of arbitrary Web Services similar to our approach presented in this chapter.

A model-driven approach is used by the ϵSOA project (see Section 2.5.1) for generating the *ϵServices* which are deployed in embedded network. In addition, to overcome the drawbacks of plain-text XML, ϵServices use binary XML with EXI as a data exchange format. This data, however, is fairly simple, using only a flat structure consisting of one data item (e.g., one sensor value) in the most cases. Information such as data context is not modeled within the XML-based data. This information is retrieved in the transport protocol beforehand. In our approach, we consider the standardized Web service with SOAP, which is able to provide data/operation context within the messages. Thus, our dispatcher generated operates on binary XML to identify the message context and is independent of any underlying transportation protocols used. Furthermore, we do not make any restrictions on XML service data modeling (excepting min/maxOccurs properties, nested element declarations, choice grouping, etc.) and we are able to efficiently handle relatively complex data structures such as the V2G data set. This is realized, among other things, by our investigation of context-based EXI grammar optimization, which leads to a reduction of code size and thus re-

quires less memory usage on microcontrollers.

Concepts for realizing small REST-based Web Services come with the Constrained Application Protocol (CoAP) [Shelby et al., 2013] by the IETF (also see Section 2.5.3). In general, REST is limited to HTTP, which has drawbacks in terms of processing, memory consumption, and bandwidth usage. CoAP is a binary alternative that uses the methods of HTTP and provides a mapping between CoAP and HTTP. Using CoAP to realize REST-based Web services with binary XML as a data exchange format would be an efficient prospect for the restricted embedded environment. However, at the time of writing this thesis, there is a lack concerning an automatized generation tool that takes a service description such as WSDL or WADL and generates REST-based Web services based on CoAP.

3.6 Summary

In this chapter we presented an innovative approach for generating a source code for developing XML-based Web services for small embedded devices with constrained resources. The generator enables the seamless adoption and usage of wide-spread, standardized Web service protocols in embedded networks to realize SOA-based applications. Based on previous knowledge that comes with the service descriptions provided, optimized EXI grammar sets were determined for the service and client contexts. This resulted in a more compact source code, which only contains the particular service and client side functionality needed for creating and interpreting service messages. Using this Web service generator eases the development of applications for the embedded domain and simplifies the integration of Web services into other networks. Evaluation results based on the ISO/IEC 15118 V2G dataset proved the efficiency by demonstrating a very low code footprint and high performance.

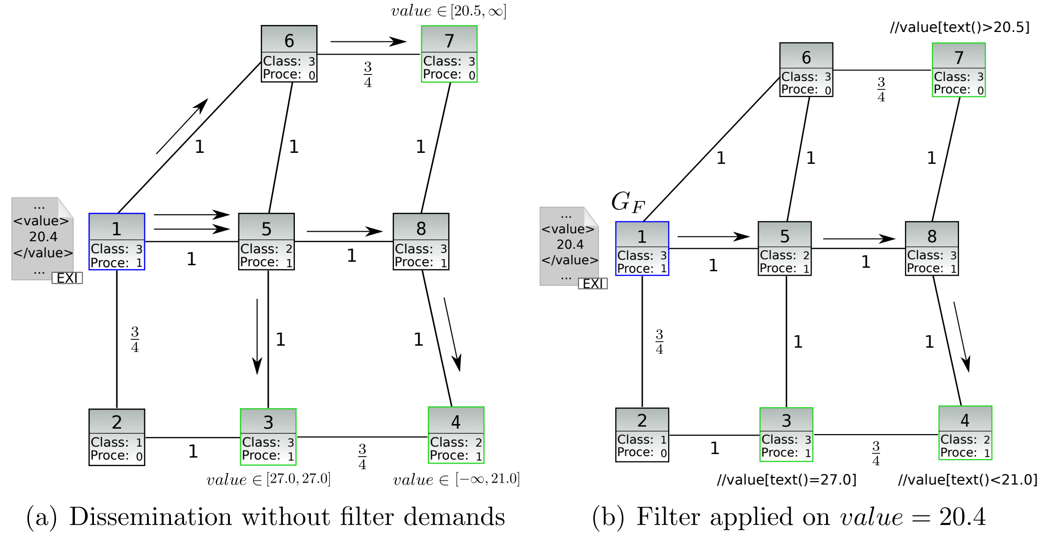
Chapter 4

Filter-Enabled Service Communication

4.1 Introduction

By using XML-based communication, as initiated by standardized Web services, it is possible to create interoperable service communication between heterogeneous systems. Our approach, presented in the previous chapter, shows that this communication philosophy with its standardized interface description and message framework for interaction can be adopted for the domain of constrained embedded networks. It eases the development of applications for embedded networks and does not require investigations into any internal implementation behind a given Web service.

Similar to distributed systems, a high number of installed applications and frequent data interaction negatively influence network resources. This holds especially true for the domain of embedded networks, which consist of microcontrollers with low processing, memory, and bandwidth capabilities. Figure 4.1(a) shows an example of an embedded network consisting of 8 nodes, which are represented by 3 different device classes (see Section 1.1). A service provider has been installed at node 1, which frequently samples data (e.g., in every millisecond) and transmits it separately, independent of message content, to each of the three client subscribers (nodes 3, 4, and 7). A number of very constrained class 3 devices (e.g., node 6 and 8) are involved in this delivery process (shown as arrows) in terms of routing; in addition, a relatively bad connection link is used, such as the one between nodes 6 and 7, which may reflect a delay or a non-stable physical link. In turn, the latter aspect may result in resubmissions of messages which would, among other things, reduce application responsiveness and affect the network resources



(a) Dissemination without filter demands (b) Filter applied on $value = 20.4$

Figure 4.1: Service data dissemination example: Nodes 3, 4, and 7 subscribe to service data (*value*) of node 1.

negatively.

A more sophisticated approach would involve using pre-knowledge of client conditions on the service data requested and to provide data only to those clients that fulfill these conditions. Such conditions could be the presence of particular data or particular value ranges that have to be met. In Figure 4.1(a), there are some conditions given at the client nodes for the *value* element that is provided by the service provider. Actually, the service data ($value = 20.4$) currently provided by node 1 is only relevant to node 4 since this node is interested in values which are smaller than 21.0. To avoid transmitting to and processing of this message at client nodes 3 and 7, it would be desirable to install a filter-mechanism at the service data origin node to evaluate relevance for all clients. Figure 4.1(b) depicts this filter mechanism, which is denoted as G_F at node 1. Service data is only transmitted to node 4. Such an approach reduces network traffic and avoids unnecessary message processing by other service requesters (e.g., node 3 and 7) as well as the intermedia routing nodes (e.g., node 6).

In this chapter we are going to present efficient approaches for constructing filter-enabled subscribe mechanisms in constrained embedded networks with service-based communication using binary XML with EXI. The filter mechanisms are constructed by using the well-known XML Path Language (XPath) [XPath2.0, 2007], which is provided by the service requesters to address the data interests and conditions (e.g., value ranges). The client nodes

in Figure 4.1(b) show sample XPath expressions based on our scenario. Our approaches can be implemented within Web services to realize similar functionality such as WS-Eventing [Malhotra et al., 2009] or beyond such as for dedicated publish/subscribe brokers. In Section 4.2, we introduce briefly the XPath query language and explain in which form we are using it in this thesis. In subsequent sections, we present our filtering approaches starting with *BasicEXIFiltering* (Section 4.3) and followed by *OptimizedEXIFiltering* (Section 4.4). In Section 4.5, we present performance results of our solutions, as compared to YFilter [Diao and Franklin, 2003] as a representative of a filtering method based on plain-text XML. Furthermore, a demo setup of an embedded network will provide data on memory usage by these approaches. We conclude with a 'related works' section (Section 4.6) and a summary (Section 4.7).

4.2 Querying with XPath Expressions

In the beginning of XML, a significant query technique was standardized by the W3C that enables extracting particular information from or testing the relevance of XML instances: the XML Path Language (XPath) [Clark and DeRose, 1999]. XPath became a basis for emerging W3C standards such as XSL Transformations (XSLT) [XSLT1.0, 1999], XML Pointer Language (XPointer) [Grosso et al., 2003], XQuery [Clark and DeRose, 1999], or XForms [Boyer, 2007]. XPath 2.0 [XPath2.0, 2007] is the current version of the query language and provides additional functionality as well as language concepts compared to the 1.0 version. At the time of writing, the W3C is working on the next generation of XPath, namely XPath 3.0 [Robie et al., 2011]. It will have additional features, such as support of a richer set of data types. Below, we only concentrate on the basic functionality of XPath already extant in version 1.0.

An XPath expression addresses parts of an XML document. Thereby, the XML document is seen as a tree structure: The *nodes* represent elements, attributes, text, namespace, processing-instruction, comment, or document nodes. The root of the tree is always represented by the root element. The formal notation of an XPath expression is based on one or more *location steps*. Each step follows the rule

$$axis :: node - test[predicate]$$

and is separated by the sign '/', which is also called as a *forward slash*. The *axis* defines the navigation step relative to the current node. The following axes are possible: *ancestor, ancestor-or-self, attribute, child, descendant,*

Full expression	Abbreviated
attribute::	@
child::	
descendant-or-self	//
self::	.
parent::	..

Table 4.1: Abbreviated syntax for XPath expressions

descendant-or-self, following, following-sibling, namespace, parent, preceding, preceding-sibling, and self. The *node-test* identifies one or more nodes within the axis. This can directly address the name (with namespace) of a node or select all kinds of nodes by using the wildcard sign '*'. Alternatively, functions can be used to select specific nodes, e.g., text nodes with *text()* and comment nodes with *comment()*. Zero ore more *predicates* enable a refinement of the node set selected. Predicates are nested in the squared brackets and can access a high number of different math operators (comparison, logical, and numerical) as well as functions such as string-based functions (e.g., *substring()* and *string-length()*) or node-set-based functions (e.g., *count()* and *name()*).

Based on the rich facets of XPath, in this thesis we limited the XPath query examples to expressions able to be represented by the abbreviated syntax shown in Table 4.1. Furthermore, we will limit the operators of the predicates to the comparison functions ($' <$', $' >'$, and $' ='$) and those are applied only once to any attributes and/or text value nodes addressed. Nested XPath expressions that can be defined within predicates will also not be considered here. If used, the descendant-or-self axis will only be applied in the first location step of an XPath expression. All of our approaches, however, can be extended to support such features.

4.3 Basic Binary XML Filtering

This first approach is based on the idea of working on the top of a given EXI grammar and using StAX events (see section 2.3.2) which are sequentially triggered during an EXI decoding process. To evaluate an input message for cases in which one or more provided queries match, we relied on the idea of the string matching algorithm given by the the Knuth-Morris-Pratt algorithm [Knuth et al., 1977]. We will start by introducing the idea behind this algorithm and will build the bridge to XML-based data structure as well as to XPath expressions. For an efficient evaluation in a binary XML environ-

ment, we will introduce a possibility for transforming XPath expressions into a more processable format in Section 4.3.2. The subsequent subsection will present our *BasicEXIFilter* algorithm for evaluating binary XML messages based on relevance to a set of XPath queries.

4.3.1 Knuth-Morris-Pratt Algorithm, XML, and XPath

The Knuth-Morris-Pratt algorithm is known as a fast pattern matching in strings whose complexity is linear with $\mathcal{O}(n)$ [Knuth et al., 1977]. Here, n denotes the length of the string on which the test pattern is applied.

Compared to other string-matching algorithms, such as the naive pattern-matching algorithm [Cormen et al., 2009], the Knuth-Morris-Pratt algorithm avoids testing useless shifts of the pattern that would logically fail due to knowledge of the characters scanned. The position shifts are based on a predetermined π function for a given pattern. In the literature, this function is typically called the *prefix function* (sometimes also known as *failure function*). Now let us define what is understand to be the *prefix* as well as its complementary *suffix* in the context of string pattern matching [Cormen et al., 2009]:

Definition 4.1 (Prefix and Suffix)
A string p is a prefix of string s, if $s = px$ for some string x. A string p is a suffix of string s, if $s = xp$ for some string x. The empty string ϵ is both a prefix and a suffix of every string.

Prefix examples of the pattern 'abcabd' would be 'ab' and 'abc'. Strings 'abd' and 'bd' would be valid suffix examples. The prefix function π function for a pattern encapsulates knowledge about how the pattern matches against shifts of itself [Cormen et al., 2009]. Let us consider a matching example where the 6th character ($T[5]$) of the test string 'abcabcabda' has a mismatch with the pattern above with the last character position $P[5]$:

P: abcabd
T: abcabcabda

It can be seen that the first five text characters (index compare position $j = 4$) from the text shift position $m = 0$ match the pattern: $P[0..j] = T[m..(m + j)]$. Taking these successful matches into account the shifts will be performed in a way that would not result in a mismatch again. Doing one or two shifts would be invalid since the first characters would not match. However, doing three shifts to the right we would be more successful and the

first three pattern characters match with three text characters. Hence, we know that $P[0..j]$ is a suffix of $T[0..(m+j)]$, and therefore, we are looking for the longest proper prefix $P[0..i]$ of $P[0..j]$ that is also a suffix of $T[0..(m+j)]$ for some $i < j$. The difference $(j - i)$ is added to m to determine the new shift:

$$m = m + (j - i)$$

Mapping the formalization for the example above, it can be seen that the pattern $P[0..4]$='abcab' is a suffix of $T[0..(0 + 4)] = T[0..4]$='abcab'. The longest proper prefix of $P[0..4]$='abcab' is $P[0..1]$='ab', this is also the suffix of $T[0..4]$='abcab'. Thus, the overall shift for the pattern as related to the input text is determined by $m = 0 + (4 - 1) = 3$ which in this case leads to a string pattern match:

P: abcabd
T: abcabcabda

The prefix function π provides the longest proper prefix for the current match progress given by index j. The following definition gives a formal description of this function [Cormen et al., 2009]:

Definition 4.2 (Prefix Function π)
Given a pattern $P[0..n]$, the prefix function for the pattern P is the function $\pi : \{0, 1, .., n\} \rightarrow \{-1, 0, .., n - 2\}$ so that

$$\pi[j] = max\{i : i < j \wedge P[0..i] \text{ is suffix of } P[0..j]\}$$

Algorithm 4.1 determines such a prefix function π for a given pattern P [Cormen et al., 2009]. The algorithm iterates each character (line 4), except the first one, of the (reference) pattern $P[j]$ and checks whether there is a mismatch (line 5) or a match (line 8) with the character of the (aligned) pattern $P[i + 1]$. If there is a match, the current pattern match progress variable i (note, $i = -1$ does not refer to a current match) is saved in π (lines 9 and 11). In other words, i represents the longest proper prefix for the (sub-)pattern $P[0..j]$ that was tested. If there is a mismatch, the test pattern will be realigned in reverse until a match occurs again (lines 5-7). If no match can be found in that step, then -1 is deposited.

Applying pattern 'abcabd' to the algorithm, we discover the following:

Algorithm 4.1 $DeterminePrefixFunction(P)$

Input: Pattern P
Output: π

```
1: n ← |P| − 1;
2: π[0] ← −1;
3: i ← −1;
4: for j ← 1 to n do
5:    while i ≥ 0 and P[i + 1] ≠ P[j] do
6:       i ← π[i];
7:    end while
8:    if P[i + 1] == P[j] then
9:       i ← i + 1;
10:   end if
11:   π[j] ← i;
12: end for
```

Position j	0	1	2	3	4	5
P:	a	b	c	a	b	d
$\pi[0] = -1$						
$\pi[1] = -1$						
$\pi[2] = -1$						
$\pi[3] = 0$				a		
$\pi[4] = 1$				a	b	
$\pi[5] = -1$						

That means that the longest prefixes, which are also suffixes, can only be found for sub-patterns $P[0..3]$ and $P[0..4]$. Putting everything together, we can present the Knuth-Morris-Pratt algorithm (see Algorithm 4.2) [Cormen et al., 2009].

As can be seen, the $KMPStringMatcher$ does not differ much from the $DeterminePrefixFunction$ algorithm: before the text is scanned, the prefix function π is determined (line 3). Each character in T (line 5) will be checked as to whether there is a match (line 9) or a mismatch (line 6) with P. In the case of a mismatch, it will be determined how many shifts of P have to be done, which is determined by prefix function π (line 7). The pattern P is found in T when the last index position i has been reached (line 12). We will not discuss correctness and complexity here; however, this analysis can be found in [Cormen et al., 2009].

Now let us explain how the idea of a string matching algorithm, such as the Knuth-Morris-Pratt algorithm, can help us evaluate whether there is

Algorithm 4.2 $KMPStringMatcher(T, P)$

Input: Text T and searched pattern P
Output: True, if pattern P found in T, otherwise false.

```
1: n ← |T| − 1;
2: l ← |P| − 1;
3: π ← DeterminePrefixFunction(P);
4: i ← −1;
5: for j ← 0 to n do
6:    while i ≥ 0 and P[i + 1] ≠ T[j] do
7:       i ← π[i];
8:    end while
9:    if P[i + 1] == T[j] then
10:      i ← i + 1;
11:   end if
12:   if i == l then
13:      return true;
14:   end if
15: end for
16: return false;
```

a match of an XPath query applied on an XML message or not: first, let us assume that there is an XPath query $//Header/status$. In other words, we are interested in the *status* element that strictly follows the *Header* path, arbitrary located in the message as parent node. Now let us consider an XML message (see Listing 2.9) that starts with <Envelope><Header><status> LowBattery</status>... . To apply the Knuth-Morris-Pratt algorithm, the XPath query will be seen as the input pattern; however, the separator '/' has been removed from within the XPath expression, so that only the names of node elements remain. Similarly, the input XML message is seen as a sequence of element and attribute names with their values. Start and end tags are removed. As can be easily observed, applying the Knuth-Morris-Pratt algorithm with the following input would lead to a successful match when shifting the pattern to the right (word by word or character by character):

P: Header status
T: Envelope Header status LowBattery ...

If using StAX[1] as XML parser, each *startElement* event encountered causes a comparison of the current entry node of the XPath expression.

[1]As alternative to StAX, SAX events can be used in a similar manner [Grust and

An *endElement* event, which exit the current tree level in the XML message structure, would result in a decrement of the comparable position of the XPath node entry. Consequently, a query match is only observed if the last XPath node is observed and matches the corresponding message content. As we know, XPath has a rich set of facets, such as wildcards, descendent-or-self, and predicates for filter refinements (based on attributes and/or value nodes). The next subsection addresses special cases of XPath expressions and in general, how can we transform XPath expression in a more efficient representation. The requirements defined there will support us in realizing an XML-based filter mechanism based on the philosophy of the Knuth-Morris-Pratt algorithm applicable in the embedded domain.

4.3.2 XPath Normaliziation

To apply XML-based documents and queries to an evaluation mechanism that follows string-matching techniques, we require a conversation that allows content comparison but neglects syntax rules such as start and end tags. This way, structure information has to still be taken into account so that we can guarantee to match the original purpose of either the XML-based document or XPath expression.

First of all, we consider the XML-based representation using binary XML format with EXI. In Section 3.2 we presented the EXI mechanism that generates EXI grammars based on a given XML schema. Furthermore, to avoid the processing overhead resulting from string comparisons of elements and namespace names, we use unique qualified identifiers, such as the EXI IDs, which are based on integer numbers (see Section 3.2.4). We benefit from these IDs and an XML-based document becomes a sequence of number pairs in the first consideration. For example, the XML fragment <SOAP-ENV:Envelope><SOAP-ENV:Header><es:status>LowBattery</es:status> ... is transformed into (1, 5) (2, 5) (9, 4) 'LowBattery' ... (see Table 3.4). It should be noted that these numbers are unique to the underlying XML schema. Thus, transformation back into a plain-text XML representation is only possible if the associated XML schema is known.

We pursue a similar strategy for XPath expressions. This means that for an XPath query such as */Envelope/Header/status* we transform each node name in the query into the unique EXI identification numbers: (1, 5)/(2, 5)/(9, 4). Removing the separator, we simply have: (1, 5) (2, 5) (9, 4). The tuple pairs consisting of EXI IDs for the local name and namespace are already a compact representations as compared to the plain-text representa-

Teubner, 2002]

tion. Since we know, that these IDs are unique in an XML schema context, we are able to further reduce this representation and map the double number to a single number representation. This process is quite simple since we can use the entry number of the ID table as a unique representation. For example, the *Envelope* with the namespace *http://www.w3.org/2003/05/soap-envelope* has the EXI ID pair $(1, 5)$ and has the entry position 10 in Table 3.4. Since such a entry number can be uniquely computed and associated (EXI sorts IDs by namespace and local name), we can use the table entry ID to represent the XPath expression. Thus, pairs (1, 5) (2, 5) (9, 4) can also be represented as 10 11 6. The same strategy can also be applied to XML messages. Instead of EXI ID pairs, the table entry ID can be used for a more compact and efficient processable representation of binary XML content.

Typically, an XPath expression also may consist of different facets such as wildcards, descendant-or-self, and predicates conditions (see Section 4.2). Each facet will now be considered and its conversation rule for the normalization representation is provided. The examples shown are based on ID Table 3.4.

Wildcard
Wildcards in XPath expressions can be used to select unknown XML elements. Since this expression can map multiple elements at the same time, a single EXI ID representation is not possible. Based on this fact, we will transform each wildcard in the XPath expression into a number with the value -1. E.g., the XPath query $/*/*/Temperature/value$ is transformed into -1 -1 4 8. Thus, the value -1 does not represent a particular element and namespace; however, it signals the acceptance of an arbitrary element at this location level.

Descendant-or-self
The descendant-or-self expression selects all descendants (children, grandchildren, etc.) of the current node and the current node itself. To use this expression in an absolute location of an XPath expression, we will employ the identifier with the value -2 in the normaliziation. E.g., the query $//status$ will be represented as -2 6.

Predicate
Predicates are used to refine the selected node set within the expressions. If a predicate is used in an XPath expression, we will signal this with the value -3. The subsequent EXI ID number addresses the element or attribute being evaluated at this point. The operator will also be represented by a number,

starting with -10 and moving downwards: -10 signalizes ’=’, -11 signalizes ’$<$’, and -12 refers to ’$>$’. If desired, we can expand the set of operators very easily. Since we know the data model’s underlying XML schema, the comparison value will be represented type aware.

For example, the XPath query $//Temperature[scale =' Celsius']/value$ is transformed into -2 4 -3 5 -10 ’Celsius’ 8.

All put together, we can formalize a normalized XPath query as follows.

Definition 4.3 (Normalized XPath Qurey Q^N)
A $Q^N[0..n]$ pattern is normalized when seamless order is observed as the original XPath expression and the transformed Q^N reflect one or more of the equivalent ID numbers: (table) entry ID (dependent on the underlying XML Schema), wildcard (-1), descendant-or-self ID (-2), predicate (-3), operations (-10 for ’=’,-11 for ’$<$’,-12 for ’$>$’), and the predicate test value.

The XPathNormalizer Algorithm 4.3 as a pseudo code provides us with the normalized representation Q^N of a given XPath query Q and the underlying XML schema XSD that defines the data structure requested.

The algorithm is straightforward. Please note that the input XPath expression is separated by the separator ’/’ and it is seen as an array of element/attribute names with predicate instructions (see the previous subsection). Thus, an descendant-or-self axis will be signalize by an empty array entry by Q which will then signalized by -2 in Q^N (lines 6-7). For an expression with a predicate, we are following the predicate pattern $nodeName$[*predicateNodeName OP value*] and are assigning the identifier in the Q^N vector (lines 8-19). For predicates that test element values with the XPath function *text*(), the *getPredicateNodeName* function (see line 11) would return the ID of the current node level. The ID entry function $IDEntry$ (e.g., lines 9 and 11) serves the entry ID of the EXI ID table that has been uniquely constructed by EXI rules based on the given XSD.

Algorithm 4.3 $XPathNormalizer(Q, XSD)$

Input: XPath expression Q as array seperated by '/' and an XML schema XSD.
Output: Normalized XPath expression Q^N

1: $n \leftarrow |Q| - 1;$
2: $i \leftarrow 0;$
3: **for** $j \leftarrow 0$ to n **do**
4:   **if** $Q[j] ==$'*' **then**
5:     $Q^N[i++] \leftarrow -1;$
6:   **else if** $Q[j] ==$'' and $j \neq 0$ **then**
7:     $Q^N[i++] \leftarrow -2;$
8:   **else if** $Q[j].containsCharacter('[')$ **then**
9:     $Q^N[i++] \leftarrow IDEntry(Q[j].getNodeName(), XSD);$
10:     $Q^N[i++] \leftarrow -3;$
11:     $Q^N[i++] \leftarrow IDEntry(Q[j].getPredicateNodeName(), XSD);$
12:     **if** $Q[j].containsCharacter('=')$ **then**
13:       $Q^N[i++] \leftarrow -10;$
14:     **else if** $Q[j].containsCharacter('<')$ **then**
15:       $Q^N[i++] \leftarrow -11;$
16:     **else if** $Q[j].containsCharacter('>')$ **then**
17:       $Q^N[i++] \leftarrow -12;$
18:     **end if**
19:     $Q^N[i++] \leftarrow Q[j].extractCompareValue();$
20:   **else**
21:     $Q^N[i++] \leftarrow IDEntry(Q[j], XSD);$
22:   **end if**
23: **end for**
24: **return** $Q^N;$

4.3.3 The *BasicEXIFilter* Algorithm

In this section we present the basic EXI filter mechanism for efficiently evaluating one or more XPath queries. The philosophy is based on the Knuth-Morris-Pratt algorithm; however, we modified and optimized the mechanism for the binary XML context. Furthermore, we will take advantage of the prefix function π (see Definition 4.2) to avoid unnecessary checks during evaluation, and hence speed up the process of finding a query match. The prefix function is determined on a normalized XPath query Q^N (see Definition 4.3) that will be similarly processed as a text pattern. The main difference to the Knuth- Morris-Pratt string matching algorithm is the usage of StAX events, based on EXI grammars, to parse the input EXI message. Algorithm 4.4 presents the $BasicEXIFilter$ mechanism as a pseudo code that takes a binary XML message M^{EXI} and a number of XPath queries $Q = \{Q_1, Q_2, ...Q_n\}$. The outcome is a set R with $R \subseteq Q$ that identifies queries with successful matches in M^{EXI}.

The first two major steps involve bringing each XPath query into normalized representation (line 1) and to determine the prefix function for each query (line 2). After pre-processing, we can run the evaluation. In general, the XPath evaluation process is simple: based on the StAX events that are identified by reading the EXI stream, we keep track of the current step i of each given XPath query. If a *startElement* event (lines 7-18) matches the next step of one or more of the queries, the current step index of all XPath queries affected is incremented (lines 14-15). If there is a mismatch, the mechanism falls back to the last step in the corresponding query that matches the entry IDs and tests once more if a match was found (lines 11-12). If the last location step of an XPath expression is reached, which signals a complete query match, we remove this query from the active working query set Q (lines 17-18).

The endElement StAX event (lines 21-23) causes a decrement of the current node / tree level l (line 22). In general, the tree level l indicator provides the current structure depth of the current input message relative to the root node/element. Thus, a *startElement* event would increment the level depth (see line 8). Within the *endElement* event, each query is checked as to whether the level l has dropped the relative location step of each query. The function *checkDecrement()* checks this case (line 23) and if found to be true, the location step of the query will be decremented by *checkDecrement()*.

The processing steps within the *attribute* (lines 24-27) and *character* (lines 28-30) events are almost identical. Both get their meaning when predicates are used within the XPath expressions. The value embedded in the EXI stream is gathered by the *getValue()* (line 25 and 29). For the *attribute*

event, we will also identify the entry EXI ID value of this attribute (line 26). This is not required for the *character* event, since we are still on the element level most recently discovered ,where its ID is still present. For each query we also call the *PredicateTest()* procedure, which is explained below.

The evaluation process continues until each XPath query matches the messages, or until the *endDocument* event is encountered, which means that at least one query in Q has a mismatch. The result set R returned carries all queries that match the input EXI message M^{EXI}.

The *PredicateTest* checks whether the given query Q^N has a predicate at the corresponding location step and whether the node addressed (element text node or attribute) matches the current entry *id* (line 2). Lines 3-8 will determine which comparison operation ('=', '<', and '>') is used and test the value within the query against value v from the stream. If the test was positive the index i of the query will be increased by 4 (skip the predicate definition) to check the next location step in the query (lines 10-11). If the last location step is reached, the query will be registered as a match by removing from active query set Q (lines 13-14).

The complexity is determined very quickly: For each StAX event (except the *startDocument* and *endDocument* events) each query in Q^N is checked for a local ID match or predicate fulfillment. Let us assume that n is the total number of StAX events which occur during the parsing process of an input EXI stream. Since we do not take any action for the *startDocument* and *endDocument* events, the total number will be $n - 2$. Furthermore, for each event all queries in Q^N will be checked locally. Thereby, the number of queries in Q^N is given by m. Consequently, the complexity is given by $\mathcal{O}((n-2)m) = \mathcal{O}(nm)$. Thus, the complexity of the*BasicEXIFiltering* mechanism is influenced by the number of StAX events encountered by the input EXI message stream and the number of quires that have to be evaluated. If only one query is given, the complexity determined matches the complexity of Knuth-Morris-Pratt algorithm (see Section 4.3.1).

Algorithm 4.4 *BasicEXIFilter*$(M^{EXI}, \{Q_1, Q_2, ...Q_n\})$

Input: EXI message M^{EXI} and a set of queries $Q \leftarrow \{Q_1, Q_2, ...Q_n\}$
Output: Subset $R \subseteq Q$ that match M^{EXI}

1: Apply each query in Q to $XPathNormalizer()$ to get normalized XPath representation $Q^N \leftarrow \{Q_1^N, Q_2^N, ...Q_n^N\}$
2: Apply each query in Q^N to $DeterminePrefixFunction()$ to get the prefix functions $\pi_1, \pi_2, ..., \pi_n$
3: Assign for each query in Q^N an index i initialized with $i_1 \leftarrow -1$, $i_2 \leftarrow -1$, ... $i_n \leftarrow -1$
4: $l \leftarrow 0$; $id \leftarrow 0$; $v \leftarrow \emptyset$; $R \leftarrow Q$;
5: **while** $(event \leftarrow nextEvent()) \neq END_DOCUMENT$ or $Q \neq \emptyset$) **do**
6:   **switch** (event)
7:   **case** $START_ELEMENT$**:**
8:     $l \leftarrow l + 1$
9:     $id \leftarrow entryID()$;
10:     **for** $j \leftarrow 1$ to n and $Q_j \in Q$ **do**
11:       **while** $i_j \geq 0$ and $Q_j^N[0] == -2$ and $(Q_j^N[i_j + 1] \neq id$ or $Q_j^N[i_j + 1] \neq -1)$ **do**
12:         $i_j \leftarrow \pi_j[i_j]$;
13:       **end while**
14:       **if** $Q_j^N[i_j + 1] == id$ or $Q_j^N[i_j + 1] == -1$ **then**
15:         $i_j \leftarrow i_j + 1$;
16:       **end if**
17:       **if** $i_j == |Q_j^N|$ **then**
18:         $Q \leftarrow Q \setminus Q_j$;
19:       **end if**
20:     **end for**
21:   **case** $END_ELEMENT$**:**
22:     $l \leftarrow l - 1$;
23:     For each $Q_j \in Q$ call $checkDecrement(l, i_j)$
24:   **case** $ATTRIBUTE$**:**
25:     $v \leftarrow getValue()$;
26:     $id \leftarrow entryID()$;
27:     For each $Q_j \in Q$ call $PredicateTest(id, v, Q_j^N, i_j, Q, Q_j)$
28:   **case** $CHARACTER$**:**
29:     $v \leftarrow getValue()$;
30:     For each $Q_j \in Q$ call $PredicateTest(id, v, Q_j^N, i_j, Q, Q_j)$
31:   **end switch**
32: **end while**
33: $R \leftarrow R \setminus Q$;
34: **return** R;

Algorithm 4.5 $PredicateTest(id, v, Q^N, i, Q, q)$

Input: Entry ID id, value in the steam v, normalized query Q^N, current index variable i of Q^N, current query set Q, and the current checked query q

1: $b \leftarrow false$;
2: **if** $Q^N[i+1] == -3$ and $Q^N[i+2] == id$ **then**
3: **if** $Q^N[i+3] == -10$ and $Q^N[i+4] == v$ **then**
4: $b \leftarrow true$;
5: **else if** $Q^N[i+3] == -11$ and $Q^N[i+4] < v$ **then**
6: $b \leftarrow true$;
7: **else if** $Q^N[i+3] == -12$ and $Q^N[i+4] > v$ **then**
8: $b \leftarrow true$;
9: **end if**
10: **if** $b == true$ **then**
11: $i \leftarrow i + 4$;
12: **end if**
13: **if** $i == |Q_j^N|$ **then**
14: $Q \leftarrow Q \setminus q$;
15: **end if**
16: **end if**

4.3.4 Example

We will conclude this subsection by presenting an example that applies the *BasicEXIFilter* algorithm presented above to three XPath queries. The underlying data model used is defined in Figure 3.4. The binary XML message instance tested here, which corresponds to the plain-text XML variant shown in Listing 2.9, follows the EXI grammar shown in Figure 3.5. The sample queries are defined as follows:

- $Q_1 = /Envelope/*/status$
- $Q_2 = //Temperature[@scale = 'Celsius']/value[text() < 22.4]$
- $Q_3 = //status[text() = 'Error']$

The first step is to normalize the XPath queries in compliance with Definition 4.3. To do so, we are applying the *XPathNormalizer* algorithm algorithm, which returns the following output:

- Q_1^N = 10 -1 6
- Q_2^N = -2 4 -3 5 -10 'Celsius' 8 -3 8 -11 '22.4'
- Q_3^N = -2 6 -3 6 -10 'Error'

For each Q^N, the prefix function is determined. However, it quickly becomes apparent that there are no longest prefixes that are also suffixes of the query expressions. Hence, each entry of π_1 and π_2 will be assigned the value -1.

The following table provides the processing steps for each query in the order of the EXI events that occur during the decoding process. Column one provides the EXI streams, the next row presents the EXI events that are encountered, and the last three columns provide the current location step i (match progress) of the normalized queries Q_1^N, Q_2^N, and Q_3^N. Once a match has been discovered ($i == |Q^N| - 1$), we stop evaluating the corresponding query. A mismatch will also be registered whenever the requested elements / attributes are not present in the EXI stream ($i \neq |Q^N| - 1$).

As can be seen, i is incremented of a query if a *startElement* provides the requested entry ID number at the corresponding position in the normalized query. Please note that the wildcard signalization (-1) functions as a universal ID and matches each ID provided at its particular position. In the case of the descendant-or-self expression (-2), an incrementation is applied at first occurrence. Furthermore, this will affect the next processing steps, which

EXI Stream	StAX Event	Step i_1	Step i_2	Step i_3
	START_DOCUMENT	-1	-1	-1
000	START_ELEMENT ⇒ 10	0	0	0
1	START_ELEMENT ⇒ 11	1	0	0
	START_ELEMENT ⇒ 6	2	0	1
'LowBattery'	CHARACTER ⇒ 'LowBattery'	-	0	1
	END_ELEMENT	-	0	1
	END_ELEMENT	-	0	1
	START_ELEMENT ⇒ 9	-	0	1
11	START_ELEMENT ⇒ 4	-	1	1
0 'Celsius'	ATTRIBUTE ⇒ 5 / 'Celsius'	-	5	1
	START_ELEMENT ⇒ 8	-	6	1
'21.1'	CHARACTER ⇒ '21.1'	-	10	1
	END_ELEMENT	-	-	1
	END_ELEMENT	-	-	1
	END_ELEMENT	-	-	1
	END_ELEMENT	-	-	1
	END_DOCUMENT	-	-	1

are similar to the Knuth-Morris-Pratt algorithm previously introduced and discussed. If there is a local mismatch, we resort to the prefix function π to determine which entry of the normalized query is to be checked again. Thus, the successful local match of Q_2 starts when the temperature element (ID entry=6) occurs.

Predicates are evaluated when the StAX *character* and *attributes* event occurs. If a predicate has been evaluated successfully, i will be increased by 4. This is justified by the fact that at this particular point the algorithm checks whether there is a predicate defined in the query (=-3), whether the requested attribute or element ID is present in the stream, and whether the operation check (-10, -11, and -12) is true as compared to the value requested in the predicate.

Based on the given EXI stream, queries Q_1 and Q_2 are successfully evaluated. Q_3 failed because the predicate evaluation missed the comparison condition, since it is 'Error'≠'LowBattery'. The case of Q_3 shows that a mismatch can only be identified once the end of the stream has been reached and the endDocument event has been triggered. An early mismatch registration is not possible, even though we left the *status* element level in the parsing process and a status element cannot occur in the stream again. This is due to the fact that the comparison is done on the top of the EXI grammar and there is a nescience as to which kinds of *startElement* events may be encoun-

tered next. Thus, query Q_3 will be kept for the duration of the evaluation since there may yet be events which match the XPath expression.

This negative aspect is underlined by this worst case scenario: let us assume that we are going to evaluate an XML-based message that carries humidity information as shown in Listing 2.10.

EXI Stream	StAX Event	Step i_1	Step i_2	Step i_3
	START_DOCUMENT	-1	-1	-1
000	START_ELEMENT⇒ 10	0	0	0
0	START_ELEMENT⇒ 9	0	0	0
	START_ELEMENT⇒ 3	0	0	0
'Relativ'	ATTRIBUTE⇒ 5 / 'Relativ'	0	0	0
	START_ELEMENT⇒ 8	0	0	0
'64'	CHARACTER⇒ '64'	0	0	0
	END_ELEMENT	0	0	0
	END_ELEMENT	0	0	0
	END_ELEMENT	0	0	0
	END_ELEMENT	0	0	0
	END_DOCUMENT	0	0	0

In this case, the step values of i_1, i_2, and i_3 would only be incremented once, namely to zero after the *startDocument* event. All other events encountered would never reach the step number to identify a match, unless the *endDocument* event occurs, which would signal that there is a total mismatch. The question arises whether we can avoid these types of scenarios and whether we can immediately evaluate a mismatch whenever a partial test is registered negatively and there is no (logical) chance of finding a further match. The *OptimizedEXIFiltering* mechanism will take this into account and will be presented in Section 4.4.

4.3.5 Conclusion

The processing steps of the *BasicEXIFiltering* approach presented here to evaluate a set of given queries can be summarized as follows:

1. Read the predetermined number (provided by the underlying EXI grammar) of bits of the input EXI stream message.

2. The StAX events encountered (e.g., *startElement, attribute,* etc.) are used to evaluate predicates or the local entry ID of the queries for a local match.

This mechanism is repeated until the last bit of the EXI stream is read (endDocument event is triggered) or until all possible queries have been checked for a match. Due to the separation of the determining the StAX event based on the EXI grammar and the subsequent query evaluations leads to the calling that *BasicEXIFiltering* works *on the top* of the EXI grammar.

This separation approach, however, has two disadvantages: this mechanism keeps queries on track even if it becomes logically impossible to find a successful match based on the input EXI stream message currently provided. Such a case was described in subsection 4.3.4. A mismatch can only be identified at the end of the parsing process. This scenario also shows that areas in the EXI grammars might be traversed that would never affect any further queries. Consequently, an improvement in terms of prompt evaluation is desirable for cases that involve particular grammar fragments that definitely won't match any one of the given queries.

The another disadvantage involves checking all active queries (no match was found so far) for some StAX events, such as attributes and characters, even if no or only a small subset of predicates are defined as refinement in some queries. This leads to an unnecessary processing overhead for a number of cases, which takes no any advantages of a partial match or mismatch decisions.

4.4 Optimized Binary XML Filtering

The *BasicEXIFiltering* approach is a filtering mechanism for binary XML that works *on top* of an EXI grammar. In this section, we present the *OptimizedEXIFiltering* approach that maps all given XPath queries within the EXI grammar and enables evaluation *on the fly*. More precisely, we evaluate the queries during the same step at which EXI decoding happens. This avoids the separation of the EXI grammar and the actual evaluation, as found in *BasicEXIFiltering*; instead, all evaluation processes are harmonized within the EXI grammar. This sophisticated grammar will also be called *EXI filter grammar*. Below, the processing steps are explained for how such a filter grammar is determined. We will start by explaining how the so-called accepting states and predicate states are determined. These distinct states are then used to construct the filter grammar.

4.4.1 Determining Accepting and Predicate States

To map all given queries in an EXI grammar, we need an analytical step to mark particular states that indirectly represent the given XPath query

expressions. In this context, we have to identify the so called *Accepting States (AS)* and *Predicate States (PS)* in the EXI Grammar. First, lets us define AS and PS:

Definition 4.4 (Accepting States (AS) and Predicate States (PS)) Let G be an EXI grammar and Q an XPath expression. An accepting states (AS) of G is a distinct state that represents the element or attribute addressed in Q. If Q contains at least one predicate refinement, a predicate state (PS) of G would also be a distinct state for which a predicate evaluation has to be performed as indicated by the corresponding location step in Q.

It should be noted that a state in G may be designated as AS and PS at the same time. In particular, this is the case whenever the node requested is refined with a predicate on the underlying type value. It should further be noted that an XPath expression may have more than one AS and PS. This effect occurs when multi identical elements and/or attributes are used; however, these are located in different sub-structures.

To understand how AS and PS of a given EXI grammar can be identified, we will consider the same queries given in Section 4.3.4 and the EXI grammar G provided in Figure 3.5. Let us start with the first query, which has the expression

$$Q_1 = /Envelope/*/status \quad .$$

Q_1 has no predicate defined and hence, we only have to identify the AS state that requests the *status* element, for which the *Header* (one element which would be logical for the wildcard in the schema context) and *Envelope* as ancestor nodes. Finding the AS for Q_1 is quite straightforward because the expression itself provides an almost direct navigation through G. Figure 4.2 explains this aspect by following the blue state and path of transitions: considering the first expression entry, we have to follow the transition that leads to the *Envelope* state from the start state within the *Root* sub-grammar. We enter the next sub-grammar, namely the *Envelope*, and read the next child node of Q_1, which is the * wildcard expression. From that start state we have to follow the transitions that leads to the *Header* and to the *Body* state. This is justified by the fact, at that point the expression accept any node and thus we have to follow each transition possible from the start state in the *Envelope* grammar. First, let us follow the *Body* state and its grammar representation respectively. Since the next and final node in Q_1 is the *status* element addressed, we remain stuck within the *Body* grammar since there is no transition that leads to the corresponding *status* state. We will stop here

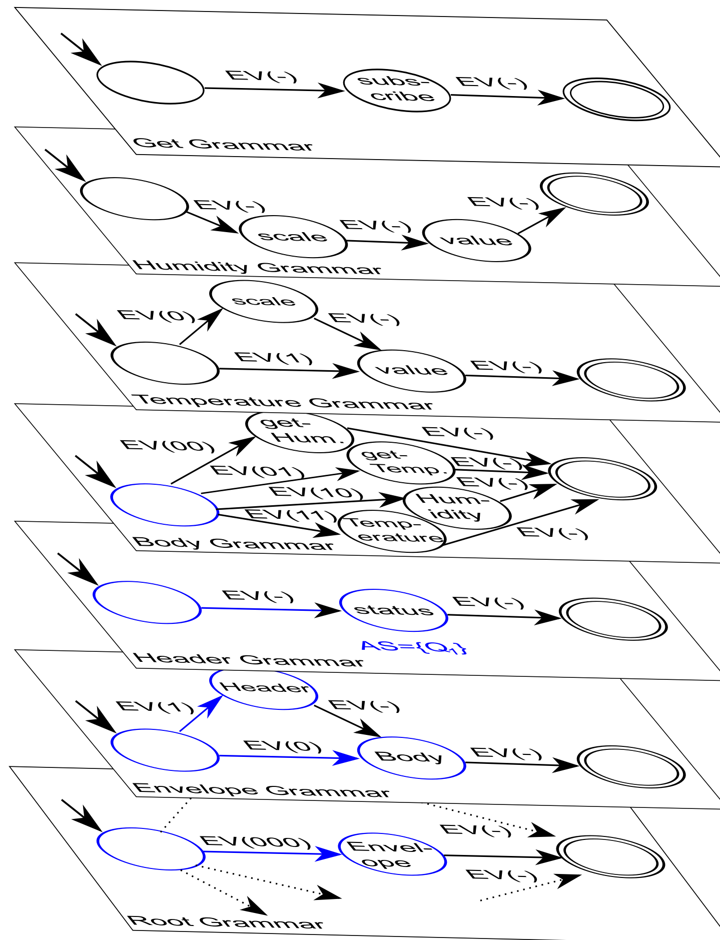


Figure 4.2: Navigation and search path for finding the AS for the query expression $Q_1 = /Envelope/*/status$ based on the EXI grammar G provided in Figure 3.5.

and go to the next grammar, namely the *Header*, which can also be accessed from the start state in the *Envelope* grammar. There we will find the *status* state requested and we will mark this distinct state as AS.

Now let us consider a more complex XPath expression, which also contains predicate refinements (see the example in Section 4.3.4):

- $Q_2 = //Temperature[@scale = 'Celsius']/value[text() < 22.4]$
- $Q_3 = //status[text() = 'Error']$

Both queries start from the root node with the descendant-or-self axis expressions. More precisely, Q_2 selects all *Temperature* nodes that are in the

node set of the root descendants (and the root self). Similarly, Q_3 selects all *status* elements that are in the node set of the root descendants (and the root self). In other words, we have to to find both all *Temperature* states and *status* states in grammar G. Since there is no pre-knowledge of which exact paths should be taken to reach all desired states, a brute force search by Depth-first search (DFS) or Breadth-first Search (BFS) has to be applied. Starting from the start state in the root grammar, all transitions are traversed to check if their subsequent states represent the desired node. If this is not the case, all transitions of that state are recursively traversed and so on. A search path terminates if either the desired node has been found or if the end state of the root grammar has been visited. Since we are using finite EXI grammar G, such a search will be finite and the worst case search complexity would be $\mathcal{O}(|S| + |T|)$ where S represent all states and T all transitions of G.

Going back to our queries Q_2 and Q_3 it can be quickly becomes apparent that the desired nodes selected by the descendant-or-self expression are represented by the state *Temperature* in the Body grammar and the *status* in the Header grammar. At this point, we only continue from these states to find the AS and PS. Let us focus on Q_2 first. The *Temperature* node selection contains a predicate with the condition that the attribute *scale* shall be *Celsius*. Thus, we will mark the *scale* state within the *Temperature* grammar (called by the *Temperature* state) as PS with the corresponding condition (see Figure 4.3). The next entry of the XPath expression is the *value* node requested. It also has a refinement based on a predicate, namely that the value content shall be smaller than 22.4. As we noted for the PS of the *scale* attribute, we will also mark this state as PS with its condition. Since the *value* state is also the state requested in Q_2, this state will additionally be designated as AS. Q_3 is straightforward then. The *status* state in the header grammar is marked as PS with the *Error* condition. Based on Q_1, this state is already an AS, and since Q_3 requests the same node, the AS set will be extended by Q_3 to identify the query in case of a match.

Figure 4.3 shows all found AS and PS that were found based on queries Q_1, Q_2, and Q_3 in the grammar G:

AS:
- *status* state in G_{Header} because of Q_1 and Q_3
- *value* state in $G_{Temperature}$ because of Q_2

PS:
- *status* state in G_{Header} because of Q_3
- *scalue* state in $G_{Temperature}$ because of Q_2
- *value* state in $G_{Temperature}$ because of Q_2

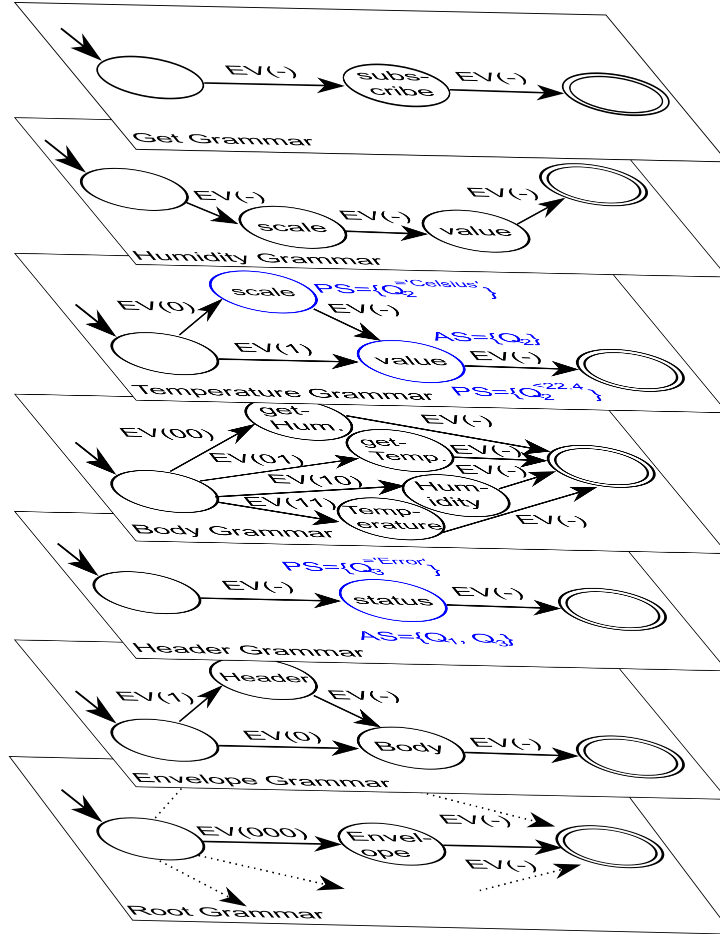


Figure 4.3: All accepting and predicate states in G based on Q_1, Q_2, and Q_3.

Before we conclude this processing, we will present the generic algorithm *DetermineASPS* which determines all AS and PS of a given EXI grammar G and XPath queries (see Algorithm 4.6). For a simple processing of the XPath expression, we are going to transform each query into its normalized representation, based on Definition 4.3, by applying the *XPath-Normalizer* algorithm (see Algorithm 4.3) in line 1. Beginning with the start state s^{root} of the root grammar (line 2), we take each normalized query applied to grammar G to search for AS and PS (lines 3-6). We implemented the strategy of DFS, which is realized by calling *CheckASPS* (Algorithm 4.7). This sub-routine checks whether the current state s passed is PS (lines 4-6) and/or an AS (lines 8-10). An AS is only possible, if the last index in the normalized query has been reached or if the last node in the expression has a predicate refinement. A PS is found when the corresponding predicate signal-

Algorithm 4.6 *DetermineASPS*(G, Q)

Input: EXI grammar G and a set of queries $Q \leftarrow \{Q_1, Q_2, ...Q_n\}$
Output: EXI grammar G^{ASPS} with distinguished accepting and predicate states

1: Apply each query in Q to $XPathNormalizer()$ to get normalized XPath representation $Q^N \leftarrow \{Q_1^N, Q_2^N, ...Q_n^N\}$
2: Let s^{root} the start state of the root grammar of G
3: **for** each $q \in Q^N$ **do**
4: $\quad i \leftarrow 0$;
5: $\quad CheckASPS(s^{root}, q, i)$;
6: **end for**
7: **return** G^{ASPS};

ization is present in the query expression ($= -2$) and the ID of the current state matches. Otherwise, we take each transition and check its successor state to determine which should be visited next (lines 12-21). This selection is performed if the next location step entry match (either an ID match or a wildcard match). For a descendant-or-self axis in a query expression, we will follow each transition and visit each state until the node addressed has been found. For a state which reflects a complex type we continue the search for AS and PS in the sub-grammar (lines 14-18). When we have found at least one AS in the sub-grammar, we will mark the state $t.s$ by the function *markQueryRelevance()* to show which query Q^N has called the sub-grammar. This is an important process when it comes to multiple queries that might address the same sub-grammar, but differ in either precursor nodes or states.

Algorithm 4.7 *CheckASPS*(s, Q^N, i)

Input: Current state s, normalized query Q^N, and location step i
Output: True, if pattern P found in T, otherwise false.

1: Let $T \leftarrow \{t_1, t_2, ...t_n\}$ the transitions of s to all possible successor states $t_1.s, t_2.s, ...t_n.s$
2: $j \leftarrow i$;
3: $b \leftarrow false$;
4: **if** $-2 == Q^N[j]$ and $ID^s == Q^N[j+1]$ **then**
5: $\quad PS \leftarrow PS \cup s$;
6: $\quad j \leftarrow j + 4$;
7: **end if**
8: **if** $(|Q^N| == i)$ or $(|Q^N| - 4 == i$ and $-2 == Q^N[i+1])$ **then**
9: $\quad AS \leftarrow AS \cup s$;
10: $\quad b \leftarrow true$;
11: **end if**
12: **for** each $t \in T$ **do**
13: $\quad$ **if** $(-3 == Q^N[i]$ and $t.ID == Q^N[i+1])$ or $(t.ID == Q^N[i])$ or $(-1 == Q^N[i])$ **then**
14: $\quad\quad$ **if** $t.s$ is complexType-based state **then**
15: $\quad\quad\quad$ Let $s^{t.s}$ the start state of the sub-grammar in G that represents $t.s$
16: $\quad\quad\quad$ $b \leftarrow CheckASPS(s^{t.s}, Q^N, j+1)$;
17: $\quad\quad\quad$ **if** b==true **then**
18: $\quad\quad\quad\quad$ $markQueryRelevance(t.s, Q^N)$;
19: $\quad\quad\quad$ **end if**
20: $\quad\quad$ **end if**
21: $\quad\quad$ $b \leftarrow CheckASPS(t.s, Q^N, j+1)$;
22: $\quad$ **end if**
23: **end for**
24: **return** b;

4.4.2 Determining Filter Grammar G_F

After determining the states involved in the EXI Grammar for each query, the actual filter grammar G_F is built. Before we explain this next step, let us first define the filter grammar G_F:

Definition 4.5 (Filter Grammar G_F)
Let $G = (S, T)$ be an EXI grammar and Q a set of XPath query expressions. The filter grammar $G_F = (S_F, T_F)$ is a subset of G ($S_F \subseteq S$ and $T_F \subseteq T$) that only contains those states and transitions deemed necessary to evaluate each query in Q.

In other words, a filter grammar G_F has removed all states and transitions that would only be used for decoding messages, and thus would never address any given query. To create a filter grammar G_F which can be used for evaluation, we propose that the following steps have to be performed:

I Find all routes from each AS in G to the start state of the root grammar.

II Remove all states and transitions that would skip a *PS* , which, however, is requested by a query, within the routes discovered.

III Delete all transitions and states which are not part of one of the routes found. All *AS* will be end states as long as no successor *AS* has been found at this point.

Below, we present and discuss each step in detail.

Step I

Based on the previous section, in step one we are taking the discovered AS as our starting point to find all routes to the start state of the root grammar. The reverse approach simplifies the route search since all possible directions sooner or later lead to the root (state). Figure 4.4 shows the result of all routes from the two AS (*status* and *value*) to the start state of the root grammar based on our sample EXI grammar. In general, we have to take into account two special cases during this processing step:

- If the start state of a grammar level has been reached, identify all relevant states which are able to call this start state and continue the search from there.

- If a new state that belongs to a possible route has been added and if this state is represented by its own sub-grammar (complex type-based state), all states and transition of this sub-grammar are also members of the route.

The first point clarifies cases in which multiple states in a grammar can call the same sub-grammar. However, at this point, we are only interested in states which are logically relevant to a given XPath expression. There are different strategies for identifying relevant states. One variant applies the XPath expression backwards to the current route and the states and transitions involved are marked for the corresponding query relevance. However, this causes processing overhead and can be avoided if we collect all relevant states when we determine all AS and PS. Algorithm 4.6 determines these relevant states and marks them already.

The second point highlights the importance of involving the sub-grammars of all states that are complexType-based and are members of a route. This special case can also be found in Figure 4.4 , which takes the complete *Header* grammar into account when the routes from value AS are discovered. The motivation for doing so is based on a valid decoding of sub-elements of messages that are still in the context of a given XPath expression even if an evaluation during the current decoding process is not yet possible. Consider our query example Q_2. This query addresses the *value* element nested within the *Temperature* element with predicates. Thus, to evaluate this query it does not matter whether a message embeds at the beginning the *Header* element or not. The crucial part starts at the beginning of the *Temperature* state and until then any information can occur beforehand. To take these varying pre-opportunities into account, we have to keep all sub-grammars in the routes that are necessary for decoding message fragments until the actual evaluation starts.

Step II

This processing step enables early evaluation of queries that are refined by predicates. In general, a message can only be successfully evaluated if the requested nodes in the XPath expression occur in that order and if the evaluation of the predicates is true. The routes discovered in step I will guarantee the order. However, these routes may also contain variants that would skip required predicate evaluations. Such a scenario is given in Figure 4.4 at the *Temperature* grammar level. Starting from the AS *value*, the route can either be taken to the PS *scale* and then to the start state of this grammar level or directly to the start state. The latter variant, however, would skip the PS that is a fundamental demand of Q_2. At this point, it would be desirable to

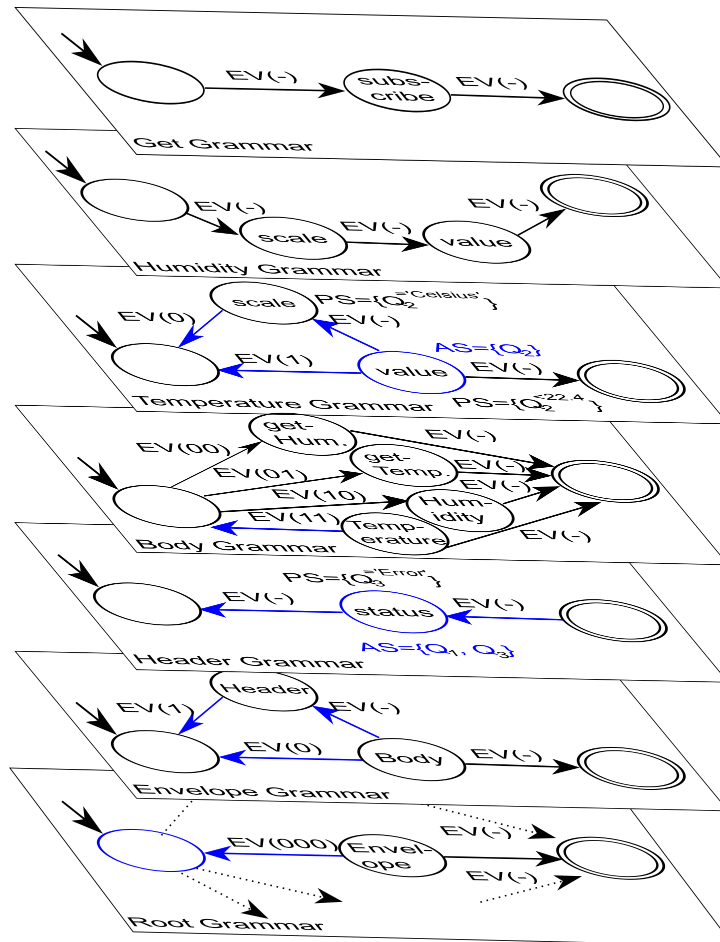


Figure 4.4: Route to the start state of the root grammar from the two marked AS (*status* and *value*)

avoid such shortcuts to evaluate early on if a message does not embed the *scale* attribute value. Consequently, this shortcut transition is removed from the route. Hence, only messages containing the *scale* attribute within the *Temperature* element can be decoded.

This processing step is only valid as long as another query is not involved in this route which would accept such a shortcut. E.g., a new XPath expression $/Envelope/Body/Temperature/value[text() < 24.7]$ accepts messages that may or may not use the *scale* attribute. Based on this query, we have to retain the transition between the start state and the AS *value*, even though Q_2 only accepts messages with the *scale* attribute. In order to still make valid evaluations at runtime, we are introducing the *Predicate Table*, which registers all successful and unsuccessful predicate evaluations. This table will

be explained in section 4.4.4.

Step III

This final step now creates the desired filter grammar G_F, which can be used to evaluate any input messages for a match of one or more queries. All transitions and states that are not part of the routes discovered based on steps I and II will be deleted. Figure 4.5 shows the result of the elimination process that leads to the filter grammar G_F. It also highlights the effect of the absence of queries that do not request likely information. For example, there was no query addressing humidity information and hence all states and transitions that represent this information were removed. This also holds true for get-based messages (*getTemperature* and *getHumidity*). It can also be seen that all end states that are not part of one of the routes were eliminated. We will reassign the end states to all AS in G_F as long as the states have no logical possible successor AS. The AS *value* is such a case. In contrast, AS *status* cannot be marked as an end state since AS *value* can be still decoded. During the decoding process of a message, an AS which is also an end state in a G_F signals that other queries can no longer evaluated at this point.

Corresponding to the number of states and transitions are used for G_F it can be said that the following relation will always be valid:

$$|G_F| \leq |G|$$

Thereby, the absolute value (|..|) provides the total number of states and transition used by the corresponding grammar. This equation is valid because G_F is a subset of G (see Definition 4.5) and may maximally contain all states and transitions when there is a demand of one or more XPath queries which covers the whole information spectrum of all possible messages. A filter grammar can also be empty: $|G_F| = 0$. E.g., this is the case if not a single query is present or if all queries address nodes that are not valid in terms of the underlying XML schema that represents all message structure variants.

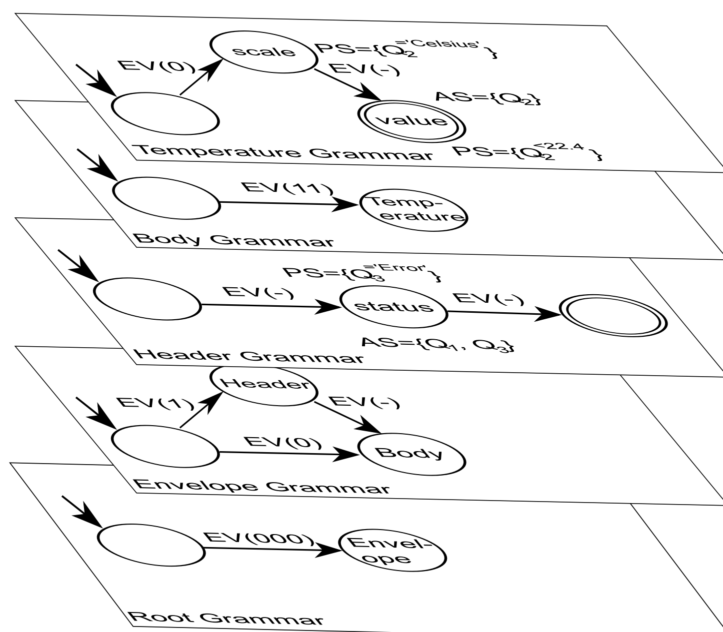


Figure 4.5: Filter Grammar G_F

4.4.3 The *OptimizedEXIFilter* Algorithm

The optimized EXI Filter algorithm (see Algorithm 4.8) brings together all aspects presented above and provides an EXI filter grammar G_F that can be used to evaluate a service message of relevance. The input takes a given EXI grammar and a set of queries using XPath expressions. The first major step involves identifying the AS and PS in G (line 2). To do this, the *DetermineASPS* algorithm is called (see Section 4.4.1 and Algorithm 4.6). Next, relevant states and transitions necessary to reach each AS and PS are determined (lines 3-14). This corresponds to the described steps I and II as presented in Section 4.4.2. We start by identifying all states and transitions of each AS (line 3). The *routeToRootStartState* function determines all possible transitions (T_R) and states (S_R) that can be visited to reach AS (line 4). After this process, we will identify all states and transitions that are able to skip a mandatory predicate evaluation (lines 7-8). The function *skippablePart* searches all possible states (S_R) and transitions (T_R) that could from an alternative route without visiting PS. This function also takes into account contradictions with other queries and selects only those states and transitions not required by other queries or for reaching an AS. All valid skippable states in S_R and transitions in T_R are removed from S_R and T_R respectively (line 13). As a final step, we are removing all leftover states and transitions in G^{ASPS} that do not at least reach one AS (lines 14-15). This

Algorithm 4.8 *FilterGrammar*(G, Q)

Input: EXI grammar $G = (S, T)$ and a set of queries $Q \leftarrow \{Q_1, Q_2, ...Q_n\}$
Output: EXI filter grammar $G_F = (S_F, T_F)$

1: $S_F \leftarrow \emptyset$; $T_F \leftarrow \emptyset$; $S_R \leftarrow \emptyset$; $T_R \leftarrow \emptyset$; $S_D \leftarrow \emptyset$; $T_D \leftarrow \emptyset$;
2: $G^{ASPS} \leftarrow DetermineASPS(G, Q)$;
3: **for** each AS in G^{ASPS} **do**
4: $\quad \{S_R, T_R\} \leftarrow routeToRootStartState(G^{ASPS}, AS)$;
5: $\quad \{S_R, T_R\} \leftarrow \{S_R, T_R\} \cup \{S_R, T_R\}$;
6: **end for**
7: **for** each PS in G^{ASPS} **do**
8: $\quad$ **if** isSkippable$(PS, \{S_R, T_R\})$ **then**
9: $\quad\quad \{S_D, T_D\} \leftarrow skippablePart(PS, \{S_R, T_R\})$;
10: $\quad\quad \{S_D, T_D\} \leftarrow \{S_D, T_D\} \cup \{S_D, T_D\}$;
11: $\quad$ **end if**
12: **end for**
13: $\{S_R, T_R\} \leftarrow \{S_R, T_R\} \backslash \{S_D, T_D\}$;
14: $S_F \leftarrow S \cap S_R$;
15: $T_R \leftarrow T \cap T_R$;
16: **return** G_F;

leads to our filter grammar G_F which is returned (line 16).

4.4.4 Example

In this section we will apply some XML-based messages to the constructed filter grammar G_F which can be seen in Figure 4.5 to demonstrate the effectiveness of the evaluation process. Let us start with the temperature-based message which can be seen in Listing 2.9. The following table provides the processing steps in the context of the input EXI stream (column 1) and states that will be affected in G_F (column 2). If a state is an AS and/or PS, column 3 provides the queries which are responsible for the distinct state. Columns 4 and 5 present the *Predicate Table* mentioned above (see processing step II in Section 4.5). This table evaluates the logical terms used within the predicates of the query. Terms t_1 and t_2 belong to Q_2 with the condition that

$$t_1 := scale == \text{'}Celsius\text{'}$$

$$t_2 := value <' 22.4'$$

and term t_3 belongs to Q_3 with the condition that

$$t_3 := status == \text{'}Error\text{'} \quad .$$

If multiple predicates are used in a query (at different node axis), the predicate table will evaluate the predicate conditions by logical conjunction (see column 4). Consequently, a query with a predicate may only be valid if all terms are true.

EXI Stream	State	AS / PS	$t_1 \wedge t_2$	t_3
	Start G_{root}	- / -	(false ∧ false)	false
000	*Envelope*	- / -	(false ∧ false)	false
	Start $G_{Envelope}$	- / -	(false ∧ false)	false
1	*Header*	- / -	(false ∧ false)	false
	Start G_{Header}	- / -	(false ∧ false)	false
'OK'	*status*	$\{Q_1, Q_3\}$ / $\{Q_3\}$	(false ∧ false)	false
	End G_{Header}	- / -	(false ∧ false)	false
	Body	- / -	(false ∧ false)	false
	Start G_{Body}	- / -	(false ∧ false)	false
11	*Temperature*	- / -	(false ∧ false)	false
	Start $G_{Temperature}$	- / -	(false ∧ false)	false
1 'Celsius'	*scale*	- / $\{Q_2\}$	(true ∧ false)	false
'21.1'	*value*	$\{Q_2\}$ / $\{Q_2\}$	(true ∧ true)	false
Result= $\{Q_1, Q_2\}$				

To evaluate the input message, we begin at the start state of the root grammar as per usual. All terms will be initialized by the value $false$. Afterwards, we start to read the EXI stream to decide which transition to traverse next. Since only a single transition is left at this state in G_F, only the signalization for the *Envelope* element is valid. The stream can fulfill this first test and hence, the *Envelope* with its grammar representation will be entered. There, we will decide whether to visit the *Header* or the *Body* state. Based on the EXI stream (= 1), we follow the transition to the *Header* and enter its grammar level. There, the *status* state is encountered, which is marked as AS (by query Q_1 and Q_3) and PS (by query Q_3). The first successful match can be evaluated at this point since Q_1 requested that element. However, a mismatch can also be evaluated due to the predicate condition of Q_3. This will fail because term t_3 expects an 'Error' status value, which, however, is 'OK'. The parsing and evaluation process is not yet finished and we continue by leaving the *Header* grammar. Next, we enter the *Body* state and its grammar representation respectively. There, we have to test whether

the EXI stream signalizes (= 11) the *Temperature* element since only this transition remains. This test is successful and we continue on to visit the *Temperature* grammar. Here, we are going to evaluate the presence of the optional *scale* attribute. This test is also successful and we are now going to visit the PS, where we have to evaluate the given predicate term t_1 based on Q_2. As can be seen, the outcome is true since the EXI stream contains this particular scale value. Subsequently, we are going to visit the final state, namely the *value* state, which is marked as AS as well as PS based on Q_2. To evaluate, whether this EXI message matches this query we have to check the term t_2 for correctness. The temperature value can fulfill the condition and the logical conjunction is *true*. The evaluation process terminates here since an AS that is also an end state has been reached. As a result set will we would get $\{Q_1, Q_2\}$ as a successful match based on the input stream provided.

Now let us apply a different message to G_F that only embeds the *Humidity* information (see Listing 2.10).

EXI Stream	State	AS / PS	$t_1 \wedge t_2$	t_3
	Start G_{root}	- / -	(false ∧ false)	false
000	*Envelope*	- / -	(false ∧ false)	false
	Start $G_{Envelope}$	- / -	(false ∧ false)	false
0	*Body*	- / -	(false ∧ false)	false
	Start G_{Body}	- / -	(false ∧ false)	false
10	-	- / -	(false ∧ false)	false
Result= {}				

As can be seen, we are starting in the same manner as before since the first two bits are identical. The paths begin to diverge in the *Envelop* grammar; there we skip the *Header* state and directly enter the *Body* state with its grammar representation. At this point, it becomes clear that Q_1 and Q_3 can no longer be fulfilled. Being at the start state in the *Body* grammar, we are going to read the next EXI stream fragment which signalize 10. However, there is no relevant transition that can be traversed and hence, we are terminating here since we do not have any decoding rules anymore. Furthermore, it is also signalized that not a single query is requesting humidity-based information. The evaluation of this message would return an empty result set and it is remarkable that this can be determined after only reading 6 bits of the EXI stream. Other use cases may requires much less bits to decide for non relevance such as with two bits reading as we presented in [Käbisch et al., 2012].

In contrast to the *BasicEXIFiltering* we do not have to parse the entire input message to decide that no query match was found. In addition, we avoid the processing overhead of checking all queries for a local match whenever a *startElement* or *attribute* event occurs. The *OptimizedEXIFiltering* avoids representing queries in an additional data structure as the *BasicEXI-Filtering* does. All queries are represented within the (reduced) EXI grammar G_F including AS and/or PS.

4.4.5 Filter Code Generation

Based on the filter Grammar G_F that contains the predicate evaluation functionality and the accepting state, we are able to use a generic EXI interpreter such as EXIficient[2] for evaluating an EXI message. However, such a solution is not a suitable one for constrained embedded devices due to their highly restricted memory and processing capacities.

Therefore, we are able to extend our code generation tool as presented in section 3.3 for filtering purpose:

I The analysis of XML schema information provides all possible XML elements, attributes, and constraints in a specific schema context.

II (a) Based on domain-specific functionalities and datatypes the EXI grammar set (G) for decoding purpose is generated.

   (b) For each query, determine AS and PS of G (see Section 4.4.1)

   (c) Build G_F by removing all states and transitions which do not lead to a AS or PS (see Section 4.4.2)

III Based on G_F the source code for the *EXI Processor* is generated, which involves only the decoding mechanism and the evaluation implementations requested.

Mainly, step II and III are modified. Step II integrates the mechanism as described in previous subsections. Step III only generates the code for decoding EXI stream messages and the evaluation methods. Not generating the code for encoding the EXI stream is explained by the simple fact that we only want to filter EXI streams to identify whether the information requested is present or not.

[2] http://exificient.sourceforge.net/

4.5 Experimental Evaluation

In this section we are going to evaluate the applicability and effectiveness of the binary XML filtering approaches presented above. Our evaluation considers two aspects. First, the performance of the approaches is tested in general by applying different sets of queries. To estimate how these approaches perform in comparison to an existing XML-based filtering mechanism, YFilter [Diao and Franklin, 2003] is involved in this test. This selecting is justified by the fact that YFilter is both a very prominent and an automaton-based approach in the domain of efficient plain-text XML document filtering [Sadoghi et al., 2011]. The idea and the functionality of YFilter will be discussed in the related work section (Section 4.6).

The second aspect involves analyzing results in terms of code footprint and RAM usage when the binary XML filtering approaches are applied to the embedded environment. To do so, we set up a demo embedded network based on microcontrollers that simulates a charging scenario based on the ISO/IEC 15118 message protocol.

4.5.1 Implementation

Both binary XML filtering approaches, *BasicEXIFiltering* and *OptimizedEXIFiltering*, are implemented using Java programming language and utilize the open source W3C EXI de-facto reference implementation[3]. For the described code-generation mechanism described in section 4.4.5 we modified our existing implementation to realize the filtering functionality described above in a code-generated way. Thus far, the generator produces source code in the C and Java programming languages, which can be used by platforms such as Contiki[4] and Java Micro Edition CLDC 1.1, respectively.

4.5.2 Performance

To evaluate the performance of both approaches, we also looked at the performance results of YFilter[5] as implemented for the same data set. The XML-based documents used in the evaluation process are based on the different ISO/IEC 15118 charging message patterns presented in Section 3.4.1. Hence, the sample message instances range in size from 700 to 3000 bytes in plain-text format. The binary XML representations range in size from 10

[3] http://exificient.sourceforge.net/

[4] Contiki is an operating system for memory-efficient networked embedded systems and wireless sensor networks (http://www.sics.se/contiki/)

[5] http://yfilter.cs.umass.edu/

bytes up to 30 bytes. Since we are unable to process plain XML on small embedded devices such as microcontrollers, and especially not the YFilter algorithm, these performance experiments were conducted on an Intel Core 2 Duo with 2.10GHz and 3GB RAM.

For all filter variants, resulting measurements do not include time spent setting up the filter itself. The filter system is determined only once, when a new XPath set is presented. For purposes of fair comparison, the parsing time of the XML-based documents is always included in the resulting measurements for *BasicEXIFiltering*, *OptimizedEXIFiltering*, and YFilter. This is based on the fact that our presented approaches do not pre-process the binary XML message stream; in addition, they evaluate XPath queries at the same time as parsing the binary XML stream (*filter-on-the-fly*). Only for purposes of comparison did we also integrate the performance results of YFilter that do not take into account the parsing and indexing of the input XML message. In other words, the numbers only show the performance of the YFilter Non-deterministic Finite Automaton (NFA).

In general, the average time for 1000 rounds is determined for each query set. In each round, a random request or response message is selected and applied to the filter mechanism. The XPath sets are based on queries with different kinds of requests. In the context of this thesis, the number of XPath query sets executed is relatively small and oriented more toward usage in the embedded domain.

Figure 4.6 shows the results in milliseconds of our performance experiments for 8 different XPath query sets. Considering the first query sets, both binary XML filtering approaches always performs much better than the YFilter implementation. For 5 queries the *BasicEXIFiltering* approach is 30 times faster and the *OptimizedEXIFiltering* approach is 40 times faster. This highlights the benefit of operating directly within the binary XML document without having to transform it into a plain-text representation. Furthermore, we are able to evaluate the XPath queries right during the decoding process. YFilter separates this process (XML document parsing and XPath evaluation by the NFA constructed) which leads to slower performances. Even if we only consider the NFA processing of YFilter, our *OptimizedEXIFiltering* is still 3-4 times faster.

BasicEXIFiltering's performance decreases as the number of XPath queries increases. This is due to the fact that all queries are checked as to whether the current step index can be incremented or not for each start element or attribute StAX event that occurs during the decoding process. This processing overhead becomes dominant and leads to lower performance as the query set is getting larger. At about > 320 queries the *BasicEXIFiltering* starts to perform more slowly than YFilter. Compared to YFilter NFA only,

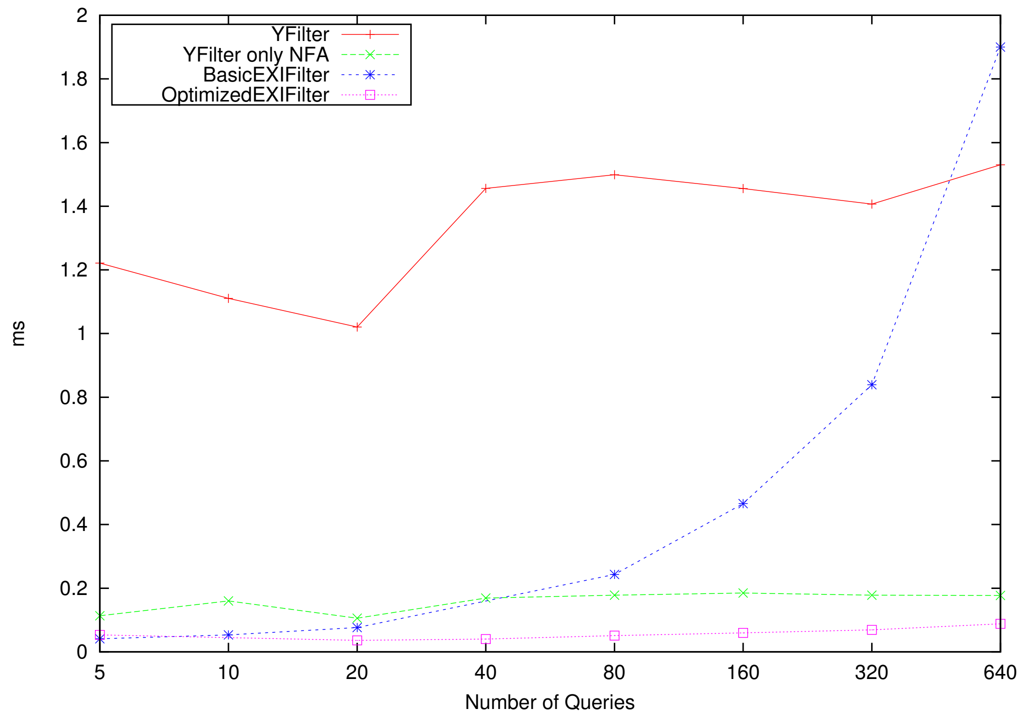


Figure 4.6: Filtering performance results of binary XML filter variants (*BasicEXIFiltering* and *OptimizedEXIFiltering*) and plain-text filter variants (YFilter with and without NFA)

BasicEXIFiltering's performance would be slower than YFilter NFA after only processing 20 queries.

When contrasting *OptimizedEXIFiltering* with the YFilter and YFilter NFA, it becomes apparent that all perform almost constantly within their time levels, even as the number of queries increases. This is explained by two facts: all XPath queries are represented as automata, and if there are duplicated queries (these occurrences arise if the number of query sets increases) do not affect the size of automata and thus automata processing. Only the query registration for the predicate evaluation and the accepting state is required. The other fact is that the data message model of ISO/IEC 15118 is structured to meet the demands of embedded resources. In other words, each message pattern has an almost similar depth in terms of the XML message structure; hence, the complexity of the XPath queries is similar.

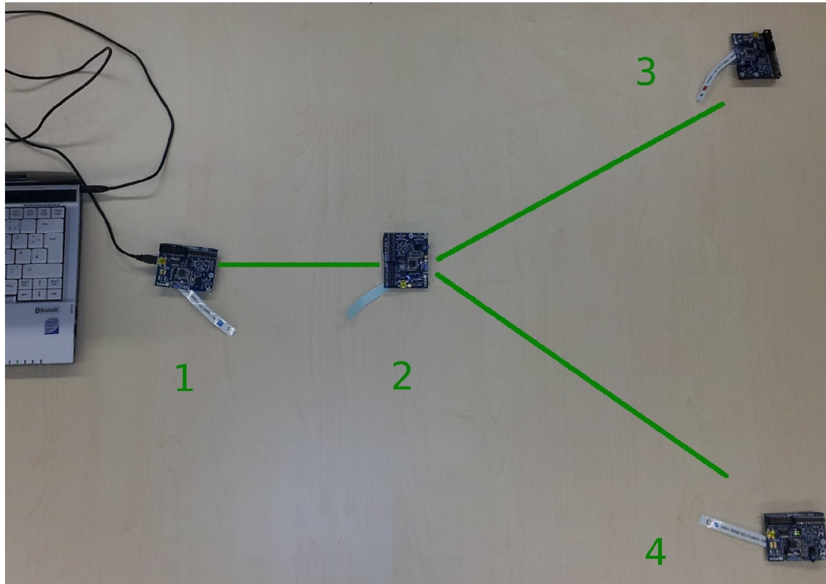


Figure 4.7: Demo embedded network

4.5.3 Demo Network

To evaluate the applicability of these binary XML filtering approaches in the embedded domain, we set up a small embedded network with four wireless, battery-powered evaluation boards from STMicroelectronics, which contains the ARM Cortex-M3 microcontroller presented in the introductory section. Our demo network and its topology can be seen in Figure 4.7.

Each node is running the Contiki OS and communication is based on IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [Montenegro et al., 2007]. Node 1 provides (in 1 seconds intervals) V2G-based messages that are defined by the ISO/IEC 15118 (see Subsection 3.4.1)). For this test, messages with embedded values are randomized. Node 3 and 4 are subscribers of node 1. We set up three different kinds of query scenarios. The first scenario consists of only two XPath expressions, the second one of four, and the last one of three kinds of XPath expressions. This way, every odd query is relevant to client node 3 and every even query is relevant to node 4. In Annex B, we are listing all queries used in these different scenarios.

Based on the XML Schema definition of the V2G data set and the XPath queries provided, we created our binary XML filter mechanisms (*BasicEXIFiltering* and *OptimizedEXIFiltering*) with the code generation variant. These filter mechanisms run in network node 2. Node 2 receives the V2G message from node 1, evaluates the content using the filter, and only forwards

the message to the corresponding node (3 and/or 4) if there is a match.

The demo was successfully executed. At runtime we used a laptop connected to node 1 to monitor the device messages and their values sent to node 2. If there was a match, a LED showed that event. A LED also switched on for nodes 3 and 4 if they received a message that was requested.

Table 4.2 provides an overview of the memory usage (ROM and static RAM) by the filtering setup on node 2 when *BasicEXIFiltering* or *OptimizedEXIFiltering* implementation was used. Please note that all numbers once again include the Contiki OS with the 6LoWPAN communication stack (see Section 3.4.3). It is evident that the memory size of *OptimizedEXIFiltering* is much smaller than that of the *BasicEXIFiltering* approach, independent of the query set scenario. This is due to the fact that the *BasicEXIFiltering* operates on top of the full EXI decoder grammar, even when only a small query set is considered. Meanwhile, *OptimizedEXIFiltering* operates directly within the grammar, and states and transitions not required to evaluate the XPath queries are removed. Consequently, the ROM exhibits a smaller size since the code generated does not contain extra grammar information.

For the *BasicEXIFiltering* variant, besides the predicate data structure, an extra data structure is required that represents all normalized XPath queries with a tracking position as well as type-aware parameters used for evaluating the predicates. Such a complex data structure is not required for the *OptimizedEXIFiltering* variant since the filter grammar is already a representation of all XPath queries. Only type-aware parameters have to be set up at the predicate states (if there are some) and the simple boolean-based predicate table has to be incorporated into the RAM. Thus, RAM usage is better as compared to *BasicEXIFiltering*. Since only two XPath queries were used, the difference is relatively small. It will increase, however, when more XPath queries are taken into account.

	BasicEXIFiltering		OptimizedEXIFiltering	
$\|Q\|$	ROM	RAM	ROM	RAM
2	70600	10260	58508	10244
4	70728	10292	60016	10251
8	71072	10356	61328	10275

Table 4.2: Memory usage (in bytes) of different filter scenarios (2 XPath queries, 4 XPath queries, and 8 XPath queries) compiled for the ARM Cortex-M3 microcontroller.

4.6 Related Work

Efficient filtering, processing, and dissemination of data has been an area of active research for many years. For research in the domain of resource constrained embedded networks, see works such as *TinyDB* [Madden et al., 2005], *Cougar* [Yao and Gehrke, 2002], and for query processing on XML templates objects (XTOs) [Hoeller et al., 2008]. These solutions are mainly focused on sensor networks. Thereby, TinyDB operates in a distributed manner and each sensor node serves a query processor. The Cougar architecture supports in-network computations by the usage of query optimizers on sensor gateways and query proxies on the nodes. Both, however, focus on relational data and data requests are based on SQL-relevant queries. Furthermore, these solutions are narrowed for sensor networks, which will always transmit relevant data to one dedicated root node. In contrast, we consider embedded networks based on sensor, actor, and processing units that follow SOA-based communication with Web services using XML-based data. This enables direct communication within embedded networks; based on our filtering approach, we are able to transmit relevant data to more than one destination node (the clients). Querying XML data is considered in [Hoeller et al., 2008] based on the XML data binding technique called XML templates objects (XTOs). The compressed representation of XTOs and the represented rewriting of XML queries can be compared to our presented normalization mechanism of XPath queries. However, our approach has a complete compact representation of XPath queries that also takes into account the descendant-or-self axis, wildcard, and predicate expressions including operations for efficient evaluation. This also includes a standardized mechanism of unique ID assignments for evaluation that is provided by the EXI standard. Furthermore, we are able to evaluate XML-based instances of relevance directly on the EXI stream and do not need another processable representation of XML contents such as it is the case with XTOs.

Beyond the domain of embedded networks, there are well-known filter mechanisms that come with publish-subscribe systems, such as Gryphon [Banavar et al., 1999] and SIENA [Carzaniga et al., 2001]. Neither focuses on XML-based data filtering, however. Consequently, below, we will concentrate on related works that use filtering mechanisms based on XML. This includes approaches such as *XFilter* [Altinel and Franklin, 2000], *XTrie* [Chan et al., 2002], *YFilter* [Diao et al., 2003, Diao and Franklin, 2003], and *Lazy DFA* [Green et al., 2003, Green et al., 2004]. Since XFilter, YFilter, and Lazy DFA use an automata approach, we will explain these solutions in further detail to understand how they differ from our approaches presented in this

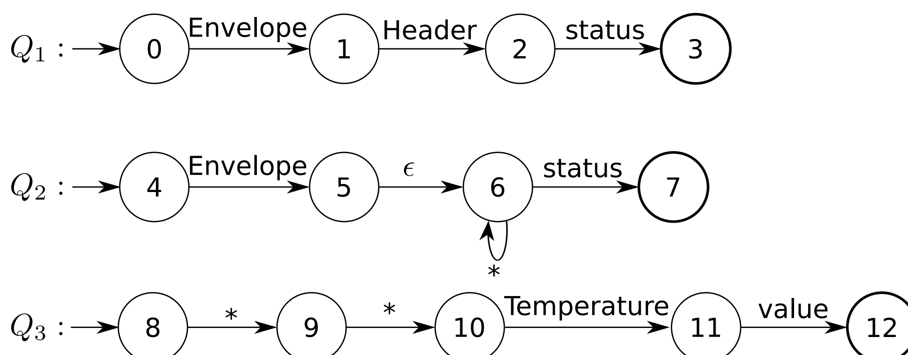


Figure 4.8: Example of XFilter NFAs

chapter.

The XFilter [Altinel and Franklin, 2000] belongs to the first XML filtering algorithm utilizing a NFA approach. The idea is to build a separate NFA for each XPath query which is simultaneously executed during processing of the input XML document. A query match is reached when an accepting state is reached. Figure 4.8 shows such a set of NFAs based on the following 3 XPath queries

$$Q_1 = /Envelope/Header/status$$

$$Q_2 = /Envelope//status$$

$$Q_3 = / * / * /Temperature/value$$

Query Q_1 addresses the path of element nodes which have to occur for requesting the status node element. This linear path is sequentially mapped onto the transitions that can be seen in the first NFA. A *descendant-or-self* expression in an XPath query, as can be seen in Q_2, is represented as a '*' loop initiated by ϵ-transition to a new state. Generally, '*' denotes any element at the particular node or state, respectively. E.g., Q_3 accepts any first 2 elements; however, at the beginning of the third location step the *Temperature* node and its child node *value* are accepted.

When a document arrives at the XFilter engine, it is run through an XML parser based on a SAX (see Section 2.3.2) interface which then drives the process of checking one or more queries. Let us assume that the XML message instance shown in Listing 2.9 is present. When the SAX parser encounters the *startDocument* event, states 0, 4, and 8 will be accessed. In the case of *startElement* event of *Envelope* is encountered by the XML parser, states 1 and 5 as well as 9 is accessed du to the *any element* condition. This process

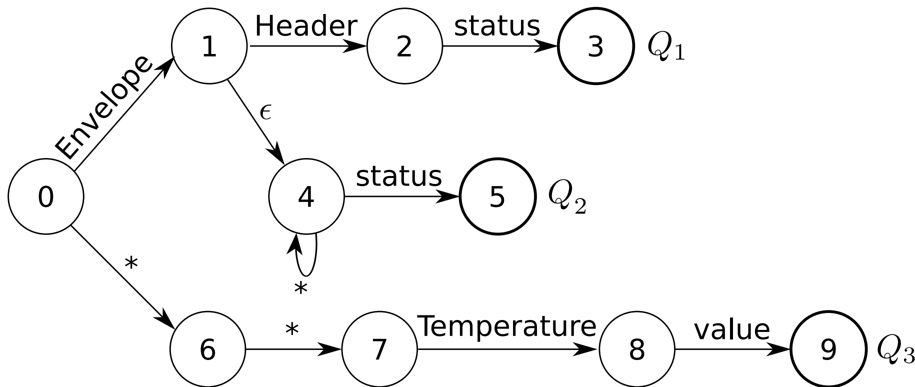


Figure 4.9: Example of a YFilter NFA engine

continues until an accepting state has been reached; then, the corresponding query matches the XML message. An *endElement* event causes backtracking in the NFAs. More precisely, the precursor state in the automata path is visited again. If the *endDocument* event is encountered, only those queries are seen as matches that reach corresponding accepting states during the parsing process of the input XML message. Otherwise, there are one or more mismatches.

It quickly becomes apparent that the more queries are present the higher the number of NFAs. Thus, the worst-case scenario during the execution process would involve a SAX event triggering state transitions for each NFA [Silvasti, 2011]. Similar to the *BasicEXIFiltering* approach presented in this chapter, as long as there are active XPath queries (no match was found thus far) all of them will be checked for a local match. In contrast, however, we operate on binary XML with an efficient EXI automata representation and the evaluation is performed on top of the EXI automata. Instead of comparing string-based element and attribute names (which would provide the XFilter SAX interface) we are able to use ID numbers for comparison as well as type-aware predicate evaluation.

A new filter approach based on NFAs was presented in [Diao et al., 2003, Diao and Franklin, 2003]: the YFilter. YFilter is a successor of XFilter. It revolves around the idea of building only one NFA for all given XPath queries. Thereby, YFilter uses a mechanism that exploits commonalities among path queries by merging the common prefixes of the paths so that they are processed once at most [Diao and Franklin, 2003]. Figure 4.9 shows a sample YFilter NFA based on the same queries that were applied to the XFilter variant.

Queries Q_1 and Q_2 address the same node element *status*; they only differ

in the second location step expression. In general, the YFilter construction algorithm builds shared transitions for as long as there is no difference in the location step between all given XPath queries. Thus, Q_1 and Q_2 share the *Envelope* transition. Since the first location step of Q_3 addresses all possible elements, it requires its own ϵ-transition. As can be seen, the same NFA fragments ('*', '//', etc.) are used here as were used for the XFilter NFAs.

Like XFilter, the YFilter evaluator is based on SAX events encountered by an input XML message. E.g., a *startDocument* event would enter state 0. Typically, efficient implementations use a stack which always pushes the states currently visited onto the stack. For the message shown in Listing 2.9 the *startElement Envelope* would causes traversal of the transitions labeled *Envelope* and *. Thus, states 1 and 6 would be pushed onto the stack, and so on and so forth.

Based on the idea of identifying paths shared by the input XPath queries, the YFilter NFA returns a relatively small number of machine states that can be more efficiently processed than XFilter [Diao et al., 2003]. Having XPath queries represented by only one automata is comparable to our *OptimizedEXIFiltering* approach. However, the YFilter (as well as the XFilter) approach separates the processes of parsing of the input message and executing the query NFA for evaluation (based on the SAX events encountered). Our *OptimizedEXIFilering* approach evaluates at the time of parsing; thus, we can immanently decide whether the message is relevant to one or more queries.

XFilter and YFilter are based on a NFA approach. An XML filter processor based on a Deterministic Finite Automaton (DFA) approach was presented in [Green et al., 2003, Green et al., 2004] and is called *Lazy DFA*. The main idea is to create NFAs for each XPath expression (similar to XFilter). These NFAs are transformed into a single NFA (similar to YFilter). Finally, based on this NFA, the DFA will be constructed. Thereby, the DFA is constructed *lazily* which means that states and transitions are determined from the corresponding NFA at runtime. A new entry in the transition table or a new state is computed only when the input XML message requires the DFA to follow that transition or to enter that state [Green et al., 2004]. In general, the concept of using a DFA to filter messages is comparable to our approach, which uses EXI DFAs. However, applying *Lazy DFA* to our constrained embedded network environment is not possible because the dynamic constructing of the DFA at runtime requires heavy memory usage and processing overhead [Chen and Wong, 2004]. Aside from handling plain-text XML data, these aspects are not compatible with constrained embedded devices.

	BasicEXIFi.	OptimizedEXIFi.	YFilter
Filter grammar construction	no	top-down	button-up
Grammar-based evaluation	no	yes	yes
XSD informed	yes	yes	no
Validation of XPath expressions	yes	yes	no
Early runtime evaluation	no	yes	no
Evaluation and decoding in one step	no	yes	no
Query extensions at runtime	yes	no	yes

Table 4.3: Comparision between BasicEXIFiltering, OptimizedEXIFiltering, and YFilter

4.7 Summary and Comparison

In this section we presented resource-efficient approaches that enable us to filter binary XML-based messages. The mechanism can be applied to embedded networks and supports reduction of network traffic; it also avoids processing overheads of messages that are not relevant to particular embedded nodes. We presented two approaches: *BasicEXIFiltering* and *OptimizedEXIFiltering*. The *BasicEXIFiltering* operates on top of an EXI grammar and evaluates normalized XPath queries by means of binary XML. This enables fast local match identification as well as type-aware predicate evaluation. *OptimizedEXIFiltering* presents a more sophisticated approach; it maps all XPath expressions within an EXI grammar. This enables evaluation at time of parsing. This speeds up evaluation time and reduces memory usage since the full EXI grammar does not have to be present at runtime. To the best of our knowledge, we are the first that propose such kind of approaches for binary XML filtering based on the EXI format that is also applicable in microcontroller domain.

We will conclude this chapter by comparing our approaches to the well-known YFilter that was used in our evaluation. The comparison is consolidated in Table 4.3. The BasicEXIFiltering approach does not construct a filter grammar for the XPath query evaluation. Instead, the evaluation is done *on the top* of the EXI grammar by using the StAX event sequences.

In contrast, the *OptimizedEXIFiltering* constructs a filter grammar based on a complete EXI grammar. This automata grammar is reduced by removing transitions and states (top-down). Only those fragments that reflect all given XPath expressions remain. YFilter starts from scratch to construct the automata grammar and always extends it by new XPath queries (button-up). Our approaches use the XSD knowledge of the underlying service data. This has the advantage of type-aware predicate evaluation and this enables a (pre-)validation of XPath queries. The validation of a given XPath expression in a particular messaging context can be accomplished through a successful XPath normalization (*BasicEXIFiltering*) or through successfully transforming the XPath expression within the corresponding EXI grammar (*OptimizedEXIFiltering*). In the case of nonsuccess, the XPath is not applicable to any of the messages that the XML Schema can instantiate. Thus, at runtime, this XPath query will never find a match. *OptimizedEXIFiltering* provides an exclusive mechanism for an early runtime evaluation by interrupting the decoding of a sample message when no or further queries can be evaluated successfully (see the example in Section 4.4.4). This is mainly possible because the evaluation is performed during the same step as decoding, and based on the filter grammar constructed, it can be identified whether any accepting states can be reached or not. Given the pruning of the underlying EXI grammar to a filter grammar, new XPath queries for evaluation can generally not simply be integrated at runtime. This is due to the fact that automata fragments may be missing which the new query might want to address. *BasicEXIFiltering* provides more flexibility; it would only require a new structure instance that represents the new XPath expression. Due to its button-up approach, YFilter would also simply extend the current filter automata.

Chapter 5

Advanced Filter-Enabled Service Data Dissemination

5.1 Introduction

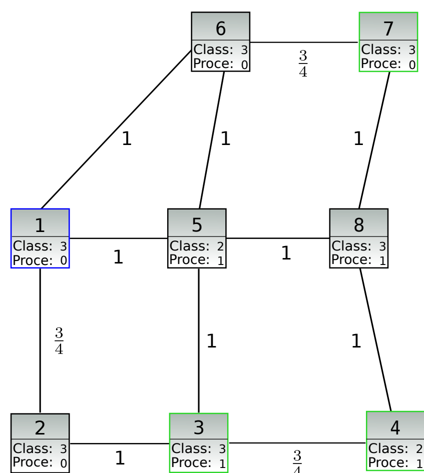
A filter-enabled subscribe mechanism in embedded networks reduces network traffic and unnecessary message processing at the client nodes. It optimizes data interaction between the service provider and service requester in terms of data novelty and supports an efficient execution of applications in embedded networks at runtime. In the previous chapter, we introduced the functionality to create an efficient filter mechanism for binary XML data based on a number of service requesters by providing XPath expressions that address the desired service data occurrences and/or data value conditions. An immediate evaluation at the node of service data origin would prevent dissemination of data when it does not fall within the scope of one or more service requesters.

In the context of constrained embedded networks, however, a desired early evaluation at the node origin can sometimes not be realized. Two aspects support this observation: First of all, the current capabilities of resources, especially when it comes to memory, are not sufficient enough for installing an additional filter application. Secondly, vendors of embedded nodes do not offer such installation opportunities. Figure 5.1(a) depicts such a scenario. Nodes 3, 4, and 7 request service data from node 1. To take into account these particular client demands - the persistent queries - a corresponding filter mechanism cannot be placed at node 1 due to the absence of processing resources ($p(v_1) = 0$). To avoid the returning to the situation discussed in Figure 4.1(a), we propose an alternative filter placement in the network which evaluates the data for relevance for one or more clients. Figure 5.1(b)

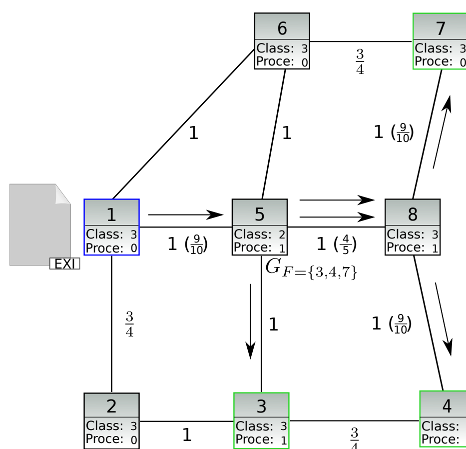
shows the effect of the placement of filter $G_{F=\{3,4,7\}}$ on node 5. The service running on node 1 is set up to always transmit new data events to node 5. Node 5 evaluates the received data content by using $G_{F=\{3,4,7\}}$ if it is relevant for node 3, node 4 and/or node 7. If values range between 20.5 and 21.0 node 5 will forward the data twice via node 8, once for node 4 and once for node 7. To improve network traffic, a sophisticated approach would include forwarding such a service data constellation to node 8 only once, while node 8 disseminates the service data, based on relevance, to the final destination nodes. Doing so, a sub-filter $G_{F'=\{4,7\}}$ is placed at node 8 only reflects the condition of node 4 and 7. In addition, filter G_F is updated to forward the data to node 8 only once. Figure 5.1(c) depicts this approach.

Depending on the application executing the determined data dissemination path, at runtime this may affect any involved network connections as well as the resource's nodes at runtime. For instance, if node 1 sends a service data message very frequently (e.g., 10kHz) and the content is relevant for each client node, this would affect, for example, the bandwidth within the determined and *reserved* dissemination paths. For scenarios putting such demands on resources, we simply reduced the connection quality by $\frac{1}{10}$ for each involved connection link along the route as an example (shown in brackets in Figure 5.1(b) and 5.1(c)). In addition, to illustrate the possible impact of filter installations, in node 8 with a class 3 property that runs now a (sub-)filter, the p function is set to 0 to signal that there are no further resources for future application installations. However, we take this into account to reduce the overall processing load and to reduce the network traffic. The result can be seen in Figure 5.1(d), which reflects the current resource capability and is the basis for the next possible application set up in the network.

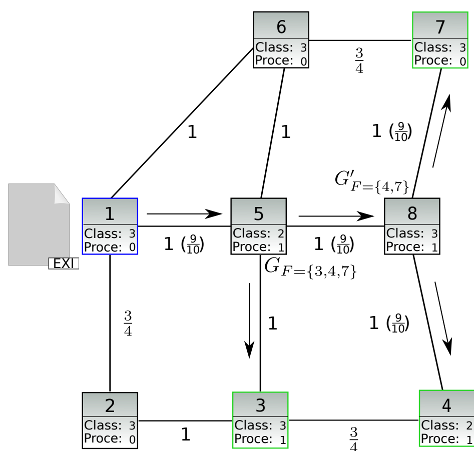
In this chapter we introduce an approach for organizing a filter-based service data dissemination in constrained embedded networks that takes network resources into account such as device classes, device processability, as well as the connection quality between nodes. The goal is to share relevant service data using binary XML as long as possible through using filters and sub-filters. This reduces both network traffic and overall node processing. Partially, this leads to content-based routing on application level based on binary XML content. In Section 5.2 we begin to formalize the problem of an optimized filter-enabled dissemination path and introduce our cost model. Section 5.3 introduces our filter-enabled dissemination algorithm that takes a new application constellation and provides a dissemination path that involves dedicated filter nodes. An execution example is given in Section 5.4 and improvements and extensions of our approach are discussed in Section 5.5. Finally, Section 5.6 shows some evaluation results that demonstrate the benefits of our filter-enabled and shared service data dissemination approach.



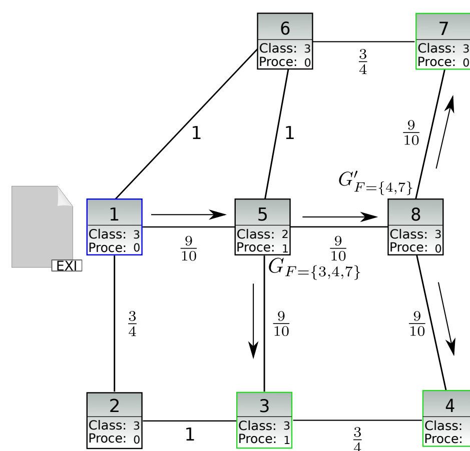
(a) A filter placement at service data origin (node 1) is not possible due to the lack of processing resources ($p(v_1) = 0$).



(b) Filter placement for $G_{F=\{3,4,7\}}$ at node 5. Relevant data for 4 and 7 is forwarded twice from node 5.



(c) A second filter (sub-filter) $G_{F=\{4,7\}}$ at node 8 shares the data for one more connection from node 5.



(d) Applied dissemination scenario and its impact on network's resources.

Figure 5.1: Optimized service data dissemination example

5.2 Problem Statement and Formalization

5.2.1 Notations and Definitions

In this section we are formalizing the stated optimization problem. As our foundation we are using the definition of an embedded network (see Definition 1.1) as presented in Section 1.1: $N_{emb} = (V, E, w, c, p)$. Let us assume that we are going to install a new application with one or more service subscribers (clients) of a Web service within N_{emb}. Clients are denoted by the set C with $C \subseteq V$ and the service by v_s with $v_s \subseteq V$. $D = v_s \cup C$ describes the combination of these emphasized nodes. A service data dissemination path from v_s to all client subscribers in C is comparable to a *spanning tree* [Prim, R. C., 1957, Cormen et al., 2009]. We will call a subnetwork of N_{emb} a tree T when each node is connected and has no cycles in the subnetwork. For our purposes, we define a tree network as follows:

Definition 5.1 (Tree Network N^T_{emb})
A tree (embedded) network of a given embedded network $N_{emb} = (V, E, w, c, p)$ is a connected and cycle free subnetwork $N^T_{emb} = (V^T, E^T, w^T, c, p^T)$ where is

1. $V^T \subseteq V$,
2. $E^T \subseteq E$, and
3. for each $(v_i, v_j) \in E^T$ is $w^T(v_i, v_j) \leq w(v_i, v_j)$ and for each $v \in V^T$ is $p^T(v) \leq p(v)$.

Property 3 points to the possible impacts for when a connection and a device node are members of a spanning subnetwork in terms of connection quality and processability. The connection quality w^T takes the maximum value of w. Similarly, the discrete processability values of p^T are the same of p or if they differ, the result is always $p^T = 0$. We will discuss the influence of both of these functions in more detail in the next subsection. The function c is seen as static and serves always the same result as in N^T_{emb}. This is justified by the fact that the device class of each node is considered as a constant over the time of a configured embedded network.

In each determined tree N^T_{emb} we dedicate a particular node as a root node. We then consider its successor paths or branches as service data dissemination direction. As discussed in the introduction to this chapter, we are using filters and sub-filters in an embedded network to enable a shared service data dissemination. Bellow, we introduce and define a pre-filter and a post-filter.

Definition 5.2 (Pre-Filter und Post-Filter)
Let Q be a set of queries of service data requesters. A filter grammar G_F is called a pre-filter if it reflects all queries in Q. We will call a filter grammar G'_F a post-filter when it reflects a subset of Q.

A node that runs a pre-filter is denoted as v_{pre} and a node which runs a post-filter is denoted as v_{post}. Before we close this subsection, let us introduce and define a cost network:

Definition 5.3 (Cost Network N^C)
The cost network of a given embedded network $N_{emb} = (V, E, w, c, p)$ is a complete network $N^C = (V, E^C, g)$ where each connection $(v_i, v_j) \in E^C$ is represented by the lowest cost value g_{v_i,v_j} from v_i to v_j in N_{emb}. We call a $N^C = (S \cap V, E^C, f)$ a connected induced sub-cost-network for which $S \subset V$.

N^C will support us later for finding a suitable dissemination path that takes into account connection quality as well as device properties.

5.2.2 Cost Function

A newly installed application in an embedded network would typically lead to additional network traffic and processing costs. To keep this overhead as small as possible, we filter to determine all relevant data and share this data as long as possible on a determined dissemination path that avoids constrained device class nodes and uses connections with relatively good quality. Consequently, we have two metrics which have to be considered: device class and connection quality. Let us first consider the device class component.

To determine the class appearance of a determined tree-based path reflected by a subnetwork N^T_{emb} we can use

$$\sum_{v_i \in V^T} c(v_i) \quad . \tag{5.1}$$

A path with a relatively high ratio of low device classes leads to smaller values compared to a path consisting of higher device classes. Thus, we are interested in lower values that reflect the absence of constrained embedded devices. Within this metric we will also consider the processability for a successful placement of a pre/post-filter service. Nodes with this capability will be preferred and the Function 5.1 will be extended to

$$\sum_{v_i \in V^T} (c(v_i) - p^T(v_i)) \quad . \tag{5.2}$$

This leads to the goal to have cost values which approximate the value of 0. However, in the case of equal device class characterization of each node in an embedded network we are not able to avoid routes with a relatively high number of hop count. To clarify, let us assume that we have an embedded network constellation as depicted in Figure 5.2 and it is $c(v_1) = c(v_2) = c(v_3) = 1$ and $p(v_1) = p(v_2) = p(v_3) = 1$. A route from v_1 to v_3 can be taken either directly or via node v_2. However, based on the Function 5.2 we are unable to decide which path variant is the best since it is

$$\sum_{v_i \in \{v_1, v_3\}} (c(v_i) - p^T(v_i)) = \sum_{v_i \in \{v_1, v_2, v_3\}} (c(v_i) - p^T(v_i)) \quad .$$

It seems reasonable to use the direct connection to avoid any overheads in the network. So as to take such a decision, we are here introducing a *hop-noise* factor that is added to each device class value:

$$\sum_{v_i \in V^T} (c(v_i) + 1 - p^T(v_i)) \quad . \tag{5.3}$$

Consequently, based on this factor it will always be

$$\sum_{v_i \in \{v_1, v_3\}} (c(v_i) + 1 - p^T(v_i)) < \sum_{v_i \in \{v_1, v_2, v_3\}} (c(v_i) + 1 - p^T(v_i))$$

for arbitrary class c and processability p constellations.

At this point, we are able to select a route in terms of device classes with processability and hop count. Now let us consider connection quality component, which we also have to take into account to achieve optimized service data dissemination.

In general, the function w describes the connection quality between two nodes. Depending on use cases and focus, the value can reflect the bandwidth, latency, packet loss likelihood, and/or physical link quality. Similar to the device's class function above, it is our desire to find a route in which each w pair approximates the value of 0. Thus, to estimate the connection quality of a route given by E^T of a N^T_{emb} we will use the function

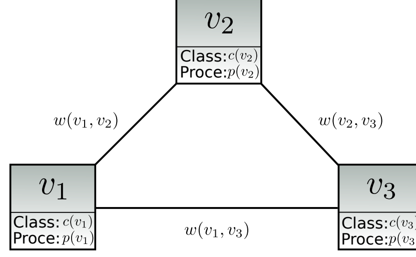


Figure 5.2: Embedded network with 3 nodes

$$\sum_{(v_i,v_j)\in E^T} \frac{1}{w^T(v_i,v_j)} \tag{5.4}$$

with $w^T(v_i, v_j) > 0$ (a connection exists) for all $(v_i, v_j) \in E^T$.

Putting it all together, we can define our cost function f for a given N^T_{emb} which spanning nodes in D:

$$\begin{aligned} f(N^T_{emb}) := \alpha \cdot \sum_{v_i \in V^T} (c(v_i) + 1 - p^T(v_i)) \\ +(1-\alpha) \cdot \sum_{(v_i,v_j)\in E^T} \frac{1}{w^T(v_i,v_j)} \quad . \end{aligned} \tag{5.5}$$

Here, $\alpha \in [0, 1]$ is a weight factor that enables us to set up a more dominant part in the cost function: the device class ($\alpha > 0.5$) or the connection quality ($\alpha < 0.5$). In this thesis, if not stated otherwise, we will always assume that $\alpha = 0.5$. We will abbreviate the cost function and use f_T as cost of a determined tree T of N_{emb} and f_{v_i,v_j} as cost of a determined route from node v_i to v_j in N_{emb}.

Using f we are able to formalize our optimization problem to find a subgraph N^T_{emb} of N_{emb} for a filter-enabled service data dissemination:

$$Minimize \quad f_T \tag{5.6}$$

subject to

$$\sum_{v_i \in V^T} p^T(v_i) \geq 1 \quad .$$

The inequality constraint specifies the occurrence of at least one processable node within N^T_{emb} that can be used to set up a pre-filter.

5.2.3 Complexity

To identify the complexity of the optimization problem we will discuss special constellations of N_{emb} in terms of number of service requesters, processing nodes, device classes, and connection quality, which all have a major impact on both the optimization problem and complexity.

Firstly, let us assume there is a $N_{emb} = (V, E, w, c, p)$ in which each node has endless resource capability ($\forall v_i \in V$ is $p(v_i) = 1$). In the case of a D with cardinality $|D| = 2$ (one service v_s and one client v_c) we encounter the well-known shortest path problem (by using our metrics) with a complexity of $\mathcal{O}(|V|log|V| + |E|)$ [Cormen et al., 2009]. Based on the precondition of N_{emb} we are able to place a filter at the node of v_s which leads to a desired pre-evaluation of relevance for v_c. In the case of $|D| = |V|$ (a service node distributes service data to each node in N_{emb}) we result in the minimum spanning tree problem [Cormen et al., 2009] with a complexity of $\mathcal{O}(|V|lg|V| + |E|)$ to determine N^T_{emb}. We are able to place a pre-filter at the service origin's node and a post-filter at each branch node to share the data in an optimized way.

Now let us consider an embedded network $N_{emb} = (V, E, w, c, p)$ that contains only one single device class variant ($\forall v_i \in V$ is $c(v_i) = 1$). In such a constellation, we can focus on determining a valuable data dissemination route based on connection quality. In such a case, the device class part within the cost function f can be disabled by $\alpha = 0$ and would lead to the single metric Function 5.4, which can then be used to determine the cost of an instance N^T_{emb} of N_{emb}. Let D be the set of service provider and service requesters with the condition $2 < |D| < |V|$ and again, assume that each node has endless resource capability ($\forall v_i \in V$ is $p(v_i) = 1$), then this leads to the *Steiner Tree Problem* [Prömel and Steger, 2002] for finding a subgraph N^T_{emb} spanning the terminal nodes in D. The *Steiner Tree Problem* is well-known as *NP-complete* and it is covered in *Karp's 21 NP-complete problems* [Karp, 1974]. The proof is typically shown by reduction of the *3SAT* problem [Prömel and Steger, 2002].

Consequently, for any constellation in N_{emb} and for an arbitrary set D with $2 < |D| < |V|$ we are not able to find an optimized solution in polynomial time. In the next section we are going to present a heuristic approach based on greedy algorithms [Cormen et al., 2009] that approximate an optimized N^T_{emb} for a filter-enabled service data dissemination.

5.3 Filter-enabled Dissemination Algorithm

Based on the complexity discussed previously, we are interested in developing an efficient heuristic algorithm to find good approximate solutions of arbitrary N_{emb} with different service and client constellations in N_{emb}. We are now going to describe our filter-enabled service data dissemination algorithm, the *FilterEnabledDissemination* algorithm (see Algorithm 5.1), for installing a new application with a service provider and a number of service requesters, which takes into account the current resources of the embedded network.

Algorithm 5.1 $FilterEnabledDissemination(N_{emb}, v_s, C, Q, XSD, \alpha)$

Input: $N_{emb} = (V, E, c, w, p)$, a service provider v_s, set of service requesters $C = \{v_{c_1}, ..., v_{c_n}\}$, set of queries Q related to client's conditions, data model represented as XML schema XSD, and an α as metric weighting factor.

Output: Tree network N^T_{emb} with a set F of dedicated selected nodes with its filter grammars (pre- and post-filters).

1: $G_F \leftarrow \emptyset$; $E^R \leftarrow \emptyset$;
2: $G_F \leftarrow FilterGrammar(G, Q)$;
3: $\{v_{pre}, E^R\} \leftarrow ClosestPreFilterNode(N_{emb}, v_s, C, G_F, \alpha)$;
4: $N^T_{emb} \leftarrow DisseminationTree(N_{emb}, v_{pre}, C, \alpha)$;
5: $F \leftarrow PostFilterPlacement(N^T_{emb}, Q, v_{pre}, G_F, XSD)$;
6: $extendTreeByPreRoute(V^T, E^T, V, E^R)$;
7: **return** $\{N^T_{emb}, F\}$

As input, the algorithm takes an embedded network N_{emb}, a dedicated service provider node v_s, a set of service requesters (the clients) C and their corresponding queries Q, the underlying data model of the service provider described in an XML schema XSD, and the metric weighting factor α (see Section 5.2.2). Its outcome is a subnetwork N^T_{emb} of N_{emb} that represent the dissemination tree/path from v_s to all clients in C and a set F that consists of the selected nodes with pre- and post-filter properties. Essentially, the processing steps of the algorithm can be divided into three parts:

1. After determining the filter grammar G_F by the *FilterGrammar* (see Algorithm 4.8) in line 2, a suitable pre-filter node is searched. Doing this, the *ClosestPreFilterNode* algorithm (line 3) is called which will be explained in further detail in Section 5.3.1.

2. Starting with the determined pre-filter node v_{pre} we discover an optimized dissemination tree N^T_{emb}. The *DisseminationTree* algorithm

(line 4) will be called to gather such a tree. A detailed explanation will be provided in Subsection 5.3.2.

3. Based on N^T_{emb} suitable nodes are selected for the post-filter functionality to share service data as long as possible. The *PostFilterPlacement* algorithm (line 5) realizes this and provides the routing information for all filter grammars. Section 5.3.3 features a detailed explanation of this algorithm.

Before the *FilterEnabledDissemination* algorithm terminates, we extend N^T_{emb} by the involved nodes and connection (given by E^R) that leads from v_s to v_{pre} (line 6).

5.3.1 Closest Pre-Filter Node Algorithm

One of our most important goals is to evaluate the data as soon as possible, starting at the source origin. An early evaluation based on a filter grammar G_F has the advantage of avoiding the distribution of service data that would never affect any client requesters. Considering that case, it is desireable to determine one processing node v_{pre} that is able to run the filter grammar G_F as well as that results in overall positive data dissemination. More precisely, we are not only considering the quality of the path to a processable node v_{pre} in terms of connection and device class, but also the quality from v_{pre} to all service subscribers. Figure 5.3 clarifies this point: Nodes 4 and 5 are service client subscribers of node 2. Let us assume that a pre-filter grammar G_F is able to run on the processable nodes 1, 3, and 4. Applying our cost function f, for the path from 2 to 1 we would achieve a cost value of

$$\begin{aligned} f_{2,1} &= \frac{1}{2} \cdot ((c(2) + 1 - p(2)) + (c(1) + 1 - p(1))) + (1 - \frac{1}{2}) \cdot w(2,1) \\ &= \frac{1}{2} \cdot ((3 + 1 - 0) + (2 + 1 - 1)) + \frac{1}{2} \cdot 1 \\ &= 3,5 \quad . \end{aligned}$$

For the path from 2 to 3 the cost value is $f_{2,3} := 3,6\overline{6}$ and for the path from 2 to 4 the cost value is $f_{2,4} := 5,6\overline{6}$. Thus, it would be logical to select node 1 as the appropriate pre-filter node v_{pre}. Based on the topology of the example embedded network, however, relevant data for the clients always has to be routed via the service data origin which is node 2. So as to estimate the overhead for such scenario, we determine the accumulated costs from a candidate node for a pre-filter installation to all client subscribers multiplied by the cost from the service data origin. In the case of node 1, the result

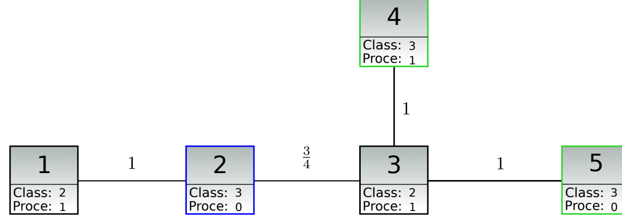


Figure 5.3: Embedded network with three processing nodes (1, 3, and 4)

would be for $f_{1,4} = 7,1\overline{6}$, for $f_{1,5} = 7,6\overline{6}$, and hence $(f_{1,4}+f_{1,5}) \cdot f_{2,1} = 54,39$. Considering node 3 as a pre-filter candidate, we get $(f_{3,4} + f_{3,5}) \cdot f_{2,3} = (3+3,5) \cdot 3,6\overline{6} = 23,8\overline{3}$. Finally, in the case of node 4, we get $(f_{4,4}+f_{4,5}) \cdot f_{2,4} = (3+5,5) \cdot 5,6\overline{6} = 48,1\overline{6}$. Based on this findings, we are going to select node 3 as a propertied candidate on which to place the pre-filter grammar G_F.

The *ClosestFilterNode* Algorithm (see Algorithm 5.2) takes all aspects discussed above into account. It takes an embedded network N_{emb}, the dedicated source node v_s, the set of client subscribers represented by C, and a predetermined filter grammar G_F constructed on the clients request (filter) conditions. The outcome is the propertied pre-filter node v_{pre}. In line 2, the algorithm checks whether the source node v_s is already a suitable processing node for G_F. The function *isProcessable* tests if there are enough resources for applying G_F on v_s. Is this test is successful, then the algorithm will terminate here, since the most suitable place for a pre-filter is at source origin, as we mentioned above. If the test is negative, we then have to find a suitable processing node that is close by. Doing so, we determine the path and cost for each candidate v that is processable from v_s (lines 5-6) as well as the overall cost to the service requester in C (lines 7-9). The current best node will be updated in v_{pre} (lines 11-13). In order to determine the cost values in term of device class and connection quality (DCCQ) and the involved edges E^T that lead to an optimized v_{pre} we then apply the *BestDCCQRoute* algorithm (see Algorithm 5.3).

The *BestDCCQRoute* algorithm follows the basic concept of the *Dijkstra* algorithm [Dijkstra, 1959, Cormen et al., 2009]. As input, the algorithm takes an embedded network N_{emb} with a start node v_i, a destination node v_j, and an α as a metric weighting factor. It returns a set of the best *DCCQ* route by describing it as a subset of edges E^T and the cost. The first investigation of the *BestDCCQRoute* is the initialization of the working parameters of set Q, successor vector s, and the current *DCCQ* vector d (lines 1-3). At runtime, s will provide the latest best route from any visited nodes to v_i. Similarly, d provides the latest *DCCQ* information for each visited node from source node v_i. Within the loop (line 4), each time a node u is determined that

Algorithm 5.2 $ClosestPreFilterNode(N_{emb}, v_s, C, G_F, \alpha)$

Input: $N_{emb} = (V, E, c, w, p)$, a service provider v_s, set of service requester $C = \{v_{c_1}, ..., v_{c_n}\}$, G_F filter grammar based on clients conditions, and an α as metric weighting factor.
Output: Pre-filter node v_{pre} and route E^T from v_s to v_{pre}

1: $k \leftarrow \infty$; $g \leftarrow 0$; $l \leftarrow 0$; $m \leftarrow 0$; $v_{pre} \leftarrow \emptyset$; $E^T \leftarrow \emptyset$;
2: **if** $isProcessable(v_s, G_F)$ **then**
3: $\quad v_{pre} \leftarrow v_s$;
4: **else** $\{v_s$ is not a processing node$\}$
5: $\quad$ **for all** $v \in V \backslash v_s$ and $isProcessable(v, G_F)$ **do**
6: $\quad\quad \{E^T, g\} \leftarrow BestDCCQRoute(N_{emb}, v_s, v, \alpha)$;
7: $\quad\quad$ **for all** $c \in C$ **do**
8: $\quad\quad\quad \{\emptyset, m\} \leftarrow BestDCCQRoute(N_{emb}, v, c, \alpha)$;
9: $\quad\quad\quad l \leftarrow l + g \cdot m$;
10: $\quad\quad$ **end for**
11: $\quad\quad$ **if** $k > l$ **then**
12: $\quad\quad\quad v_{pre} \leftarrow v$;
13: $\quad\quad\quad k \leftarrow l$;
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: **end if**
17: **return** $\{v_{pre}, E^T\}$

has the best *DCCQ* route from v_i so far (line 5). This node is used to check its direct neighboring nodes and to calculate how the *DCCQ* value will be affected when the route is continued to u's neighbors (lines 6-8). If the evaluation is positive, meaning that there is a better *DCCQ* cost value from v_i via u to a u's neighbor v, the current *DCCQ* vector d will be updated with the new cost value (line 9). Furthermore, we memorize the new (better) route node by updating the successor vector s with this new information (line 10). Since each u that is determined in line 5 is removed from the working node set Q (line 13), the *loop* will be terminated after $|V|$ rounds. After completing the loop, the successor vector s will contain the route with the best *DCCQ* value from the source node v_i to the client node v_j. Based on the constellation in s, the set E^T will finally be constructed (starting from $s[v_d]$) and return with the DCCQ cost value (lines 14-15).

Algorithm 5.3 *BestDCCQRoute*$(N_{emb}, v_s, v_d, \alpha)$

Input: Embedded network $N_{emb} = (V, E, c, w, p)$, start node v_s, destination node v_d, and a metric weighting factor α
Output: Best device class and connection quality route E^T and the cost
1: $Q \leftarrow V$; $m \leftarrow 0$;
2: Let s the successor node vector; init with $s[v] \leftarrow -1$ for all $v \in V$
3: Let d the current $DCCQ$ vector from v_s; init with $d[v_s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V \backslash v_s$
4: **while** $Q \neq \emptyset$ **do**
5: Let be u the node where $\forall v \in Q$ is $d[u] \leq d[v]$
6: **for all** $v \in V$ with $w(u, v) > 0$ **do**
7: $m \leftarrow \alpha \cdot (c(v) + c(u) + 2 - (p(v) + p(u))) + \frac{(1-\alpha)}{w(u,v)}$;
8: **if** $d[v] > (d[u] + m)$ **then**
9: $d[v] \leftarrow d[u] + m$;
10: $s[v] \leftarrow u$;
11: **end if**
12: **end for**
13: $Q \leftarrow Q \backslash u$;
14: **end while**
15: $E^T \leftarrow constructRoute(s, v_d)$;
16: **return** $\{E^T, d[v_d]\}$

5.3.2 Dissemination Tree

At this point we have determined the most suitable pre-filter node v_{pre}. Relevant service data shall be delivered from v_{pre} to the service subscribers in a resource-optimized manner. More precisely, the data shall be routed via high quality connections, avoid very constrained embedded devices, and be shared for as long as possible if there are multi client destinations. The latter can be fulfilled when one or more post-filters can be placed that retain the information of the final client destination nodes or the next post-filter nodes. Consequently, it is desireable to find a dissemination tree from v_{pre} to all clients in C that takes into account the device class and connection quality metrics as well as the current processability of the potential post-filter placement.

Below we present the *DisseminationTree* algorithm (see Algorithm 5.4) which constructs such a dissemination tree. More precisely, it will construct an N^T_{emb} from N_{emb} that spans the involved nodes, namely Q with $Q = v_{pre} \cup C$. Our algorithm is based on the concept of the Kou-Markowsky-Berman (KMB) algorithm [Kou et al., 1981] which is a well-known heuristic

for the Steiner Tree problem.

First, we are setting up an induced sub-cost-network $N^C = (Q, E^C, g)$ (see Definition 5.3) in lines 4-9. Thereby, we consider and construct each node combination (u, v) with $u, v \in Q$, determine its best DCCQ route from u to v (line 6) based on the input network N_{emb} (see Algorithm 5.3), and assign the cost value m to the cost value function g (line 8) as well as memorize the route E^T in E^S (line 9). The next major step in the algorithm is the determination of the best DCCQ tree E^T by the *BestDCCQTree* algorithm (see Algorithm 5.5) of the cost network N^C (line 12). Doing this, *BestDCCQTree* will follow the idea of the prim algorithm [Prim, R. C., 1957, Cormen et al., 2009] and will be explained below.

The best DCCQ tree represented by E^T may contain connections which do not exist in the original input E of N_{emb}. Those types of connections will be identified and replaced by the best DCCQ route (lines 14-16). Based on the cost network construction process above, the best DCCQ route is already present in the set E^S between u and v with $u, v \in Q$. The function *replaceByBestDCCQRoute* (line 16) takes a connection (u, v) which is not member of E and replaces it in E^T with the best DCCQ route that is kept in E^S. The node set V^T will be updated in this procedure with the intermedia nodes that are involved in the best DCCQ route. At this stage we have a N^T_{emb} that is a sub network of N_{emb}, however, it does not necessarily have the tree property for our service data dissemination. This time, the best DCCQ tree is determined based on the current status of N^T_{emb} and its connection set E^T is updated (line 19). As mention above, the *BestDCCQTree* algorithm is based on the minimizing spanning tree concept and will return a tree that spans all nodes in the current V^T. However, we are interested in a tree which only spans the nodes that are members of Q since we want to deliver the service data to the corresponding service requesters only. Thus, the final step in the *DisseminationTree* algorithm is removal of all node leaves $v \notin Q$ and their branches to get the desired spanning tree (lines 20-21). The function *removeLeafAndBranch* takes a node that is a leaf of the current E^T and identifies the branch involved in E^T that routes to v. E^T and V^T will be updated by removing this branch and leaf.

The *BestDCCQTree* algorithm takes an embedded network N_{emb} or a cost network N^C, a start node v_s, and a metric weighting value α. The outcome will be a spanning tree E^T that spans all nodes in a given V. The procedure is similar to the Prim algorithm [Prim, R. C., 1957, Cormen et al., 2009] which, however, applies our cost function and involves the different kinds of network constellations. As with the *BestDCCQRoute* algorithm (see Algorithm 5.3) we employ two working vectors: s which registers the successor node, and d, the DCCQ vector that holds the current DCCQ cost

value to a particular node from and within the latest constructed tree in s (lines 2-3). In each round, we determine the current smallest DCCQ node u within the working set Q (line 5) and check all its directly connected neighbors if the current DCCQ cost value m (determined in line 8 or 10) from u to v is lower compared with the one that is kept in d (line 12). If so, we will update the cost value in d and memorize v as a successor node of u (lines 13-14). Please note that m is determined based on the given network. In the case of a cost network N^C we already have the cost value given by the function g for each connection (line 10). In contrast, a given N_{emb} does not include this information and it has to be calculated based on our cost metric (line 8). After checking all nodes in Q we then finally construct the E^T based on the function *constructTree* that takes the successor vector s (line 19).

Algorithm 5.4 $DisseminationTree(N_{emb}, v_{pre}, C, \alpha)$

Input: Embedded network $N_{emb} = (V, E, c, w, p)$, pre-filter node v_{pre}, set of client nodes C, and a metric weighting factor α

Output: A tree $N^T_{emb} \leftarrow (V^T, E^T, c, w, p)$ which spanning $v_{pre} \cup C$

1: $Q \leftarrow v_{pre} \cup C$; $m \leftarrow 0$; $E^C \leftarrow \emptyset$; $V^T \leftarrow \emptyset$; $E^T \leftarrow \emptyset$; $E^S \leftarrow \emptyset$;
2: Let g a cost value function with $g : Q \times Q \rightarrow \mathbb{R}_{\geq 0}$; init with $g(u, v) \leftarrow \infty$ for each $v, u \in Q$ pair
3: $N^C \leftarrow (Q, E^C, g)$;
4: **for all** $v \in Q$ **do**
5:   **for all** $u \in Q \backslash v$ and $\{u, v\} \notin E^C$ **do**
6:     $\{E^T, m\} \leftarrow BestDCCQRoute(N_{emb}, v, u, \alpha)$;
7:     $E^C \leftarrow E^C \cup \{v, u\}$;
8:     $g(v, u) \leftarrow m$;
9:     $E^S \leftarrow E^S \cup E^T$;
10:   **end for**
11: **end for**
12: $E^T \leftarrow BestDCCQTree(\{\emptyset, N^C\}, v_{pre}, \alpha)$;
13: $V^T \leftarrow Q$;
14: **for all** $(v, u) \in E^T$ **do**
15:   **if** $(v, u) \notin E$ **then**
16:     replaceByBestDCCQRoute((v, u), E^S, E^T, V^T);
17:   **end if**
18: **end for**
19: $E^T \leftarrow BestDCCQTree(\{N^T_{emb}, \emptyset\}, v_{pre}, \alpha)$;
20: **for all** $v \in V^T \backslash Q$ and isLeaf(v) **do**
21:   removeLeafAndBranch(v, E^T, V^T);
22: **end for**
23: **return** N^T_{emb};

Algorithm 5.5 $BestDCCQTree(\{N_{emb}, N^C\}, v_s, \alpha)$

Input: Either embedded network $N_{emb} = (V, E, c, w, p)$ or a cost network $N^C = (V, E^C, g)$, start node v_s, a metric weighting factor α

Output: Best device class and connection quality tree E^T

1: $Q \leftarrow V$; $m \leftarrow 0$; $E^C \leftarrow \emptyset$;
2: Let s the successor node vector; init with $s[v] \leftarrow -1$ for all $v \in V$
3: Let d the $DCCQ$ vector; init with $d[v_s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V \backslash v_s$
4: **while** $Q \neq \emptyset$ **do**
5:    Let be u the node where $\forall v \in Q$ is $d[u] \leq d[v]$
6:    **for all** $v \in Q$ and $w(u, v) > 0$ or $g(u, v) > 0$ **do**
7:       **if** $N_{emb} \neq \emptyset$ **then**
8:          $m \leftarrow \alpha \cdot (c(v) + c(u) + 2 - (p(v) + p(u))) + \frac{(1-\alpha)}{w(u,v)}$;
9:       **else**
10:          $m \leftarrow g(v, u)$;
11:       **end if**
12:       **if** $d[v] > m$ **then**
13:          $d[v] \leftarrow m$;
14:          $s[v] \leftarrow u$;
15:       **end if**
16:    **end for**
17:    $Q \leftarrow Q \backslash u$;
18: **end while**
19: $E^T \leftarrow constructTree(s)$;
20: **return** E^T;

5.3.3 Post-Filter Placement

In this part in our service data dissemination approach, we are trying to find suitable and processable nodes in N^T_{emb} that enables us to a shared service data delivery from v_{pre} to all or to some particular service subscribers as long as possible. In order to realize this, we will place one or more post-filters that evaluate the relevance for same tree branches and provide node destinations for either the client nodes or the next post-filter nodes. Explaining this in more detail, let us look at an example: Let us assume that we have an N^T_{emb} given as depicted in Figure 5.4 that was constructed by the $DisseminationTree$ algorithm. Furthermore, based on the data model given in Figure 3.4 the service subscriber queries are

- Node 4: $Q_1 = //Humidity$

  $Q_2 = //status[text() =' LowBattery']$
- Node 7: $Q_3 = //Temperature/value$
- Node 8: $Q_4 = //Temperature/value[text() < 21.0]$
- Node 9: $Q_5 = //Temperature/value[text() > 20.5]$.

v_{pre} (node 1) already runs pre-filter G_F that evaluates whether queries are of relevance or not. If all client queries are relevant (message contains status value 'LowBattery' and temperature value 20.7), it is desired that the data be sent only once from v_{pre} to all client nodes. This should also be the case if only a subset of queries match (e.g., the message contains the status value 'OK' and the temperature value of 20.0). For realizing a shared delivery we will place post-filters at node 3 (G'_F) as well as node 6 (G''_F). More precisely, node 3 has multi successor branches, one of which leads to client node 4 and the other to client nodes 7, 8, and 9. At this point, we are no longer able to share the service data for all clients. Consequently, we have to use a post-filter to decide along which branch we are going to forward the service data. If queries Q_1 and/or Q_2 match, then the service data will be forwarded to node 4, and/or if one of the queries Q_3, Q_4, or Q_5 match, then the data will be forwarded on along the branch that leads to client nodes 7, 8, and 9. So as to forward the service data in the latter scenario only once, node 6 will provide the next post-filter. However, it only contains evaluation constructions of queries Q_3, Q_4, or Q_5 since node 4 can no longer be reached at this point. Based on the received data and data content, node 6 will forward the message to the corresponding client nodes as final step. Summing up, the pre-filter and the post-filter contain the following routing information:

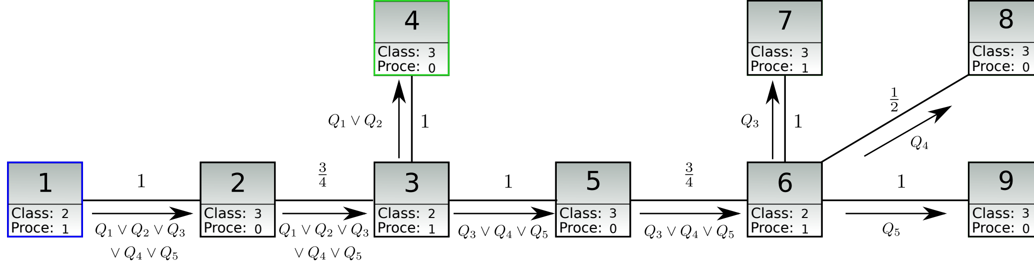


Figure 5.4: N_{emb}^T that spans v_{pre} (node 1) and service subscribers (nodes 4, 7, 8, and 9). An optimized service data dissemination is realized by placing post-filters at node 3 and 6.

- G_F (at node 1): forwards service data to node 3 when at least one of the queries Q_1, Q_2, Q_3, Q_4, or Q_5 matches.
- G'_F (at node 3): forwards service data to node 4 when Q_1 or Q_2 match and/or to node 6 when Q_3, Q_4, or Q_5 match.
- G''_F (at node 6): forwards service data to node 7 (for any cases), to node 8 when Q_4 matches, and/or to node 9 when Q_5 matches.

As the examples shows the best position for post-filters are nodes that have multiple successor branches, powerful device classes, and are processable to run a filter mechanism. However, if processability is missing, for example, then we have to find alternative nodes nearby to keep the shared data property. The *PostFilterPlacement* algorithm (see Algorithm 5.6) take sall this aspects into account and tries to find the best places for post-filters in N_{emb}^T to enable a shared service data delivery.

As input it takes a tree network N_{emb}^T, the client's queries Q, the determined pre-filter node v_{pre} with its filter grammar G_F, and the underlying data model XSD. The outcome is a tuple set F that contains the dedicated filter nodes with the corresponding filter grammar. The algorithm starts to determine all nodes that have multi successor branches in the tree (line 2). Next, we start to consider each branch node in B separately. Thereby, in each round, we select the node in B that is closest (in terms of hop-count) to the root node (the pre-filter node v_{pre}) in the tree (line 5). One of the most important aspects for placing a post-filter mechanism is the processability of a selected branch node. Doing this, it is first and foremost determined which kind of queries can be addressed from v (line 7). For instances, consider the example above (Figure 5.4): Node 3 would address all client queries (Q_1,..,Q_5) and hence, the function *reachableClientQueries* would return these queries. However, node 6 will only address queries Q_3, Q_4, and

Q_5. Thus, *reachableClientQueries* would return only these queries. In general, it always will be $R \subseteq Q$. Based on R, we will construct a filter grammar G'_F (line 8). If, after investigation, it turns out that G'_F cannot be deployed because of leak of resources (line 9), we will try to find another processable node which is on the path to the root (lines 10-12). After finding a v we will test if this node has not yet been registered for a post-filter placement. If so, we will continue to select another branch node (lines 14-15) and check it for processability.

In the case of a suitable v, the algorithm will continue to collect this corresponding node with its filter grammar G'_F in F and will register v in P as a processed node (lines 18-19).

The last major step in this algorithm is determining the routing information that has to be assigned to the filter mechanism. At this point, we also consider pre-filter node v_{pre} with its filter grammar G_F (line 21). For each tuple in F, we determine the post-filter occurring first with the successor branches of v (lines 22-23). The nodes determined in N will be associated with the filter grammar G_F, identified by the underlying queries Q (lines 24-25). In other words, each query that is represented in G'_F will be assigned the delivery destination information to which the service data has to be forwarded next. In cases where there is no next post-filter node ($N = \emptyset$), the queries in G'_F will forward the service data to the final client destinations.

Algorithm 5.6 $PostFilterPlacement(N^T_{emb}, Q, v_{pre}, G_F, XSD)$

Input: Embedded network $N^T_{emb} = (V^T, E^T, c, w, p)$ which has a tree characteristics, a set Q of client's queries, the dedicated pre-filter node v_{pre} with its filter grammar G_F, and the data model as XML schema definition XSD.

Output: A set F that consist of a tuple of dedicated post-filter nodes with its filter grammars.

1: $P \leftarrow \emptyset$; $R \leftarrow \emptyset$; $G'_F \leftarrow \emptyset$; $F \leftarrow \emptyset$;
2: $B \leftarrow nodesWithMultiSuccesorBranches(E^T)$;
3: $B \leftarrow B \backslash v_{pre}$;
4: **while** $B \neq \emptyset$ **do**
5: $\quad v \leftarrow closestNodeToRoot(B, v_{pre})$;
6: $\quad B \leftarrow B \backslash v$;
7: $\quad R \leftarrow reachableClientQueries(v, Q)$;
8: $\quad G'_F \leftarrow FilterGrammar(XSD, R)$;
9: $\quad$ **if** $isProcessable(v, G'_F) == false$ **then**
10: $\quad\quad v \leftarrow getAncestor(v)$;
11: $\quad\quad$ **while** $isProcessable(v, G'_F) \neq true$ and $isRoot(v) \neq true$ **do**
12: $\quad\quad\quad v \leftarrow getAncestor(v)$;
13: $\quad\quad$ **end while**
14: $\quad\quad$ **if** $v \in P$ **then**
15: $\quad\quad\quad$ continue;
16: $\quad\quad$ **end if**
17: $\quad$ **end if**
18: $\quad F \leftarrow F \cup \{v, G'_F\}$;
19: $\quad P \leftarrow P \cup v$;
20: **end while**
21: $F \leftarrow \{v_{pre}, G_F\} \cup F$;
22: **for all** $\{v, G'_F\} \in F$ **do**
23: $\quad N \leftarrow nextPostFiltersInBranches(v)$;
24: $\quad$ **if** $N \neq \emptyset$ **then**
25: $\quad\quad associateQueries(G'_F, Q, N)$;
26: $\quad$ **end if**
27: **end for**
28: **return** F;

5.4 Example

We will now consider the embedded network $N_{emb} = (V, E, c, w, p)$ which is shown in Figure 5.1(a). Node 1 is a service provider based on the data model presented in Figure 3.4. Nodes 3, 4, and 7 are the service requesters with the following conditions:

- Node 3: $Q_1 = //Humidity$

  $Q_2 = //status[text() =' OK']$
- Node 4: $Q_3 = //Temperature/value[text() < 21.0]$
- Node 7: $Q_4 = //Temperature/value[text() > 20.5]$.

Figure 5.5 shows the filter grammar G_F based on this query after applying the *FilterGrammar* algorithm (see Algorithm 4.8) that was presented in chapter 4.4. Since the service requester node does not provide us with the opportunity to set up an filter mechanism for clients' subscription requests, we have to find an alternative node for placing a pre-filter mechanism. In order to do so, we have to identify any processable nodes within the network that have enough resources to run G_F. We use the variable r_{cur} with $r_{cur} \in [0, 100]$ to represent the current resources of each node in the network. 0 means no resources and 100 means that full resources are available. Table 5.1 shows the nodes that are processable in Figure 5.1(a) as well as their current resources capability.

Node v	r_{cur}	p(v)	$p(v, 50)$
3	30	1	0
4	45	1	0
5	70	1	1
8	55	1	1

Table 5.1: Ressources capability of nodes in Figure 5.1(a)

In our example we will apply the following processability test function (comparable to the *isProcessable* procedure applied in Algorithm 5.2 and 5.6) for each node

$$p(v, r_{req}) = \begin{cases} max\{0, \lfloor 2 - \frac{r_{req}}{r_{cur}} \rfloor\} & , r_{cur} > 0 \\ 0 & , r_{cur} = 0 \end{cases} \quad . \tag{5.7}$$

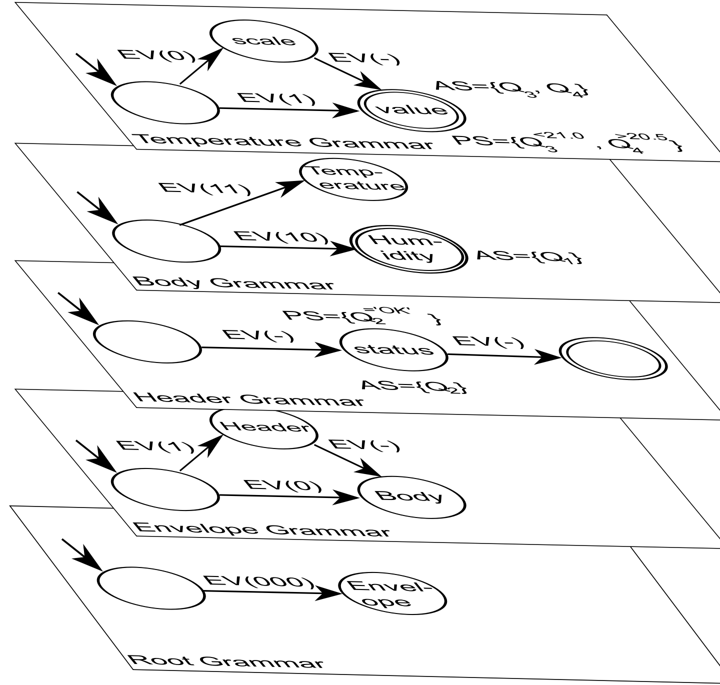


Figure 5.5: Pre-filter grammar

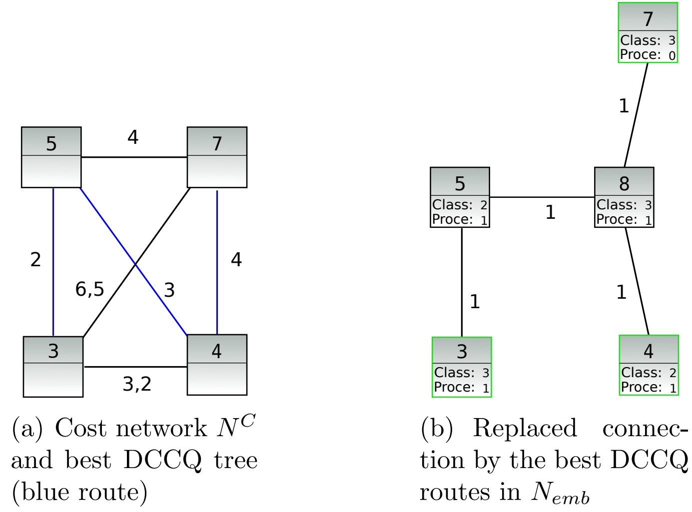
Please note that Function 5.7 can also be adjusted individually for each node class, however, for the sake of simplicity we will not take this into account here. Let us assume that G_F has a resource load of 50. Applying this to p for each current processable node in N_{emb} produces the outcome that is shown in the last column of Table 5.1.

Thus, the *ClosestPreFilterNode* (see Algorithm 5.2) will test two candidates: nodes 5 and 8. Starting with service node 1, it will determine the costs $f_{1,5}$ and $f_{1,8}$. Each cost will be multiplied by the sum of cost to the client nodes 3, 4, and 7. The result is as follows

- $f_{1,5} \cdot (f_{5,3} + f_{5,4} + f_{5,7}) = 2,5 \cdot (2 + 3 + 4) = 22,5$
- $f_{1,8} \cdot (f_{8,3} + f_{8,4} + f_{8,7}) = 4 \cdot (3,5 + 2 + 3) = 34$

Consequently, the outcome of algorithm *ClosestPreFilterNode* shows that node 5 is the best pre-filter node.

The next step involves determining a dissemination tree that spans node 5 and client nodes 3, 4, and 7. In order to do so, we will call the *Dissemination-Tree* (see Algorithm 5.4). There, the first step includes constructing a cost network N^C based on these nodes. Figure 5.6(a) shows the outcome. Next, we will determine the best cost tree in this complete connected network by



(a) Cost network N^C and best DCCQ tree (blue route)

(b) Replaced connection by the best DCCQ routes in N_{emb}

Figure 5.6: *DisseminationTree* algorithm

using the subroutine *BestDCCQTree*. The blue route marked within Figure 5.6(a) shows the result. All connections of this tree that do not exist in the origin network N_{emb} will be replaced with the best DCCQ route. E.g., the tree provides a direct connection from node 4 to node 7; this, however, did not exist in N_{emb}. The best route from 4 to 7 in N_{emb} is via node 8. Thus, this route will be used and it replaces the direct connection from 4 to 7 in the tree. The replacement process results in a subnetwork of N_{emb} which does not contain any cycles (see Figure 5.6(b)). At this point, the *DisseminationTree* algorithm has no further impact on the network seen in Figure 5.6(b) due to two aspects: First of all, the subnetwork already has the desired tree characteristics and applying subroutine *BestDCCQTree* again would lead to the same result. Secondly, all leaves involve those nodes which were desired to be spanned. Thus, we don't have to remove any leaves and branches.

The last major processing step in our dissemination algorithm involves determining suitable post-filter nodes to enable a high ratio of shared service data from service provider to service requesters. Starting with (root) node 5, the *PostFilterPlacement* (see Algorithm 5.6) will first select all nodes that contain multi successor branches. Nodes 5 and 8 are candidates. Since node 5 already is a dedicated pre-filter node, we will not consider it further and instead check node 8 directly for processability of a post-filter grammar. The post-filter grammar is constructed based on the queries that can be reached from node 8. This is true for the queries Q_3 and Q_4. Figure 5.7 shows the post-filter as based on these queries. If we assume that this post-filter

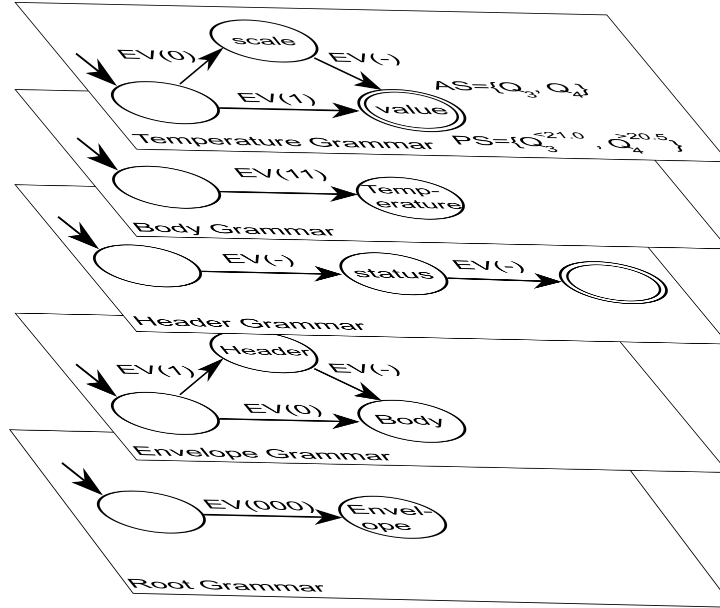


Figure 5.7: Post-filter grammar

request has a resource capability of 30, then our processability Function 5.7 above would return $p(8, 30) = 1$ based on current resource capabilities given in Table 5.1. Hence, node 8 would be selected as a post-filter node. Before the *PostFilterPlacement* algorithm is terminated, we are going to update the network's defined filters in terms of routing information. Node 5 is set up with the pre-filter G_F that is shown in Figure 5.5. There it contains the associated information Q_1 and Q_2 relates to node 3, Q_3 relates to node 4, and query Q_4 to node 7. Based on the post-filter to be placed on node 8, service data that matches queries Q_3 and Q_4 shall be forwarded to node 8, which then will send the data only once. Thus, G_F is updated with this information. In summary, we obtain the following routing information:

- G_F (at node 5): forwards service data to node 3 when queries Q_1 and/or Q_2 match; forwards service data to node 8 when queries Q_3 and/or Q_4 match.

- G'_F (at node 8): forwards service data to node 4 when Q_3 matches; forwards service data to node 7 when Q_4 matches

Finally, we have achieved service data dissemination as shown in Figure 5.1(d). All new service data is sent to node 5 for evaluation. Relevant service data is forwarded to client node 3 and/or to the next (post-) filter

node 8. Depending on its relevance, node 8 forwards this data to client nodes 4 and/or 7.

5.5 Extensions and Optimizations

The dissemination approach introduced in this chapter provides a number of opportunities for extension and optimization which we will be discuss below.

Cost Penalties and Path Sharing Awards: Our selected cost function is mainly based on the metrics of device classes with processability and connection quality. A weighting factor (α) in our cost function f can be used to adjust the focus on node property (device class and processing ability) or on connection quality. An exponential weighting or penalty factor applied to the device classes or connection quality value would additionally *punish* very constrained device classes or bad connection qualities.

A high number of processable nodes in a determined dissemination tree indirectly results in a high ratio of shared connections of service data messages. It is also possible to take this aspect into account for f. In this way, a dissemination tree is rewarded when it consists of a high number of shared connections.

Complexity The complexity of our dissemination approach is mainly dominated by the finding a suitable pre-filter node. Let us assume that in an given embedded network N_{emb} each node, except for the node of service provider v_s, is processable and has enough resource capability to run a pre-filter grammar. In addition, each node in N_{emb} is a service requester of node v_s. In our approach we would consider $|V|-1$ nodes and would determine the best DCCQ route to all other nodes. More precisely, we end up with a complexity of $\mathcal{O}((|V|-1)(|V|log|V|+|E|)) = \mathcal{O}(|V^2|log|V|+|V||E|)$. So as to avoid such a scenario we can select a pre-defined value k that restrict the observation number of processable nodes. A variant is to consider each branch of v_s and its nodes until the branch depth k is reached. All nodes beyond k are not considered anymore.

Complexity can also be improved if the intermediary DCCQ result of the *ClosestPreFilterNode* is taken to the *DisseminationTree* algorithm. This is especially true for constructing the cost network, where we can fall back to this intermediary result and do not have to determine it again.

Direct Delivery Our service data dissemination approach tries to evaluate as soon as possible whether service data is relevant (for the clients in the

embedded network) at the pre-filter node. In the case of a multi query match of different client nodes we attempt to share service data for as long as possible by the usage of post-filters. However, if there are only disjunct queries or if there is no overlap in value constraints in the pre- and post-filters, service data can be delivered directly without being forwarded to the next post-filter.

5.6 Evaluation

So as to organize service data dissemination of each new applied application and to estimate its influence in terms of traffic and device capacity usage of real embedded networks we wrote an embedded network simulator. The simulator provide us with the opportunity to load particular network topologies and characteristics as well as service provider and the service subscribers with its queries. Another alternative is to setup randomized embedded networks by providing different kinds of generation parameters: number of nodes, number of different kind of device classes, and the ratio of device classes and connection quality. Based on such a network, we are able to set up new applications by selecting particular nodes, which operate a service with the provided service description, and the client nodes that subscribe service data with the predefined conditions on the service data. We can then run our dissemination algorithm for each new installed application.

In order to test the effectiveness of the approach presented in this chapter, we randomly generated two kinds of embedded networks. The first network has a complexity of 50 nodes with three device classes, the second one consists of 100 nodes with four device classes. An overview of the different network setups with their different ratios can be found in Table 5.2. Both network setups are initially feature a balanced ratio of processable and non-processable nodes. The first scenario comes with three different device classes. Their ratio consists of 5 times device classes 1, 10 times device classes 2, and 35 times device classes 3. The second scenario uses four classes with a ratio of 10 times device classes 1, 10 times device classes 2, 20 times device classes 3, and 60 times device classes 4. Initially, for both network scenarios, we uniformly distributed the connection quality weighting values with numbers between 0.8 and 1.

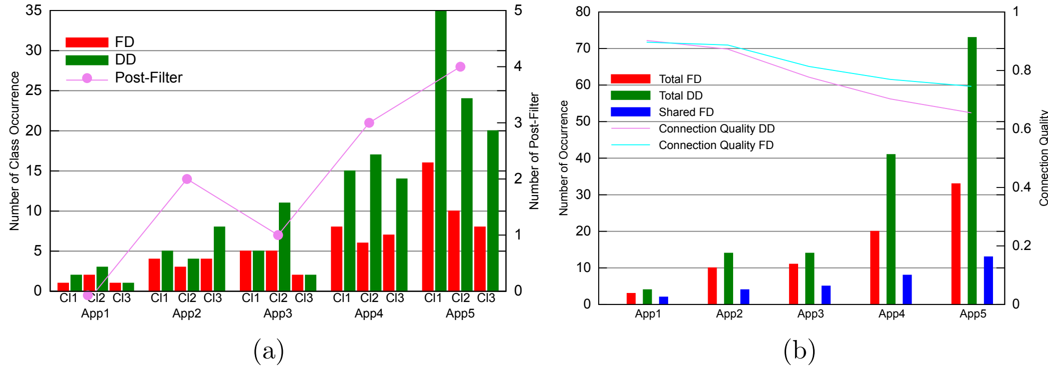
For each network we sequentially installed five different kinds of applications. In general, an application is based on a service provider and different kind of service requester (the clients). The distance (in terms of hop count) and client distribution to the service provider node is increased with each new

$\|V\|$	Processable	#Classes	Ratio	App/#Clients
50	25:25	3	5:10:35	1/2, 2/3, 3/4, 4/5, 5/6
100	50:50	4	10:10:20:60	1/4, 2/6, 3/8, 4/10, 5/12

Table 5.2: Network configurations for evaluation

installed application. For the first network, we start with the first application, which has two clients; subsequently, the second has 3 clients, the third has 4 clients, there are 5 clients in the fourth application, and finally the fifth application has 6 different service requesters. The second network scenario with $|V| = 100$, we double the number of clients for each application, which can be seen in the last column of Table 5.2. For each installed application we evaluated the service data dissemination for two variants: *Filter-enabled dissemination* (abbreviated with *FD*) represents our filter-enabled dissemination approach and the separate and *direct dissemination* (abbreviated with *DD*) reflects the direct, non-filtered service data delivery which, however, takes into account the device class and connection quality. In both cases, we always consider the worst case scenario for dissemination delivery. In other words, each generated service data matches at least one query of the clients and hence, the data has to be delivered to each registered client in this application. For each determined dissemination path variant of an application (both for FD and DD) we decremented the weighting connection quality by the value of 0,1. This simply reflects the additional connection load in terms of, among other factors, delay or bandwidth. In the context of FD, we set each dedicated filter (pre- and post-filter) node along the path to be non-processable so as to simulate the impact of the additional filter mechanisms in the network.

Figure 5.8 shows the evaluation result for network scenario $|V| = 50$. Figure 5.8(a) depicts the result for each application in terms of device class occurrences (Cl1=Class 1 nodes, Cl2=Class 2 nodes, and Cl3=Class 3 nodes) in the dissemination path of our approach (FD Optimized) as compared to the simple approach, wherein each service data is delivered separately (DD Simple). In other words, we count the occurrence of the device classes in the determined dissemination path (tree) that reflects the worst case scenario when a service message is relevant for all service requesters in the network. As can be seen in for all cases, our approach, as presented in this thesis, results in a lower usage of class occurrences as compared to the simple service data distribution variant. This becomes especially apparent the more complex the application is. Furthermore, the occurrences also show that our determined dissemination paths always consist of the desirable, relatively small number



(a)

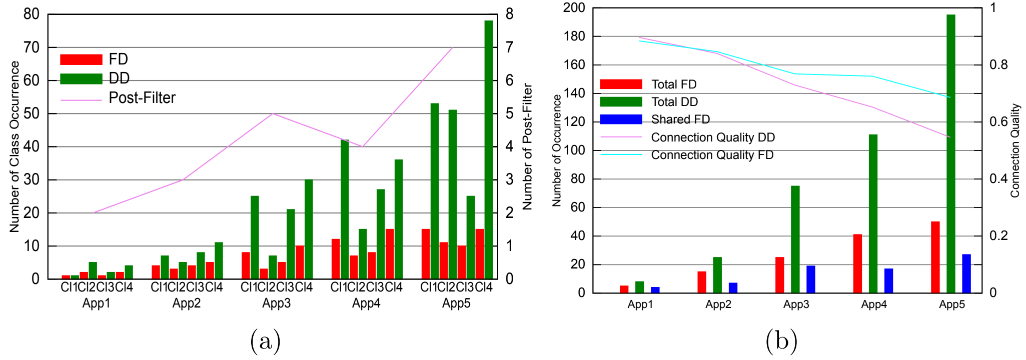
(b)

Figure 5.8: Embedded network with $|V| = 50$: (a) Count of used classes in the dissemination path and number of used post-filters for each application. (b) Number of links of the dissemination path and average value of connection quality.

of constrained nodes (class 3). For instance, in a worst case distribution scenario for application five, our dissemination approach uses the device class 1 sixteen times, class 2 ten times, and the most constrained device class 3 eight times. In total, 34 nodes are involved in the dissemination process. In contrast, a simple dissemination would lead to a device class ratio of class 1 thirty-five times, class 2 twenty-four times, and class 3 twenty times. In total, this involves 79 nodes. Consequently, our approach results in a better resource usage of the nodes in the embedded networks since less total nodes are involved in the dissemination tree; the number of constrained nodes (class 3) is kept as small as possible.

The evaluation results in Figure 5.8(a) also shows the number of used post-filter nodes in the application. Consequently, the more complex the application the number of post-filter rises. E.g., four post-filters are used in application 5.

Figure 5.8(b) show the evaluation result in terms of the number of connection links used and average connection quality. As can be seen, the number of connections used in a dissemination process is smaller for our approach as compared to the simple variant. The graphic also shows the ratio of the shared connections of the optimized variant in each application. We determined the number based on whether each connection between two nodes can reach a pre-filter or a post-filter node. If so, the number of shared connections is incremented (Shared FD). The presented numbers shows the effectiveness of our approach, since for each application we determine a dissemination tree that consist of a high ratio of shared connections. Figure 5.8(b) also shows



(a) (b)

Figure 5.9: Embedded network with $|V| = 100$

the average connection quality for each application and its dissemination based on both our approach and the simple variant. As can be observed, the simple dissemination variant loses the average connection quality faster than our approach. This is explained by the fact that the simple variant involves a lot more connection links and causes potential more network traffic. The more applications are installed in the network, the greater the impact on connection quality will be.

The Figure 5.9 confirms our previous findings; however, now the network scenario consists of 100 nodes, 4 device classes, and a double ratio of service requesters. Subfigure 5.9(a) shows the almost equal distribution of device classes in each application even if there is a high ratio of constrained class 4 devices (60%). In contrast, many outliers can be seen in the simple variant.

Subfigure 5.9(b) shows the connection ratio and the average connection quality. Due to an increased number of service requesters, the number of used routes increases in the simple variant. Consequently, the average connection quality has a higher impact than that of our optimized dissemination.

5.7 Related Work

To the best of our knowledge, we are the first to propose a filter-enabled dissemination mechanism based on binary XML with EXI and for constrained embedded networks. Basically, the concept of data sharing is not a new topic and investigations have especially focused on the domain of Data Stream Management Systems (DSMS). The same is true for the content-based routing in networks. In the following subsections, we will provide related works that focus on these two topics.

5.7.1 Data Stream Management Systems

Finding a suitable pre-filter node outside of the service data origin node and the position of post-filters in a dissemination tree opens the opportunity to share relevant service data with a number of service subscribers. This leads to a reduction of resources used within embedded network in terms of network traffic as well as processing overhead. Similar topics are addressed by and can be found in Data Stream Management Systems (DSMS). Data Stream Management Systems (DSMS) complement the traditional Database Management Systems (DBMSs) [Kemper and Eickler, 2011]. Typically, a DBMS handles persistent and random accessible data and executes volatile queries. Meanwhile, in DSMS persistent queries are executed over volatile and sequential data. Examples of DSMSs includes *Aurora* [Abadi et al., 2003], *Borealis* [Abadi et al., 2005], *TelegraphCQ* [Chandrasekaran et al., 2003], and *StreamGlobe* [Kuntschke et al., 2005]. The main focus of such systems is on the efficient processing of potentially infinite data streams against a set of continuous queries. In contrast to publish/subscribe systems such as XFilter [Altinel and Franklin, 2000], YFilter [Diao and Franklin, 2003], or our highly efficient binary XML filtering mechanisms introduced in Chapter 4, continuous queries in DSMSs can be far more complex than simple filter subscriptions. Some researches develops new query languages such as WindowedXQuery (WXQuery) [Kuntschke, 2008] to extend query operations. In the domain of constrained embedded networks, however, we presume the presence of relatively simple data models and have found that XPath expressions are sufficient to address data interests and simple constraints by predicates. Other important topics in distributed DSMSs such as StreamGlobe and Borealis revolve around network-aware stream processing and operator placement. These are also issues relevant to constrained embedded networks and, similarly, we took them into account for our approach by positioning the pre-filter and, if possible, post-filter mechanism at the embedded nodes.

Most DSMSs, such as TelegraphCQ for example, are based on relational data. StreamGlobe, however, focuses on plain-text XML data streams as well as on XML-based query languages such as XQuery [Boag et al., 2007] or the above mentioned WXQuery. Consequently, nodes used for distributed data stream processing in systems such as StreamGlobe and Borealis generally need to be far more powerful than the microcontrollers for constrained embedded devices that we aim for in this thesis. E.g., our reference hardware, such as the ARM Cortex M3 microcontroller with 256kB RAM, 16kB ROM, and 24MHz, would be overcharged by the complex XML parsing libraries as well as the query engine to evaluate data relevance. Our approach of realizing efficient binary XML (see Chapter 3) and constructing of high performance

filter mechanisms (see Chapter 4) enables us to bring DSMS topics to the domain of constrained embedded networks.

5.7.2 Content-based Network Routing

In our approach, filter nodes such as pre-filter or post-filter ones decide how to best forward service messages if there are one or more matches. The destinations may include service requester nodes and/or other post-filter nodes. In the literature, this is called content-based routing or application-level routing since routing depends on constellation of data within a message. In that context, we can refer to works such as the *combined broadcast and content-based* (*CBCB*) routing scheme [Carzaniga and Wolf, 2002], the *application layer multicast algorithm* (*ALMA*) [Ge et al., 2006], the usage of *XML Router* [Snoeren et al., 2001], and *view selection for stream processing* based on XML data [Gupta et al., 2003]. Below, we will concentrate on the last mentioned related works since they also involve XML-based data content.

The XML Router approach [Snoeren et al., 2001] creates an overlay network that is implemented by multi XML routers. An XML router is a node that receives XML packets and forwards a subset of these XML packets. The XML packets are forwarded to other routers or the final client node destinations. Thereby, the output links represent the XPath queries that describe the portion of the router's XML stream that should be sent to the host on that connection link. XML routers are comparable to our pre- and post-filter concept. However, additional strategies, such as reassembling a data packet stream from diverse senders provided by the diversity control protocol (DCP) or the banking on plain-text XML and XPath interpreters, are not feasible in a resource constrained embedded environment.

The view selection for stream processing method is an interesting approach followed in [Gupta et al., 2003, Gupta et al., 2002]. The main concept includes selecting a set of XPath expressions which are called *views*. The service data producers evaluate the views and add the result to the data package in the form of a (binary) header. The advantage is that servers which keep a local set of queries can evaluate their workload by inspecting only the values in the header and do not need to parse the XML document. This leads to a speed-up of routing decisions. However, this is only true for cases in which the evaluation in the header is positive. Otherwise, the complete (plain-text) XML document has to be parsed and the query has to be evaluated in a normal way. Again, this is an obstacle in the constrained embedded environment. In addition, one of our goal is to achieve seamless protocol usage and to work with standardized message representations to support interoperability in a heterogeneous network environment. Adding a

header to a message would break this principle and necessitate an adjustment of communication protocols.

5.8 Summary

In this chapter we presented an approach to realize efficient service data dissemination in the context of known service data demands of the service subscribers. Finding a suitable pre-filter node in an embedded network leads to an early evaluation of relevant service messages. By using post-filters in a determined dissemination tree we are able to avoid redundant transmissions and share the service data, especially if there is a multi-query match of different kinds of service requesters. The effectiveness in terms of device classes occurrences, connection quality, and number of shared connections was demonstrated in a simulated environment based on our embedded network simulator that compared our approach to a non-shared and non-filter dissemination variant. Our approach reduces both network traffic and computational load of embedded nodes. In general, this also leads to less energy usage within the network which is an important resource boundary of WSNs.

Chapter 6

Project Experiences

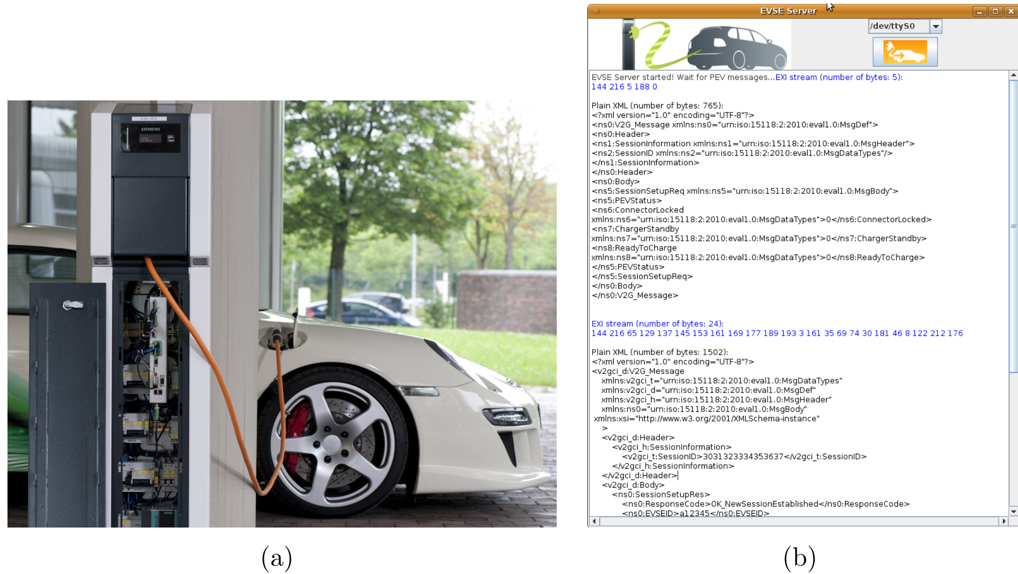
6.1 Introduction

In the following chapter we are going to show how our developed approaches can be beneficial to realize cross-domain applications in an embedded environment consisting of resource-constrained hardware. As an example how a Web service-based adoption can be used in a real-world scenario, we refer to the field of electro mobility. In that context we will share our experiences from the participation in standardization activities in ISO/IEC 15118 and from the cooperation with our project partners such as electricity suppliers and original equipment manufacturers (OEMs).

6.2 Vehicle-to-Grid Standardization

The evaluation section in Chapter 3 already introduced V2G messaging based on ISO/ICE 15118 standardization (see Section 3.4.1). The major goal of this standard is to define a charging protocol valid worldwide. Vehicles from any vendor and from any country shall be able to charge their batteries at any arbitrary charging station of any utility and energy provider. Users of electric vehicles shall have the same user experience as provided by today's conventional refueling at the gas station.

To reach this goal, there has to be agreement on all different kinds of communication levels, starting with the physical layer up to the application layer. Based on the electric cable connectors between the Electrical Vehicle (EV) and Electric Vehicle Supply Equipment (EVSE), power line communication (PLC) is obvious. The actual application communication units are called Electrical Vehicle Communication Controller (EVCC) for the EV side and Supply Equipment Communication Controller (SECC) for the EVSE side



(a) (b)

Figure 6.1: (a) EVSE, SECC (vertical white box within the EVSE), and EV; (b) Charging message trace in binary XML (highlighted in blue) with EXI and in plain-text XML.

(see Figure 6.1(a)). Between these units and via PLC media, different kinds of information have to be exchanged between an EV and an EVSE. This includes which kind of charging mode is desired (e.g., AC or DC), physical values (e.g., max/min voltage and max/min current), charging schedules, tariff tables, and value-added services (VAS) such as traffic information or remote access via a smartphone to the vehicle [Käbisch et al., 2010b]. Furthermore, security aspects also have to be considered since confidential data such as contract IDs and tariff information are exchanged under certain circumstances. The decision for selecting an XML-based Web service approach can be considered by different kind of aspects:

Heterogeneous infrastructure A suitable approach shall be independent of any used hardware and used software platforms which may be specific to the particular vehicle or supply equipment manufacturers. Furthermore, different kind of V2G stakeholders such as energy retailers and clearing houses shall be able to participate in a charging process. As we already know and have discussed in this thesis, a Web service-based approach, which uses hardware and platform independent standards, perfectly meets the criteria for this.

Modelling XML schema provides a powerful mechanism for modeling XML data in a very precise manner in terms of content structuring and type restrictions (see also Section 2). Based on the usage of microcontroller units and the constrained amount or memory on both the EV and EVSE side, each data item has to be declared as it is actually used. Commercial and open-source professional XML Schema tools support the development of the data model by supplying a graphic interface and validating correctness. This eases the development process and enables faster identification of mismodelling.

Encoding Messaging in plain-text XML puts high demands on hardware resources; these, however, do not meet the needs of the embedded environment of the charging domain. Due to issues of time constraint in DC-based charging (20ms for a roundtrip message via PLC) and the usage of constrained embedded devices on both sides (EV and EVSE), an efficient message encoding method has to be selected to facilitate interaction. Based on our research and demonstrations, we were able to show that EXI perfectly meets these requirements. Thus, parallel to the V2G standard and in terms of proof of concept we published project OpenV2G[1] which provided a Web service-based reference implementation of ISO/IEC 15118 based on our developed approach in this thesis. OpenV2G is world wide used in products (e.g., Siemens EVSEs), in laboratory environments and in projects of different vehicle OEMs and energy providers. Latter ones include, e.g., our project contribution to *Electromobility Model Region Munich*[2] with partners such as BMW Group and the municipal utility Stadtwerke München (SWM). OpenV2G could be successfully applied to fast DC charging on a SECC for a Siemens EVSE and on an EVCC for a BMW ActiveE EV.

Security Web services and their extensions such as the ones WS-* (see Section 2.5.2) provide a rich number of standardized protocols for all different kinds of applications. The ISO/IEC 15118 profited from that by utilizing the XML Signature framework [Eastlake et al., 2002], which has been standardized by the W3C. Instead of defining a unique mechanism for signing confidential data content, we were able to simply embed the XML Signature framework. In this context, we initiated to have EXI Canonical [Käbisch and Peintner, 2013] standardized by the W3C so as to implement normative requirements for having a unique representation of the different EXI modes (see Section 3.2.2) for any application (e.g., for signature purposes) that requires

[1]http://openv2g.sourceforge.net/

[2]http://www.siemens.com/press/pool/de/events/corporate/2010-10-ecartec/eCarTec2010-Factsheet-MODELLGREGION-MUENCHEN_e.pdf

the canonical form of the data.

Debugging Due to the binary XML representation of V2G messages, a direct debugging, e.g., on the *wire*, is not possible, since the highly compact EXI format is designed to be machine readable and not human readable. However, based on the XML Infoset requirements, EXI can easily be transformed into a human readable format such as a plain-text XML representation. Figure 6.1(b) shows a snippet of our trace tool that shows the EXI representation on the one hand and features the plain-text representation below. In conclusion, applications profit from the highly efficient machine-readable binary XML format, and on the other hand, developers always have the opportunity to have the high structure plain-text XML as a representation available to them for debugging or other monitoring reasons.

There are further benefits, such as the flexibility and extendibility of XML-based messaging. This enables transmission of additional information (e.g., vendor or costumer specific), which, however, does not need to be domain specific standardized. Thus, vehicle vendors or energy providers are able to support extra features and market them to costumers.

6.3 Complex Charging Infrastructure

Charging standards such as ISO/IEC 15118 are mainly focused on the communication protocol between an EV and an EVSE. The infrastructure is getting more complex when congeries of EVSEs are considered, which may be found in parking areas. Figure 6.2 shows a sample infrastructure and its possible interaction variants based on 4 fixed charging stations. It is not necessary to place the charging control unit, the SECC, within the EVSE. In terms of cost reduction, the unit can be outsourced and the EVSE node only has routing functionality. Figure 6.2 illustrates this case by having two SECCs. Depending on the current resource load of the SECCs, the EVSE can select one of the SECCs for a new charging session. The SECCs can also communicate with a backend server, maintained by an energy provider, for example, to obtain current local grid load information. Another case involves the SECC communicating with a VAS that is triggered by an EV. A VAS can be a vehicle specific application such as a remote control (e.g., by smartphone) for setting up the air conditioning in advance.

The scenario shows just how complex the interaction can become when a high number of heterogeneous instances, such as vehicles, charging stations, and service providers, are taken into consideration. The SOA approach meets

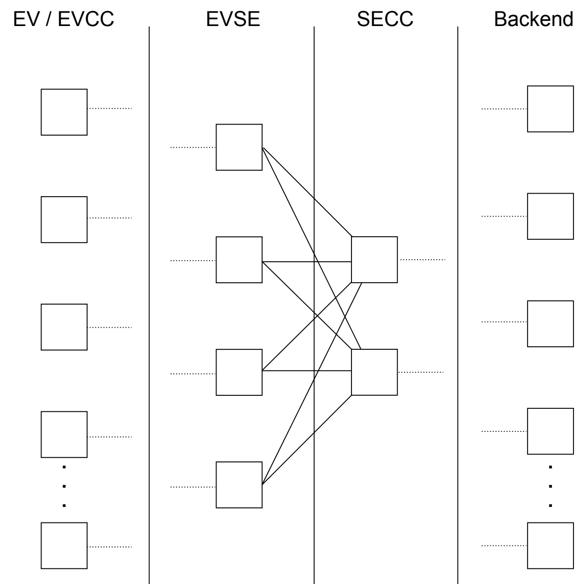


Figure 6.2: Different instances of a standardized charging infrastructure based on 4 EVSEs: Arbitrary EVs are able to connect and communicate (by the EVCC) with one of the EVSE (represented as dotted lines). The communication unit of the EVSE, the SECC, is outsourced. The EVSE routes the messages to one of the SECCs (depending on resource load). Depending on the charging session and costumer demands, the SECCs are able to communicate with arbitrary backend services (represented as dotted lines).

these conditions perfectly. Filter approaches as presented in the previous chapters in this thesis can be used to identify, e.g., irregular values such as exciting power values. For example, this can be checked by the EVSE nodes. In these types of situations, emergency messages can be sent to all SECCs commanding them to interrupt current charging sessions.

6.4 Related Work

ISO/IEC 15118 is not the only standardization related to vehicle-to-grid or smart grid, respectively. DIN 70121 [DIN, 2012] is a subset of the DIS status of ISO/IEC 15118 that mainly addresses DC charging and skips VAS as well as security topics. It is assumed that ISO/IEC 15118 will replace this standard once it is finally released.

The Smart Energy Profile 2.0 (SEP 2.0) [ZigBee, 2013] from the ZigBee alliance is a standard running parallel to ISO/IEC 15118, which mainly addresses smart grid applications such as smart metering. Like ISO/IEC 15118,

SEP 2.0 uses XML-based technologies. Originally, SEP 2.0 was supposed to have a unique charging protocol for EV; however, this has been deferred and most likely the ISO/IEC 15118 will be used instead.

6.5 Conclusion

Using XML-based technologies such as Web services provides many benefits in terms of development, maintenance, and debugging of applications for embedded networks. In this chapter we discussed the positive usage of Web service-based applications and our associated developed approaches in a real-world scenario, the V2G domain. Based on our initiative such as proof of concept demonstrator or the OpenV2G project, we were able to show that V2G service interacting between EV and EVSE using XML-based messages are highly efficient and feasible to constrained devices such as microcontrollers. This supported the decision to make XML-based service interaction with EXI encoding mandatory in the ISO/IEC 15118 standard.

Chapter 7

Conclusion and Outlook

The Service-oriented Architecture (SOA) concepts such as independence and reusability of *services* perfectly meets the unique characteristics of embedded networks that may consist of a high number of heterogeneous and very constrained embedded devices, when it comes to the development of, e.g., cross-domain applications. Standardized Web services - the popular implementation that would perfectly follow the SOA paradigm - provide communication technologies to enable interaction between distributed and heterogeneous instances in a very dedicated manner. However, these technologies are highly demanding of hardware resources and are therefore not scalable to embedded networks, which may consist of small microcontrollers that only have a few kBytes of memory, restricted processing capabilities, and very limited communication bandwidth.

This thesis has presented innovative approaches for efficiently realizing standardized Web services and interaction with them in the domain of embedded networks, which takes constrained microcontrollers into account also. XML Infoset describes in an abstract manner how XML-based data content shall be structured in a normative way. Typically, Web services mainly use plain-text XML; this, however, is not feasible for constrained embedded devices. We are pursuing a novel approach by using binary XML with the Efficient XML Interchange (EXI) format. EXI fulfills XML Infoset requirements and is based on a regular grammar philosophy used to convey XML-based content into binary form (encoding) and vice versa (decoding). This way, we do not consider the transformation from or to plain-text XML (into or from binary XML), instead we operate directly within the binary representation to gather the relevant information.

Our Web service generator developed in this thesis takes a service description, such as the well-known WSDL, and creates a source code that is also scalable to constrained embedded devices. Through a detailed analysis,

we determined the relevant information required for both the client and Web service side contexts. We used our findings to optimize the EXI grammar for each context and to adjust the data binding and service message interpreter such as the dispatcher. We proved applicability in terms of memory, processing speed, and message size for bandwidth usage by taking a real world scenario from the electro mobility domain. Based on these findings, we were able to show - despite a relatively complex Web service serving different kinds of RPCs with over 100 data elements being exchanged - that XML-based Web services are feasible for the embedded domain, even if very constrained devices such as microcontrollers are used.

A high number of service requesters with different kinds of service data demands would increase service data interaction within constrained embedded networks. Keeping limited bandwidth and the constrained embedded devices in mind, an efficient interaction is desirable that only delivers messages to the service requester if relevant information is present in the message. This thesis presented different approaches for constructing filter mechanisms for efficient service data dissemination. XPath expressions can be used to evaluate the relevance of binary XML streams. As a baseline, we introduced the *BasicEXIFiltering* approach that runs on top of a given EXI grammar. In contrast, the *OptimizedEXIFilter* operates directly within the grammar and prunes it. The outcome is a filter grammar that represents all given XPath expressions. This enables very fast evaluation with a very low resource and processing overhead on microcontrollers.

Furthermore, we investigated a possible approach for organizing and improving service data dissemination by placing so-called pre- and post-filters in constrained embedded networks. A suitable pre-filter node leads to an early evaluation of the relevance of service data by at least one service requester. By using post-filters in a determined dissemination tree we were able to avoid redundant transmissions and share service data. Based on our cost model that takes into account device class, link quality, and processability, we approximated the best node position for the pre- and post filter to find a dissemination tree from service provider to service requesters. In a simulated environment, we were able to demonstrate that our approach would improve service data dissemination by avoiding very constrained device classes and weak links as well as enabling branch sharing to avoid redundant submissions.

There are other very interesting topics that would extend this work. The most important ones are as follows:

- At the time of writing, the IETF was developing a very promising protocol specialized for usage in constrained embedded networks: CoAP (see Section 2.5.3). An interesting investigation would involve extend-

ing our Web service generator to generate efficient Web services based on CoAP in combination with binary XML with EXI as a payload format.

- Our binary XML filtering approaches, which were introduced in Chapter 4 may be extended to support additional XPath facets such as the node-set functions (e.g., *count()* and *name()*) and math operations (e.g., *div* and *mod*).
- Another focus could include research into dynamic filtering changes at runtime when it comes to modifying query requests. Realizing this in our *OptimizedEXIFilter* approach, which removes non-required grammar fragments for evaluation, would be challenging.
- A similar investigation could involve modifying the dissemination tree for service data distribution at runtime. This could be useful when encountering changes in the topology of constrained embedded networks, which would cut dissemination branches or improve the cost function.

In conclusion, the approaches presented in this thesis provide a fundamental concept for realizing standardized Web service technologies for the constrained embedded domain. In a real world scenario, we were able to prove that V2G service implementation is applicable to constrained devices such as microcontrollers, which resulted in the agreement to make XML-based service interaction with EXI encoding obligatory in the ISO/IEC 15118 standard. Thus our approaches have contributed to the future of electromobility.

Appendix A

Web Service Description

Full Web service description as presented in Section 2.3.4.

Listing A.1: WSDL document of the temperature/humidity Web service

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:edws="embedded:device:ws"
    xmlns:edd="embedded:device:data"
    targetNamespace="embedded:device:ws">

    <!-- ************************************************ -->
    <!-- @author: Sebastian Kaebisch -->
    <!-- Embedded Device Web Service -->
    <!-- *************************************************-->


    <types>
        <xs:schema targetNamespace="embedded:device:data" xmlns="
            embedded:device:data"
            elementFormDefault="unqualified" attributeFormDefault="
                unqualified">

            <!-- getTemperature request definition -->
            <xs:element name="GetTemperature" type="GetType"/>

            <!-- Temperature response definition -->
            <xs:element name="Temperature" type="TemperatureType"/>

            <!-- getHumidity request definition -->
            <xs:element name="GetHumidity" type ="GetType"/>
```

```
<!-- Humidity response definition -->
<xs:element name="Humidity" type ="HumidityType"/>

<!-- generic device information -->
<xs:element name="status" type="statusType"/>

<!-- Complex Types -->
<xs:complexType name="GetType">
    <xs:sequence>
        <xs:element name="subscribe" type="xs:boolean"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="TemperatureType">
    <xs:sequence>
        <xs:element name="value" type="xs:float"/>
    </xs:sequence>
    <xs:attribute name="scale" type="tempScaleType"/>
</xs:complexType>

<xs:complexType name="HumidityType">
    <xs:sequence>
        <xs:element name="value" type="xs:byte"/>
    </xs:sequence>
    <xs:attribute name="scale" type="humScaleType" use="
        required"/>
</xs:complexType>

<!-- Simple Types -->
<xs:simpleType name="tempScaleType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Celsius"/>
        <xs:enumeration value="Fahrenheit"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="humScaleType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Absolute"/>
        <xs:enumeration value="Relative"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="statusType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Standby"/>
        <xs:enumeration value="OK"/>
        <xs:enumeration value="Error"/>
```

```
            </xs:restriction>
        </xs:simpleType>
    </xs:schema>
</types>

<!-- Header definations -->
<message name="SOAPHeader">
    <part name="header" element="edd:status"/>
</message>

 <!-- Body definitions -->
<!-- getTemperature message -->
<message name="GetTemperatureMsg">
    <part name="parameters" element="edd:GetTemperature"/>
</message>

<!-- Temperature message -->
<message name="TemperatureMsg">
    <part name="parameters" element="edd:Temperature"/>
</message>

<!-- getHumidity message -->
<message name="GetHumidityMsg">
    <part name="parameters" element="edd:GetHumidity"/>
</message>

<!-- Humidity message -->
<message name="HumidityMsg">
    <part name="parameters" element="edd:Humidity"/>
</message>

<!-- Port defination for request/response -->
<portType name="embeddedDeviceReqResInterface">

    <!-- Access Operation -->
    <operation name="Temperature">
        <input message="edws:GetTemperatureMsg"/>
        <output message="edws:TemperatureMsg"/>
    </operation>

    <!-- Humidity Operation -->
    <operation name="Humidity">
        <input message="edws:GetHumidityMsg"/>
        <output message="edws:HumidityMsg"/>
    </operation>
</portType>

<!-- Port defination for eventing -->
<portType name="embeddedDeviceEventingInterface">
```

```
    <!-- Temperature Operation -->
    <operation name="Temperature">
        <output message="edws:TemperatureMsg"/>
    </operation>

    <!-- Humidity Operation -->
    <operation name="Humidity">
        <output message="edws:HumidityMsg"/>
    </operation>
</portType>

<binding name="embeddedDeviceWSReqRes" type="edws:
    embeddedDeviceReqResInterface">
    <soap:binding style="document" transport="http://schemas.xmlsoap
        .org/soap/http"/>

    <!-- Temperature operation -->
    <operation name="Temperature">
        <!-- <soap:operation soapAction="http://localhost:20012/
            EVSEAccessInterface/#Access"/> -->
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:header message="edws:SOAPHeader" part="header" use=
                "literal"/>
            <soap:body use="literal"/>
        </output>
    </operation>


    <!-- Humidity operation -->
    <operation name="Humidity">
       <!-- <soap:operation soapAction="http://localhost:20012/
           EVSEAccessInterface/#Status"/> -->
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:header message="edws:SOAPHeader" part="header" use=
                "literal"/>
            <soap:body use="literal"/>
        </output>
    </operation>

</binding>
```

```xml
    <binding name="embeddedDeviceWSEventing" type="edws:
        embeddedDeviceEventingInterface">
        <soap:binding style="document" transport="http://schemas.xmlsoap
            .org/soap/http"/>

        <!-- Temperature operation -->
        <operation name="Temperature">
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>

        <!-- Humidity operation -->
        <operation name="Humidity">
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>

    <!-- Service location -->
    <service name="EmbeddedDeviceWS">
        <port name="EmbeddedDeviceReqRes" binding="edws:
            embeddedDeviceWSReqRes">
            <soap:address location="http://localhost:20012/
                embeddedDeviceWS"/>
        </port>

        <port name="EmbeddedDeviceEventing" binding="edws:
            embeddedDeviceWSEventing">
            <soap:address location="http://localhost:20013/
                embeddedDeviceWS"/>
        </port>
    </service>
</definitions>
```

Appendix B

XPath Queries

Used queries for the experimental evaluation with microcontrollers in Section 4.5.3.

Scenario 1:

- $/V2G_Message/Header/Notification/FaultCode$
- $//ServiceDiscoveryReq/ServiceCategory[text() =' EVCharging']$

Scenario 2:

- Queries of scenario 1
- $/V2G_Message/*/ChargeParameterDiscoveryReq/*...$
  $.../EVMaximumCurrentLimit/Value[text() > 90]$
- $//ChargeParameterDiscoveryReq/DC_EVChargeParameter...$
  $.../FullSOC[text() > 100]$

Scenario 3:

- Queries of scenario 2
- $V2G_Message/Body/PowerDeliveryReq/*/*...$
  $.../EVErrorCode[text() =' FAILED_EVRESSMalfunction']$
- $V2G_Message/Body/PowerDeliveryReq/DC_EVPowerDeliveryParameter...$
  $.../ChargingComplete[text() = "true"]$
- $//CurrentDemandReq/EVTargetVoltage/Value[text() > 450]$
- $V2G_Message/Body/CurrentDemandReq...$
  $.../EVTargetCurrent/Value[text() > 110]$

Bibliography

[DOM, 1998] (1998). *Document Object Model DOM Level 2 Events Specification*. World Wide Web Consortium.

[BiM, 2005] (2005). *Information technology - MPEG systems technologies - Part 1: Binary MPEG format for XML*. ISO copyright office.

[Abadi et al., 2005] Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. B. (2005). The design of the borealis stream processing engine. In *CIDR*, pages 277–289.

[Abadi et al., 2003] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. B. (2003). Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139.

[Altinel and Franklin, 2000] Altinel, M. and Franklin, M. J. (2000). Efficient filtering of xml documents for selective dissemination of information. In Abbadi, A. E., Brodie, M. L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., and Whang, K.-Y., editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 53–64. Morgan Kaufmann.

[Amundson et al., 2006] Amundson, I., Kushwaha, M., Koutsoukos, X., Neema, S., and Sztipanovits, J. (2006). Efficient integration of web services in ambient-aware sensor network applications. In *Broadband Communications, Networks and Systems, 2006. BROADNETS 2006. 3rd International Conference on*, pages 1–8.

[Apache, 2010] Apache (2010). Apache Xerces. http://xerces.apache.org/.

[Avaro and Salembier, 2001] Avaro, O. and Salembier, P. (2001). MPEG-7 systems: overview. *IEEE Trans. Circuits Syst. Video Techn*, 11(6):760–764.

[Banavar et al., 1999] Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R., and Sturman, D. (1999). An efficient multicast protocol for content-based publish-subscribe systems. In *19th International Conference on Distributed Computing Systems (19th ICDCS'99)*, Austin, Texas. IEEE.

[Bartel et al., 2008] Bartel, M., Boyer, J. M., Fox, B., LaMacchia, B., and Simon, E. (2008). XML signature syntax and processing (second edition). World Wide Web Consortium, Recommendation REC-xmldsig-core-20080610.

[Benson, 2004] Benson, R. (2004). Streaming API for XML. Java Specification Request (JSR) 173.

[Boag et al., 2007] Boag, S., Chamberlin, D. D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J. (2007). XQuery 1.0: An XML query language. World Wide Web Consortium, Recommendation REC-xquery-20070123.

[Bormann and Mulligan, 2009] Bormann, C. and Mulligan, G. (2009). Ipv6 over low power wpan (6lowpan). http://datatracker.ietf.org/wg/6lowpan/charter/.

[Bournez, 2009a] Bournez, C. (2009a). Efficient XML Interchange Evaluation. http://www.w3.org/TR/exi-evaluation/. W3C Working Draft 7 April 2009.

[Bournez, 2009b] Bournez, C. (2009b). Efficient XML interchange evaluation. W3C working draft, W3C. http://www.w3.org/TR/2009/WD-exi-evaluation-20090407.

[Boyer, 2007] Boyer, J. M. (2007). XForms 1.0 (third edition). World Wide Web Consortium, Recommendation REC-xforms-20071029.

[Bray et al., 2008] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible markup language (XML) 1.0 (fifth edition). World Wide Web Consortium, Recommendation REC-xml-20081126.

[Brown and Hamilton, 2006] Brown, P. F. and Hamilton, R. M. B. A. (2006). Reference model for service oriented architecture 1.0.

[Carzaniga et al., 2001] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2001). Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383.

[Carzaniga and Wolf, 2002] Carzaniga, A. and Wolf, A. L. (2002). Content-based networking: A new communication infrastructure. *Lecture Notes in Computer Science*, 2538:59–??

[Chan et al., 2002] Chan, C.-Y., Felber, P., Garofalakis, M., and Rastogi, R. (2002). Efficient filtering of XML documents with XPath expressions. *VLDB Journal: Very Large Data Bases*, 11(4):354–379.

[Chandrasekaran et al., 2003] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. (2003). Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*.

[Chen and Wong, 2004] Chen, D. and Wong, R. K. (2004). Optimizing the lazy DFA approach for XML stream processing. In Schewe, K.-D. and Williams, H. E., editors, *Fifteenth Australasian Database Conference (ADC2004)*, volume 27 of *CRPIT*, pages 131–140, Dunedin, New Zealand. ACS.

[Chinnici et al., 2007] Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. (2007). Web services description language (WSDL) version 2.0 part 1: Core language. World Wide Web Consortium, Recommendation REC-wsdl20-20070626.

[Christensen et al., 2001] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[Chumbley et al., 2010] Chumbley, R., Durand, J., Pilz, G., and Rutt, T. (2010). Web Services Interoperability (WS-I) Basic Profile Version 2.0. urlhttp://ws-i.org/profiles/BasicProfile-2.0-2010-11-09.html.

[Clark and DeRose, 1999] Clark, J. and DeRose, S. (1999). XML path language (XPath) version 1.0. w3c recommendation 16 november 1999. internet.

[Cormen et al., 2009] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2009). *Introduction to Algorithms*. McGraw-Hill Higher Education, 3rd edition.

[Cowan and Tobin, 2004] Cowan, J. and Tobin, R. (2004). XML Information Set (Second Edition). http://www.w3.org/TR/xml-infoset/. W3C Recommendation 4 February 2004.

[Crockford, 2006] Crockford, D. (2006). The application/json media type for JavaScript Object Notation (JSON). Internet Request for Comment RFC 4627, Internet Engineering Task Force.

[Dargie and Poellabauer, 2010] Dargie, W. and Poellabauer, C. (2010). *Fundamentals of wireless sensor networks: theory and practice*. Wiley. com.

[Davis et al., 2011] Davis, D., Malhotra, A., Warr, K., and Chou, W. (2011). Web services metadata exchange (WS-metadataexchange). World Wide Web Consortium, Recommendation REC-ws-metadata-exchange-20111213.

[De Souza et al., 2008] De Souza, L. M. S., Spiess, P., Guinard, D., Köhler, M., Karnouskos, S., and Savio, D. (2008). Socrades: a web service based shop floor integration infrastructure. In *Proceedings of the 1st international conference on The internet of things*, IOT'08, pages 50–67, Berlin, Heidelberg. Springer-Verlag.

[Deutsch, 1996] Deutsch, L. P. (1996). DEFLATE compressed data format specification version 1.3. Internet RFC 1951.

[Diao et al., 2003] Diao, Y., Altinel, M., Franklin, M. J., Zhang, H., and Fischer, P. M. (2003). Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst*, 28(4):467–516.

[Diao and Franklin, 2003] Diao, Y. and Franklin, M. J. (2003). High-performance xml filtering: An overview of yfilter. *IEEE Data Eng. Bull.*, 26(1):41–48.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

[DIN, 2012] DIN (2012). DIN SPEC 70121. http://www.din.de.

[Driscoll and Mensch, 2009] Driscoll, D. and Mensch, A. (2009). Devices Profile for Web Services Version 1.1. Technical report.

[Dunkels et al., 2004] Dunkels, A., Grönvall, B., and Voigt, T. (2004). Contiki - A lightweight and flexible operating system for tiny networked sensors. In *LCN*, pages 455–462. IEEE Computer Society.

[Eastlake et al., 2002] Eastlake, D. E., Reagle, J. M., and Solo, D. (2002). XML-signature syntax and processing. World Wide Web Consortium, Recommendation REC-xmldsig-core-20020212.

[ECMA, 2006] ECMA (2006). *ECMA-376: Office Open XML File Formats.* ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA:adr.

[Ferraiolo et al., 2003] Ferraiolo, J., Fujisawa, J., and Jackson, D. (2003). Scalable vector graphics (SVG) 1.1 specification. World Wide Web Consortium, Recommendation REC-SVG11-20030114.

[Fialli and Vajjhala, 2005] Fialli, J. and Vajjhala, S. (2005). Java architecture for XML binding (JAXB) 2.0. Java Specification Request (JSR) 222.

[Fielding, 2000] Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of Califormia, Irvine, USA.

[Fielding et al., 1997] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and Berners-Lee, T. (1997). Hypertext transfer protocol – HTTP/1.1. RFC 2068, Internet Engineering Task Force.

[Fielding et al., 1999] Fielding, R. T., Gettys, J., Mogul, J., Nielsen, H. F., Masinter, L., Leach, P. J., and Berners-Lee, T. (1999). Hypertext Transfer Protocol — HTTP/1.1. Internet Request for Comment RFC 2616, Internet Engineering Task Force.

[Freed and Borenstein, 1996] Freed, N. and Borenstein, N. S. (1996). Multipurpose internet mail extensions (MIME) — part one: Format of internet message bodies. Internet RFC 2045.

[Ge et al., 2006] Ge, M., Krishnamurthy, S. V., and Faloutsos, M. (2006). Application versus network layer multicasting in ad hoc networks: the ALMA routing protocol. *Ad Hoc Networks*, 4(2):283–300.

[Goldman and Lenkov, 2005] Goldman, O. and Lenkov, D. (2005). XML binary characterization. World Wide Web Consortium, Note NOTE-xbc-characterization-20050331.

[Green et al., 2004] Green, T. J., Gupta, A., Miklau, G., Onizuka, M., and Suciu, D. (2004). Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst*, 29(4):752–788.

[Green et al., 2003] Green, T. J., Miklau, G., Onizuka, M., and Suciu, D. (2003). Processing XML streams with deterministic automata. In Calvanese, D., Lenzerini, M., and Motwani, R., editors, *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, pages 173–189. Springer.

[Grosso et al., 2003] Grosso, P., Maler, E., Marsh, J., and Walsh, N. (2003). XPointer framework. World Wide Web Consortium, Recommendation REC-xptr-framework-20030325.

[Grust and Teubner, 2002] Grust, T. and Teubner, J. (2002). XPath evaluation with SAX, DTDs. http://www.inf.uni-konstanz.de/dbis/teaching/ss02/xml/.

[Gudgin et al., 2003] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., and Frystyk Nielsen, H. (2003). SOAP version 1.2 part 1: Messaging framework. World Wide Web Consortium, Recommendation REC-soap12-part1-20030624.

[Gudgin et al., 2007] Gudgin, M., Hadley, M., Rogers, T., and Yalçınalp, Ü. (2007). Web services addressing 1.0 — metadata. World Wide Web Consortium, Recommendation REC-ws-addr-metadata-20070904.

[Gupta et al., 2002] Gupta, A. K., Halevy, A. Y., and Suciu, D. (2002). View selection for stream processing. In *WebDB*, pages 83–88.

[Gupta et al., 2003] Gupta, K., Suciu, and Halevy (2003). The view selection problem for XML content based routing. In *PODS: 22th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.

[Haas and Brown, 2004] Haas, H. and Brown, A. (2004). Web services glossary. W3C note, W3C. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.

[Hoeller et al., 2008] Hoeller, N., Reinke, C., Neumann, J., Groppe, S., Boeckmann, D., and Linnemann, V. (2008). Efficient xml usage within wireless sensor networks. In *Proceedings of the 4th Annual International Conference on Wireless Internet*, WICON '08, pages 74:1–74:10, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Hondo et al., 2007] Hondo, M., Yalçinalp, Ü., Hirsch, F., Orchard, D., Yendluri, P., Boubez, T., and Vedamuthu, A. S. (2007). Web services policy 1.5 - primer. W3C note, W3C. http://www.w3.org/TR/2007/NOTE-ws-policy-primer-20071112.

[Hunter, 2001] Hunter, J. (2001). JDOM 1.0. Java Specification Request (JSR) 102.

[IEC, 2007] IEC (2007). IEC 60929.

[IEEE, 2011] IEEE (2011). IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). http://standards.ieee.org/about/get/802/802.15.html.

[Insights, 2013] Insights, I. (2013). MCU Market on Migration Path to 32-bit and ARM-based Devices. http://www.icinsights.com/data/articles/documents/541.pdf.

[International Telecommunication Union, 2001] International Telecommunication Union (2001). Information technology — ASN.1 encoding rules — XML encoding rules (XER). ITU-T Recommendation X.693.

[International Telecommunication Union, 2002a] International Telecommunication Union (2002a). Information technology — ASN.1 encoding rules — specification of basic encoding rules (BER), canonical encoding rules (CER), and distinguished encoding rules (DER). ITU-T Recommendation X.690.

[International Telecommunication Union, 2002b] International Telecommunication Union (2002b). Information technology — ASN.1 encoding rules — specification of packed encoding rules (PER). ITU-T Recommendation X.691.

[International Telecommunication Union, 2004] International Telecommunication Union (2004). Information technology — ASN.1 encoding rules: Mapping W3C XML schema definitions into ASN.1. ITU-T Recommendation X.694.

[ISO, 2003] ISO (2003). Building automation and control systems – Part 5: Data communication protocol. http://www.iso.org/.

[ISO, 2006] ISO (2006). *ISO/IEC 26300: Open Document Format for Office Applications (OpenDocument) v1.0*. International Organization for Standardization, pub-ISO:adr.

[ISO, 2010] ISO (2010). ISO TC 22/SC 3 JWG 1. http://www.iso.org/iso/iso_technical_committee.html?commid=46752.

[ISO/IEC, 2012] ISO/IEC (2012). Iso/iec dis 15118-2: Road vehicles - vehicle to grid communication interface – part 2: Network and application protocol requirements.

[ITU, 2002] ITU, T. S. S. (2002). Abstract Syntax Notation One (ASN.1) Specification of Basic Notation. ITU-T Rec. X.680.

[ITU, 2005] ITU, T. S. S. (2005). Information technology Generic applications of ASN.1: Fast infoset. ITU-T Rec. X.891.

[Jammes and Smit, 2005] Jammes, F. and Smit, H. (2005). Service-oriented paradigms in industrial automation. *Industrial Informatics, IEEE Transactions on*, 1(1):62–70.

[Käbisch et al., 2012] Käbisch, S., Kuntschke, R., Heuer, J., and Kosch, H. (2012). Efficient Filtering of Binary XML in Resource Restricted Embedded Networks. In *Proc. of the 8th International Conference on Web Information Systems and Technologies (WEBIST 2012)*, pages 174–182.

[Käbisch and Peintner, 2013] Käbisch, S. and Peintner, D. (2013). Canonical EXI. http://www.w3.org/TR/canonical-exi/. W3C Draft.

[Käbisch et al., 2010a] Käbisch, S., Peintner, D., Heuer, J., and Kosch, H. (2010a). Efficient and Flexible XML-based Data-Exchange in Microcontroller-based Sensor Actor Networks. 5th International IEEE SOCNE Workshop on Service Oriented Architectures in Converging Networked Environments.

[Käbisch et al., 2011] Käbisch, S., Peintner, D., Heuer, J., and Kosch, H. (2011). Optimized XML-based Web Service Generation for Service Communication in Restricted Embedded Environments. 16th IEEE International Conference on Emerging Technologies and Factory Automation.

[Käbisch et al., 2010b] Käbisch, S., Schmitt, A., Winter, M., and Heuer, J. (2010b). Interconnections and Communications of Electric Vehicles and Smart Grids. pages 161–166. 1th IEEE International Conference on Smart Grid Communications (IEEE SmartGridComm).

[Karp, 1974] Karp, R. M. (1974). Reducibility among combinatorial problems. In *Complexity of Computer Computations, Miller,Tatcher, Plenum.*

[Kemp and Modi, 2009] Kemp, D. and Modi, V. (2009). *Web Services Dynamic Discovery (WS-Discovery) Version 1.1.* Number OASIS Standard 1 July 2009. Organization for the Advancement of Structured Information Standards.

[Kemper and Eickler, 2011] Kemper, A. and Eickler, A. (2011). *Datenbanksysteme - Eine Einführung, 8. Auflage.* Oldenbourg.

[Knuth et al., 1977] Knuth, Morris, and Pratt (1977). Fast pattern matching in strings. *SICOMP: SIAM Journal on Computing*, 6.

[Kou et al., 1981] Kou, Markowsky, and Berman (1981). A fast algorithm for steiner trees. *ACTAINF: Acta Informatica*, 15.

[Kuntschke et al., 2005] Kuntschke, R., Stegmaier, B., Kemper, A., and Reiser, A. (2005). Streamglobe: Processing and sharing data streams in grid-based p2p infrastructures. In Böhm, K., Jensen, C. S., Haas, L. M., Kersten, M. L., Larson, P.-Å., and Ooi, B. C., editors, *VLDB*, pages 1259–1262. ACM.

[Kuntschke, 2008] Kuntschke, R. B. (2008). *Network-aware optimization in distributed data stream management systems*. PhD thesis, Technical University Munich. urn:nbn:de:bvb:91-diss-20070806-625762-1-3; http://d-nb.info/989053113.

[Kushwaha et al., 2007] Kushwaha, M., Amundson, I., Koutsoukos, X., Neema, S., and Sztipanovits, J. (2007). Oasis: A programming framework for service-oriented sensor networks. In *Communication Systems Software and Middleware, 2007. COMSWARE 2007. 2nd International Conference on*, pages 1–8.

[Lafon and Mitra, 2007] Lafon, Y. and Mitra, N. (2007). SOAP version 1.2 part 0: Primer (second edition). Technical report, W3C. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[Lerche et al., 2011] Lerche, C., Laum, N., Moritz, G., Zeeb, E., Golatowski, F., and Timmermann, D. (2011). Implementing powerful web services for highly resource-constrained devices. In *PerCom Workshops*, pages 332–335. IEEE.

[Liang et al., 2010] Liang, J.-J., Yuan, Z.-W., Lei, J.-J., and Kwon, G.-I. (2010). Reliable routing algorithm on wireless sensor network. In *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on*, volume 1, pages 47–51.

[Lublinsky, 2007] Lublinsky, B. (2007). Defining soa as an architectural style. Technical report, IBM. http://www.ibm.com/developerworks/architecture/library/ar-soastyle.

[Madden et al., 2005] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2005). Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173.

[Malhotra et al., 2009] Malhotra, A., Warr, K., Davis, D., and Chou, W. (2009). Web services eventing (WS-eventing). W3C working draft, W3C. http://www.w3.org/TR/2009/WD-ws-eventing-20091217.

[Megginson, 2000] Megginson, D. (2000). "SAX 2.0: The Simple API for XML". Web page. http://www.megginson.com/SAX/index.html.

[Meinel and Sack, 2003] Meinel, C. and Sack, H. (2003). *WWW: Kommunikation, Internetworking, Web-Technologien*. Springer-Verlag, Berlin, Germany.

[Melzer, 2007] Melzer, I. (2007). *Service-orientierte Architekturen mit Web Services - Konzepte, Standards, Praxis (2. Aufl.)*. Spektrum Akademischer Verlag.

[Montenegro et al., 2007] Montenegro, G., Kushalnagar, N., Hui, J., and Culler, D. (2007). Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard).

[Moreau and Schlimmer, 2003] Moreau, J.-J. and Schlimmer, J. C. (2003). Web services description language (WSDL) version 1.2 part 3: Bindings. World Wide Web Consortium, Working Draft WD-wsdl12-bindings-20030611.

[Moritz et al., 2010] Moritz, G., Timmermann, D., Stoll, R., and Golatowski, F. (2010). encdpws - message encoding of soap web services. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 784–787.

[Moritz et al., 2009] Moritz, G., Zeeb, E., Prter, S., Golatowski, F., Timmermann, D., and Stoll, R. (2009). Devices profile for web services in wireless sensor networks: Adaptations and enhancements. In *Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–8.

[Nezhad et al., 2006] Nezhad, H. R. M., Benatallah, B., Casati, F., and Toumani, F. (2006). Web services interoperability specifications. *IEEE Computer*, 39(5):24–32.

[Niedermeier et al., 2002] Niedermeier, U., Heuer, J., Hutter, A., and Stechele, W. (2002). MPEG-7 binary format for XML data. In *DCC*, page 467. IEEE Computer Society.

[Oracle, 2009] Oracle (2009). Java Micro Edition CLDC 1.1. http://www.oracle.com/technetwork/java/javame/index.html.

[Peintner, 2012] Peintner, D. (2012). EXIficient - W3C EXI Reference Implementation. http://exificient.sourceforge.net/.

[Peintner, 2014] Peintner, D. (2014). *Efficient Exchange and Processing of Semi-structured Data in the Embedded Domain*. PhD thesis.

[Peintner et al., 2009] Peintner, D., Kosch, H., and Heuer, J. (2009). Efficient XML interchange for rich internet applications. In *Multimedia and Expo, 2009. ICME 2009*, pages 149–152.

[Perera et al., 2006] Perera, S., Herath, C., Ekanayake, J., Chinthaka, E., Ranabahu, A., Jayasinghe, D., Weerawarana, S., and Daniels, G. (2006). Axis2, middleware for next generation web services. In *ICWS*, pages 833–840. IEEE Computer Society.

[Postel, 1980] Postel, J. (1980). User datagram protocol. Technical Report RFC 768, DARPA.

[Prim, R. C., 1957] Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell Systems Technology Journal*, 36:1389–1401.

[Prömel and Steger, 2002] Prömel, H. J. and Steger, A. (2002). *The Steiner Tree Problem; a Tour through Graphs, Algorithms, and Complexity*. Vieweg.

[Richardson and Ruby, 2007] Richardson, L. and Ruby, S. (2007). *RESTful Web Services*. O'Reilly.

[Robie et al., 2011] Robie, J., Chamberlin, D., Dyck, M., and Snelson, J. (2011). XML path language (XPath) 3.0. World Wide Web Consortium, Working Draft WD-xpath-30-20111213.

[Rosen et al., 2008] Rosen, M., Lublinsky, B., Smith, K. T., and Balcer, M. J. (2008). *Applied SOA: Service-Oriented Architecture and Design Strategies*. Wiley Publishing.

[Sadoghi et al., 2011] Sadoghi, M., Burcea, I., and Jacobsen, H.-A. (2011). GPX-matcher: a generic boolean predicate-based XPath expression matcher. In Ailamaki, A., Amer-Yahia, S., Patel, J. M., Risch, T., Senellart, P., and Stoyanovich, J., editors, *EDBT*, pages 45–56. ACM.

[Schneider and Kamiya, 2011] Schneider, J. and Kamiya, T. (2011). Efficient XML Interchange (EXI) Format 1.0. http://www.w3.org/TR/exi. W3C Recommendation 10 March 2011.

[Scholz, 2011] Scholz, A. (2011). *Adaptive Data Processing in Embedded Networks*. PhD thesis.

[Scholz et al., 2009] Scholz, A., Gaponova, I., Sommer, S., Kemper, A., Knoll, A., Buckl, C., Heuer, J., and Schmitt, A. (2009). ϵSOA - Service Oriented Architectures adapted for embedded networks. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 599–605.

[Shannon and Weaver, 1949] Shannon, C. E. and Weaver, W. (1949). The mathematical theory of communication. *Scientific American*.

[Shelby et al., 2013] Shelby, Z., Hartke, K., and Bormann, C. (2013). Constrained application protocol (coap). Technical report, IETF. http://datatracker.ietf.org/doc/draft-ietf-core-coap/.

[Silvasti, 2011] Silvasti, P. (2011). *Algorithms for XML Filtering*. PhD thesis.

[Snoeren et al., 2001] Snoeren, A. C., Conley, K., and Gifford, D. K. (2001). Mesh based content routing using XML. In *SOSP*, pages 160–173.

[Sommer et al., 2009] Sommer, S., Buckl, C., and Knoll, A. (2009). Developing service oriented sensor/actuator networks using a tailored middleware. In *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, ITNG '09, pages 1036–1041, Washington, DC, USA. IEEE Computer Society.

[Sutter et al., 2005] Sutter, R. D., Timmerer, C., Hellwagner, H., and de Walle, R. V. (2005). Multimedia metadata processing: A format independent approach. In Hamza, M. H., editor, *EuroIMSA*, pages 343–348. IASTED/ACTA Press.

[TIA/EIA, 1998] TIA/EIA (1998). Electrical characteristics of generators and receivers for use in balanced digital multipoint systems (tia/eia-485). Technical report, TIA/EIA.

[van der Vlist, 2002] van der Vlist, E. (2002). *XML Schema*. O'Reilly & Associates, Sebastopol, California.

[van Engelen, 2004] van Engelen, R. (2004). Code generation techniques for developing light-weight XML Web services for embedded devices. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 854–861. ACM Press.

[van Engelen and Gallivan, 2002] van Engelen, R. and Gallivan, K. (2002). The gSOAP toolkit for web services and peer-to-peer computing networks. In *CCGRID*, pages 128–135. IEEE Computer Society.

[Vedamuthu et al., 2007] Vedamuthu, A. S., Orchard, D., Hirsch, F., Hondo, M., Yendluri, P., Boubez, T., and Yalçınalp, Ü. (2007). Web services policy 1.5 — primer. World Wide Web Consortium, Note NOTE-ws-policy-primer-20071112.

[XML Schema, 2001] XML Schema (2001). XML schema part 1: Structures, W3C recommendation. http://www.w3c.org/TR/xmlschema-1/.

[XPath2.0, 2007] XPath2.0 (2007). *XML Path Language (XPath) 2.0*. W3C.

[XSLT1.0, 1999] XSLT1.0 (1999). *XSL Transformations (XSLT) Version 1.0*. W3C.

[Yao and Gehrke, 2002] Yao, Y. and Gehrke, J. (2002). The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18.

[ZigBee, 2013] ZigBee (2013). Smart Energy Profile 2 (SEP 2). http://www.zigbee.org/.