EXIP: A Framework for Embedded Web Development

RUMEN KYUSAKOV, PABLO PUÑAL PEREIRA, JENS ELIASSON,
and JERKER DELSING, Luleå University of Technology

Developing and deploying Web applications on networked embedded devices is often seen as a way to reduce the development cost and time to market for new target platforms. However, the size of the messages and the processing requirements of today's Web protocols, such as HTTP and XML, are challenging for the most resource-constrained class of devices that could also benefit from Web connectivity.

New Web protocols using binary representations have been proposed for addressing this issue. Constrained Application Protocol (CoAP) reduces the bandwidth and processing requirements compared to HTTP while preserving the core concepts of the Web architecture. Similarly, Efficient XML Interchange (EXI) format has been standardized for reducing the size and processing time for XML structured information. Nevertheless, the adoption of these technologies is lagging behind due to lack of support from Web browsers and current Web development toolkits.

Motivated by these problems, this article presents the design and implementation techniques for the EXIP framework for embedded Web development. The framework consists of a highly efficient EXI processor, a tool for EXI data binding based on templates, and a CoAP/EXI/XHTML Web page engine. A prototype implementation of the EXI processor is herein presented and evaluated. It can be applied to Web browsers or thin server platforms using XHTML and Web services for supporting human-machine interactions in the Internet of Things.

This article contains four major results: (1) theoretical and practical evaluation of the use of binary protocols for embedded Web programming; (2) a novel method for generation of EXI grammars based on XML Schema definitions; (3) an algorithm for grammar concatenation that produces normalized EXI grammars directly, and hence reduces the number of iterations during grammar generation; (4) an algorithm for efficient representation of possible deviations from the XML schema.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Data compaction and compression; H.3.5 [**Information Storage and Retrival**]: Online Information Services—*Web-based services*

General Terms: Performance, Design, Algorithms, Standardization

Additional Key Words and Phrases: Information Exchange, EXI, XML, data formats, CoAP, data processing, XHTML, embedded systems, internet of things, Web of things

ACM Reference Format:
Rumen Kyusakov, Pablo Puñal Pereira, Jens Eliasson, and Jerker Delsing, 2014. EXIP: A framework for embedded Web development. ACM Trans. Web 8, 4, Article 23 (October 2014), 29 pages.
DOI: http://dx.doi.org/10.1145/2665068

1. INTRODUCTION

Web technologies are rapidly expanding to networked embedded devices with studies showing that in 2013 there were more Web-connected gadgets than people in

This work is supported by the EU FP7 Project IMC-AESOP and ARTEMIS Innovation Pilot Project Arrowhead.
Author's addresses: R. Kyusakov, Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Luleå; email: rumen.kyusakov@ltu.se.

1559-1131/2014/10-ART23 $15.00
DOI: http://dx.doi.org/10.1145/2665068

the United States.[1] This process is expected to accelerate due to the increased IPv6 adoption rate and the availability of small-sized, cheap, off-the-shelf hardware that is powerful enough to execute full-featured network stacks. Already now, the number of TCP/IP connected sensor and actuator devices using low-power wireless technologies or even power-line communication is huge. The application areas cover home automation [Kovatsch et al. 2010], energy management, and industrial process monitoring and control [Bumiller et al. 2010].

With the increase in the number of devices, the requirements on their interfaces are also higher. Consumers are demanding "smart" gadgets that are easy and intuitive to deploy, configure, interact with, and integrate with other devices and systems. An example from the home automation domain is a smart thermostat that can communicate with the user's smart phone to display the current temperature in the house along with energy costs as well as control settings. It is becoming more common to equip the traditionally simple sensor and actuator devices with additional diagnostics, logging, and security capabilities. This phenomenon leads to developing more complex embedded applications, which are often required to support Web connectivity for human-machine interfacing. As the code base increases, so are the product cost and time-to-market for new devices. The development and support for different hardware platforms becomes especially challenging, and thus the need for a common development platform based on established and globally adopted standards. The Web development has proved successful in leveraging a set of global standards for unification of the development for front-end tools and applications over a large number of desktop and mobile platforms. In addition, ICT research as argued by Borriello and Want [2000] suggests that embedded computing will also benefit from Web development platforms.

The trade-off between Web and native applications has been a turning point for development strategies in the mobile market. As discussed by Charland and Leroux [2011], Web applications are cheaper to build, deploy, and maintain, but are often lagging behind in performance and user experience when compared to the native apps. This gap is narrowing, thanks to HTML5 and new Web toolkits such as Argos [Gossweiler et al. 2011] which provides direct access to devices' capabilities from JavaScript code. While the app stores made the management of native applications much easier and user-friendly, their main drawback remains: supporting different platforms often requires substantial rebuild of the code base that needs to be kept up-to-date with new versions of the different operating systems. As Charland et al. conclude, one size does not fit all, and there are use cases when it is better to use one or the other approach. While there are a number of differences between the smart phone and the embedded systems segments, it is possible to draw some similarities and list a number of applications where building Web applications is more beneficial even for resource-constrained hosts when compared to developing proprietary solutions. The simplified use case presented in Section 6, which demonstrates a human-machine interface with a sensor platform, provides an example of such application. In this scenario, the user interface is implemented as dynamic Web application based on CoAP/EXI/XHTML and using the EXIP framework.

The approach of using standard binary protocols for enabling Web connectivity for constrained hosts differs from the most common methods described in the literature. The state-of-the-art solutions to the problems of embedded Web development (e.g., memory, network, and processing constraints) can be classified into two groups. The methods in the first group rely on powerful gateway devices that translate the standard Web protocols to some lightweight messaging framework, and vice versa. An example of

[1]According to data from research firm NPD Group.

this approach is the work by [Trifa et al. 2009], which describes a gateway architecture for providing Web connectivity to highly resource-constrained nodes. The methods in the second group focus on implementing efficient and stripped-down version of the standard text-based Web protocols. High-impact research results based on this method are the techniques for implementing an efficient HTTP server for embedded devices presented by Kang et al. [2006] and Duquennoy et al. [2009], as well as the small-footprint XML Web service implementation by [Van Engelen 2004a].

Using text-based protocols that rely on simple character encoding such as ASCII, was important requirement in the early days of distributed computing systems. During that time, the ability to debug the interactions between the systems with one's bare hands was crucial to the acceleration of the adoption of the protocols. Nowadays, practically all text editors and development tools support UTF-8 character encoding. The tools also parse the XML documents before printing them to the screen to support syntax highlighting. Proper tool support opens up new possibilities for efficient representation of the information on the wire. The new binary encoding schemes are transparent for the user - if, in any case, the XML documents are parsed before printing them, then it is better to use faster, binary encoding which is easier to process than text-based representation. However, implementing highly optimized binary coding schemes is much more challenging than processing text-based streams. Even more challenging, is the use of such binary processors on resource-constrained embedded devices where the memory footprint and CPU usage are crucial. As an example, a common way to compress the size of an XML document is by indexing frequently used tags and value items. Instead of encoding each occurrence in the stream, the repeated information items are represented by their index. Using more extensive indexing increases the compression, but also makes the memory footprint required to store the indexed information larger. Providing efficient methods to build and store the indexes is just one example of optimization that is needed for running binary encoding schemes on embedded hosts.

In this work, we present design and implementation strategies for running an Efficient XML Interchange processor on embedded devices for enabling Web connectivity through RESTful interface that is based on Constrained Application Protocol. The RESTful interface can be used for human-machine interactions with Internet of Things hosts as well as for implementing embedded distributed systems based on the Service Oriented Architecture, as discussed by Shelby [2010].

Unlike XML, the EXI specification mandates the use of schema-specific parsing [Chiu and Lu 2004] when the EXI document is encoded with schema knowledge, that is, using schema mode. In order to address all possible use cases, the presented EXI processor supports both the schema and schemaless modes of operation. This is achieved by using dynamic state machine abstraction that can evolve through addition of new states and state transitions. The main benefit of using static state machines, as in the EIGEN [Doi et al. 2012] and libEXI [Castellani et al. 2011] libraries, is the small footprint and hence the ability to implement highly optimized, dedicated EXI processors. In order to efficiently support a static mode of operation - in other words, strict schema processing with no deviations, the EXIP library needs to be configured to strip the code responsible for evolving the state machines. This can be done easily during compile time due to EXIP modular architecture.

One important component of EXI implementations supporting schema-enabled processing is the automatic generation of the state machines based on XML schema language definitions. These definitions are used to construct a set of formal grammars that describe a particular XML language which is then recognized by the generated state machines. EXIP includes an optimized and lightweight grammar generation utility that can be executed efficiently at run time. This allows it to support dynamic XML

schema negotiations even on embedded hosts. The main contributions of this article are the grammar generation algorithms that are the core of the high performance of this utility. To the best of our knowledge, all other EXI implementations use an external library for processing the XML Schema definitions that are used for the grammar extraction. A commonly used external XML Schema library is Apache Xerces. However, its usage for embedded Web development is limited to static compile-time generation of the EXI state machines.

A prominent research work that is based on the approach of compile-time generation of the state machines is presented by Käbisch et al. [2011]. The authors show that the use of EXI for embedded Web service development brings substantial benefits in hardware utilization (network, CPU, RAM and programming memory). Moreover, their work includes the design of a Web service code generator based on Simple Object Access Protocol (SOAP) and the HTTP/EXI/SOAP protocol stack. Promising future research work, as stated by the authors of that study, is to add support for CoAP RESTful Web service interface to the proposed generator. As such, the EXIP framework described herein is extending and further specifying the suggested CoAP RESTful Web service generator.

EXI is not the only possible data format that can meet the requirements of the embedded Web programming, but it has been shown to provide the highest efficiency compared to rival binary XML solutions [White et al. 2007]. Lightweight text formats such as JSON and Comma-separated values (CSV) or binary encoding schemes (ASN.1, BSON, Protocol Buffers, Thrift etc.) are also capable of representing very efficiently structured information. However, the lack of formally defined mapping between these technologies and the XML Information Set [Cowan and Tobin 2004] makes them unable to guarantee interoperability with existing Web technologies and protocols such as XHTML, Scalable Vector Graphics (SVG), Extensible Messaging and Presence Protocol (XMPP), and RSS feeds to name a few.

The initial goal of the EXIP library was only to provide efficient implementation of an EXI processor for embedded systems. Since the initial version of the prototype EXI processor, the EXIP library was used in a number of research projects and prototypes as in Kyusakov et al. [2011a] and Caputo et al. [2012]. Based on the recurring need of higher processing efficiency and Web integration, the scope of the EXIP project has now extended, and new processing algorithms are employed. In addition to the grammar generation algorithms that are part of the EXI processor prototype implementation, this work defines the overall architecture of the EXIP Web development toolkit. The architecture consists of three main modules: the EXI processor library, EXI data binding, and the CoAP/EXI/XHTML Web page engine. Their functionality, required properties, and overall design in the context of embedded Web development are discussed in Section 2. Detailed descriptions of each of these modules and the associated research questions that are investigated are presented in Sections 3, 5, and 6, respectively.

2. BACKGROUND

Optimizing the hardware utilization by the Web protocols is a key requirement for their application on embedded platforms. Very often the connected devices have limited memory (both RAM and programming memory), and use low-cost CPUs. If the device is battery powered, the communication overhead is a main contributor to the power consumption that needs to be carefully modeled in order to guarantee the intended up-time periods. Simulation tools such as PowerTOSSIM [Shnayder et al. 2004] can be employed to highlight areas of the protocol implementations that are mostly responsible for draining the battery. Among the use of radio duty cycling and CPU sleep modes, reducing the number of packets sent and received is another way of cutting the power consumption, especially in wireless applications.

W3C performed an extensive evaluation of the EXI format [Bournez 2009; White et al. 2007] that shows substantial improvements in compactness compared to text encoding as well as other XML binary formats. Additionally, EXI has superior processing performance compared to plain XML. Both the compactness and processing efficiency depend heavily on the structure of the encoded documents and the options used for processing. For example, the use of XML schema information during encoding and decoding can cut the size of small documents more than 50%, as the element and attribute qualified names are encoded as indexes instead of strings. This allows for substantial reduction of the number of packets required for communication of structured information over the network, and thereby minimizes the power consumption. Existing Web technologies that are formally described using XML schema language such as XHTML, for example, can then be efficiently represented for use in embedded applications.

Compaction and processing improvements of CoAP compared to HTTP are also significant, as reported by Kuladinithi et al. [2011]. Moreover, the asynchronous design of the CoAP protocol makes it much more suitable for event-driven interactions. Publish/subscribe protocols are often preferred in embedded systems, as they provide better hardware and network utilization compared to polling schemes that are used by HTTP, for example.

Figure 1 provides an overview of the state of the art of embedded Web development along with a high level architectural view of the use of binary protocols for network communication and information exchange. The suggested components of the architecture are grouped depending on their role: client-side, networking layer, and server-side execution; and their application domain: user tools and applications, technologies/protocols/specifications, and development tools. The goal of this categorization is to show how the work presented in this article relates to the current technologies and applications, and to further motivate the need for this research.

As shown in Figure 1, client-side user applications of the embedded Web include browsers, graphical Web clients (HMI devices), embedded Web services, and proxy devices translating the binary Web protocols to their text-based counterparts. The technologies to implement these client-side user applications are CoAP client, EXI parser, lightweight client-side scripting engine, and EXI/XHTML/CSS rendering engine. The concrete development tools that can be used for implementing these technologies are the EXIP parser library, which is the primary objective of this work, EXI/XHTML to DOM translator, and CoAP libraries such as libcoap [Kuladinithi et al. 2011], Erbium [Kovatsch et al. 2011] and Californium [Kovatsch et al. 2014].

Similarly, the networking layer shows different wired and wireless network stacks and protocols grouped according to the OSI model along with developing tools used for debugging.

The server-side is represented by resource-constrained embedded devices that are conforming to the *thin server architecture* suggested by Kovatsch et al. [2012]. The server technologies include CoAP server, EXI serializer, and EXI/XHTML page engine. The proposed development server-side tools are the EXI data binder and CoAP/EXI/XHTML Web page engine that are described in detail in Sections 5 and 6 of this work. Other server-side tools for embedded Web development are again, the CoAP libraries libcoap, Erbium, and Californium.

2.1. EXI

EXI data format significantly reduces the size of XML when stored on disk or transferred over the network and also speeds up the parsing and serialization. According to White et al. [2007] the compression level varies between 1% of the original size for large and sparse documents with compression and schema options enabled to 95% for

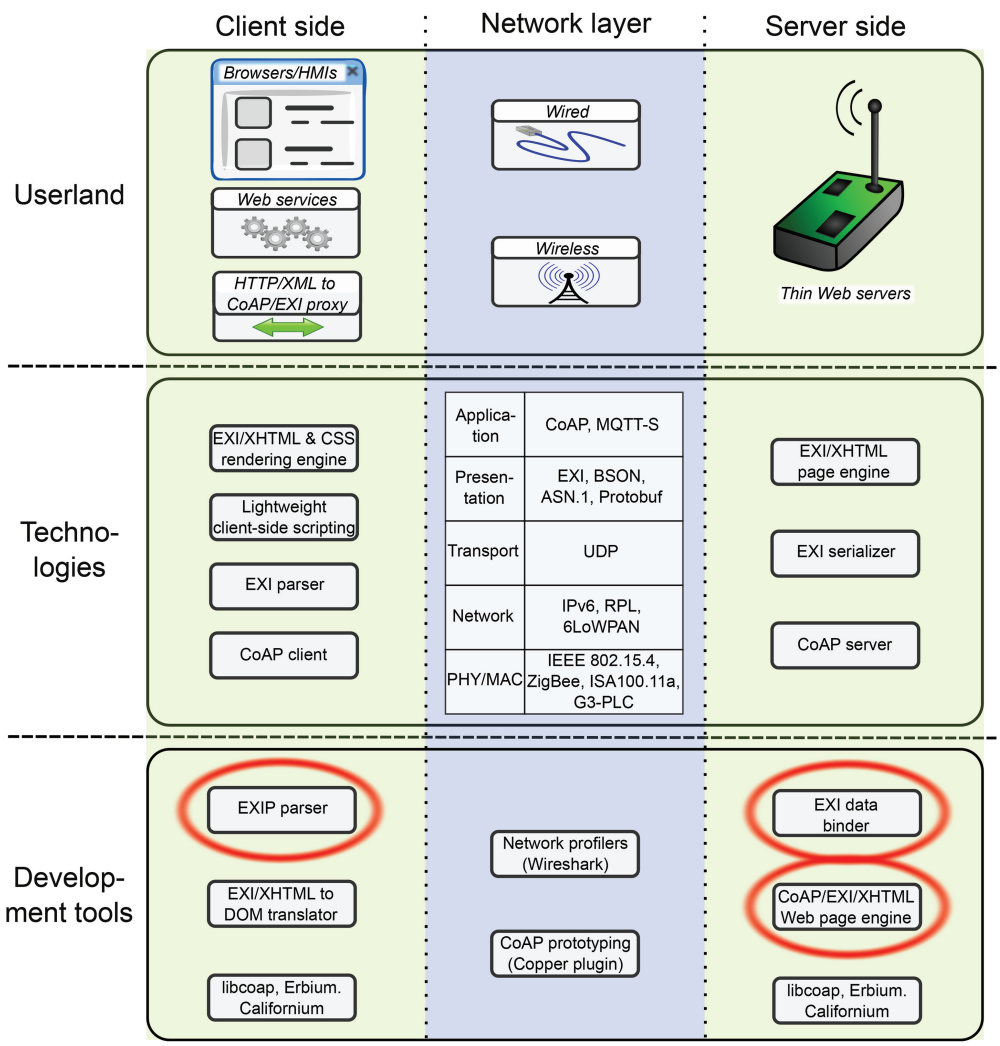

Fig. 1. Overview of the tools and technologies for embedded Web development that are based on standard binary protocols. The tools that are the main focus of this work are marked with red ellipses.

schema-less encoding of very small and dense documents. Nevertheless, EXI format has few drawbacks that are inherited from XML and must be taken into account in the discussions that follow in this article. XML notation and semantics are perceived as complex both for humans to understand but also for machines to process which stems from the design goal of the format to be flexible and easily extendable for application in variety of use cases. This flexibility creates a lot of special cases and exceptions that must be specifically handled with if-then-else statements during serialization and parsing. While EXI is very efficient in removing the redundancy in the XML syntax, it does not simplify the processing - it merely speeds it up. Besides, EXI adds another level of flexibility by introducing encoding options that can be used to influence the level of compression, processing speed and RAM usage during parsing and serialization. Providing support for all possible EXI options requires large and complex code base that can hardly fit into the programming memory of a highly resource constrained

embedded device. Therefore, the application of EXI on such platforms often requires defining a profile of the EXI specification which restricts the supported EXI options to particular values and predefines the XML Schema. Different EXI profiles and how they are supported by EXIP are further discussed in Section 3.

Selecting the values for the EXI options is often a trade-off between memory usage, processing speed and level of compression (for example when setting the values of valuePartitionCapacity, valueMaxLength and compression options). Furthermore, as these parameters heavily depend on the structure of the documents and even on the schema design (as shown by Bournez [2009] and White et al. [2007]) it is difficult to predict the level of efficiency when applying EXI on a particular set of XML documents without performing an extensive empirical study.

2.1.1. EXI theoretical Foundations. The goal of this section is to provide the necessary background information for supporting the discussions on the EXI processor architecture and algorithms for embedded processing that follow without going into details of the inner workings of the EXI specifications. For in-depth overview of the EXI format, the reader is advised to refer to the W3C specification [Schneider et al. 2014] and white paper [Peintner and Pericas-Geertsen 2009].

An EXI stream is a sequence of events that describe the content of the XML document. These events are analogous to the streaming XML events and denote the start of an element or attribute, value items, closing tags and so on. For achieving higher compactness, the events are represented by a simplified Huffman coding [Huffman 1952] scheme. The occurrence of each event in the EXI stream is controlled/described by a set of formal grammars. The EXI specification very broadly identifies the formal grammars used as being in restricted Greibach normal form [Greibach 1965]. Support for the theoretical fitness of the discussed grammar generation algorithms is given in the next paragraph. It provides more concrete classification of the EXI grammars.

Unlike Greibach grammars, the EXI grammars have at most one nonterminal symbol on the right-hand side of the grammar productions. Therefore, all EXI grammar rules are in one of the following two forms: 1) $Z \rightarrow aY$ or 2) $Z \rightarrow a$,where Z and Y are intermediate (nonterminal) symbols and a is a terminal symbol. As all grammar rules are in one of these two forms, the EXI grammars are also *regular* and in particular *right linear grammars* as they require exactly one terminal on the right-hand side and at most one nonterminal which is at the end of the grammar rule. The regular grammars are strict subset of the context-free grammars according to the Chomsky hierarchy, and as every context-free grammar can be represented in Greibach normal form [Greibach 1965], they are also a subset of the Greibach grammars.

Identifying the EXI grammars as regular grammars provides much more insight into their properties. For example, context-free grammars define very broad class of languages and are equivalent to pushdown automaton (PDA), while regular grammars are equivalent to nondeterministic finite automaton (NFA). Moreover, the EXI grammars are *simple* aka *s-grammars* [Korenjak and Hopcroft 1966] as each pair $Z \rightarrow a...$ appears only once in each EXI grammar. Based on this constraint, the EXI grammars are also *unambiguous* and support linear parsing time by deterministic finite automaton (DFA).

The process of converting a set of XML Schema definitions to EXI grammars includes four steps.

(1) Create a set of proto-grammars that describe the content model according to the schema. The EXI proto-grammars are strictly context-free grammars that are neither regular nor in Greibach normal form as they allow unit productions: $Z \rightarrow Y$ where both Z and Y are intermediate (nonterminal) symbols.
(2) Normalize the proto-grammars to EXI grammars. The normalization includes simplification of the proto-grammars by removal of the unit productions. This

creates regular grammar that can be *ambiguous*, in other words, lacking unique leftmost derivation tree for every input. In this case a second simplification is performed in which the *ambiguous* regular grammars are transformed to unambiguous *s-grammars*.

(3) Assign event codes to grammar productions
(4) Extend the EXI grammar with additional productions that describe the possible deviations from the XML Schema

Section 3.3 describes an extension to the algorithm for creating proto-grammars from schema definitions [step (1)] that guarantees that the resulting grammars are regular *s-grammars*. This allows for avoiding the normalization of the proto-grammars as a separate second-step process.

Section 3.4 describes a modified version of the algorithm for augmenting the EXI grammars for handling schema deviations [step (4)]. The new version of the algorithm allows the removal of redundant grammar productions that are otherwise required by the approach described in the EXI specification.

2.1.2. Related Work for XML Grammars. The formal grammars used in the EXI specification express the constraints defined in the XML Information Set [Cowan and Tobin 2004] and are not specific to EXI format itself. As such, the formal models and theoretical results developed for XML are also valid for EXI. There are two main theoretical models for studying the properties of XML languages and XML schema languages. The first model treats XML instances as strings and schema languages as formal languages that define particular sets of strings representing the possible XML instances that are valid according to a certain schema. This model is based on context-free (word) grammars and their more restricted forms such as parenthesis and balanced grammars as presented by Berstel and Boasson [2000].

In the second model, the XML instances are treated as trees and the schema languages as formal languages defining sets of trees representing the valid instances according to a certain schema [Chidlovskii 2000; Neven 2002]. The nested structure of the XML forms ordered unranked trees, that is, trees with nodes allowed to have any number of ordered child nodes. The theoretical foundation of this model are regular tree grammars which can be seen as a generalization of regular word grammars. The tree model is appropriate when studying the expressive power of different XML schema languages as shown by Murata et al. [2005]. In this work, Murata et al. present a formal classification and comparison between DTD, W3C XML Schema, and RELAX NG based on the regular tree grammar theory.

Context-free word languages and regular tree languages are closely related. For example, it is proven that the set of derivation trees for a language defined by a context-free word grammar forms a regular tree language [Comon et al. 2007]. In addition, Brüggemann-Klein et al. show that tree grammars, and even more generally hedge grammars, are effecively identical to balanced grammars and that balanced languages are identical to regular tree languages, modulo encoding [Brüggemann-Klein and Wood 2004]. These results demonstrate that the two models are equally expressive and can be used interchangeably when studying or characterizing languages based on XML Information Set.

The discussions in this article are following the first model, because the EXI specification defines the XML content with a set of regular word grammars as already presented in Section 2.1.1. For that reason, all grammars in this work are assumed to be *word* grammars even if not explicitly stated.

Instead of defining the terminal alphabet in terms of ASCII or UTF-8 characters, which is commonly used in word grammars, the EXI grammars use XML events (start element, attribute definition, end element etc.) as terminal symbols. This provides high

Table I.
Extended context-free grammar for a sample XML instance where element <notebook> can have zero or more <note> elements with optional <subject> and mandatory <body>. The following operators are used in the regular expressions in ECFG: **.** - denotes concatenation, ***** - Kleene star operator (zero or more occurrences), **?** - zero or 1 occurrence and **[ ]** - matches a single character from the specified set within the brackets. The nonterminal symbols are in uppercase letters.

Sample XML	Corresponding ECFG
<notebook> <note> <subject>Sample</subject> <body>XML Instance</body> </note> </notebook>	NOTEBOOK → <notebook>.(NOTE)*.</notebook> NOTE → <note>.(SUBJECT)?.BODY.</note> SUBJECT → <subject>.[UTF-8 characters]*.</subject> BODY → <body>.[UTF-8 characters]*.</body>

level description of the XML content model without affecting the theoretical results developed for regular grammars. As XML Information Set defines context-free language parsed by pushdown automaton, a single regular grammar (a single DFA) is, in general, unable to represent (parse) the content of a whole XML document. Using a single regular grammar (or a single DFA) for describing (parsing) the whole content of an XML is possible when certain restrictions on the document structure are met by the XML/EXI instances. For example, this approach is used for efficient processing of SOAP Web services that are *ordered* XML documents with predefined schema [Van Engelen 2004b]. A less restrictive form of schema-specific XML parsing that uses an extended version of PDA is presented by Chiu and Lu [2004]. Unlike these approaches, the EXI specification defines the parsing and serialization of XML Information Set documents based on a stack of regular grammars. Each regular grammar in the stack describes the content of particular XML element. The stack of grammars is used to model the nesting of elements (e.g., parsing a nested element equals adding its regular grammar on the stack) similarly to the role of the stack in the PDA.

For illustrating how the grammar stack is used during processing in EXI it is convenient to represent the XML Information Set in terms of extended context-free grammars (ECFG) which describe exactly the context-free languages and are the basis for DTD schema language [Wood 1995]. In an extended context-free grammar each right-hand side of a production consists of a regular expression which is in turn equivalent to regular grammar or finite automaton. Consider the example XML instance and its corresponding ECFG shown in Table I. The set of regular grammars, used during processing of EXI documents, corresponds to the set of regular expressions in ECFG which describe the content of all possible elements. At every step the EXI processor uses the regular grammars on top of the grammar stack to process the content of the current element. Starting of a nested element involves pushing its grammar to the stack and closing an element pops its grammar from the stack. In this way, parsing the XML document shown in Table I involves: (1) parse the content of <notebook> element according to the regular grammar for that element which is initially the only grammar in the stack; (2) the start of the nested <note> element requires pushing its regular grammar on the stack and parsing its content according to that grammar; (3) on start of the nested <subject> element its grammar is pushed to the stack and used for parsing; (4) When all the content of <subject> element is parsed and there are no more nested elements at this level pop its grammar from the stack and continue processing according to the <note> grammar that is currently on the top of the stack; (...) the same procedure is repeated for the rest of the elements in this example.

Unlike DTD which defines a *local* language, the language defined by the set of regular grammars in EXI is a *single-type* language that corresponds to the expressive

power of W3C XML Schema [Murata et al. 2005]. This essentially means that two or more elements sharing the same name but having different types are evaluated using different regular grammars that match their type. This differs from DTD where the name of an element uniquely identifies its content model (or, equivalently, the regular expression or the regular grammar of its content).

2.2. CoAP

The Constrained Application Protocol [Shelby et al. 2013a] is specially designed for use with resource-constrained hosts over low-bandwidth network links. CoAP functionality resembles the HTTP request/response interaction model, and is based on the Representational State Transfer (REST) architecture of the Web [Fielding and Taylor 2002]. CoAP also supports well established concepts of the Web such as URIs and Internet media types. This allows for transparent translation between CoAP and HTTP traffic while enabling Web interactions with embedded systems.

CoAP fulfills the requirements of the embedded domain such as providing support for asynchronous message exchange, multicast capabilities, lightweight discovery mechanism, very low overhead, and implementation simplicity. This is possible by using UDP as a transport protocol with optional reliable unicast support and Datagram Transport Layer Security (DTLS) instead of TCP and TLS. The use of UDP enables the implementation of CoAP lightweight publish-subscribe mechanism [Hartke 2013] supporting dynamic content exchange between embedded servers and Web clients. The built-in asynchronous exchange of events encoded with EXI provides features similar to the AJAX framework, but with much lower cost in terms of network bandwidth and hardware requirements for the hosts.

Application areas that would greatly benefit from an open and standard way to connect embedded hosts to the Web include various Internet of Things and machine-to-machine (M2M) applications such as home automation and energy management.

3. EXI PROCESSOR DESIGN AND IMPLEMENTATION

Deploying EXI-based RESTful Web services on resource-constrained hosts requires a modular implementation of the EXI processor library that can support different compile-time configurations depending on the application scenario. For example, some target platforms can make use of hash tables for fast lookups in the string tables, while others have too little RAM for that. In other cases, certain EXI options (e.g., compression, random access, etc.) are not allowed, and hence the code for processing them can be pruned from the library.

In this section, we present the modular design of the EXIP library [Kyusakov 2013] that enables compile-time profiling of the code base. As shown in Figure 2, by using fine-grained components that have low interdependencies, it is possible to define different profiles of the library that support a variety of use cases. Such profiles can be application-specific (e.g., full-featured, most-restricted, etc.), or defined as part of different communication standards - EXI Profile for limiting usage of dynamic memory [Fablet and Peintner 2014], Vehicle to grid communication interface (ISO 15118), or other energy management standards [Kyusakov et al. 2012] such as Smart Energy Profile 2.0 [Petrick and Ausdall 2013], and OpenADR, for example.

The encapsulation of the components' source code is done with the standard mechanisms available in the C programming language - splitting the code into different header and source files, and hiding the implementation in static functions, strictly avoiding the use of global variables and, where needed, using conditional C preprocessor macros. This enables the implementation of a simple and easy-to-maintain *Makefile* build system which can track the dependencies between the components. With this build system in place the developers can cherry-pick only the components that are

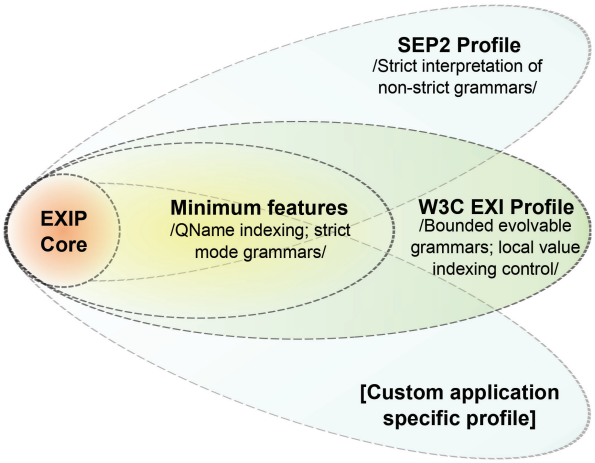

Fig. 2. EXIP modular architecture and application profiles.

needed during compile time which allows using the EXI Processor library for different application profiles or contexts.

3.1. Problem Formulation

The first step in supporting the requirements of the EXI-based embedded Web programming is to provide efficient Application Programming Interface (API) to encode and decode EXI streams. Already established XML APIs such SAX, DOM, and StAX are widely used in Java processors, but are shown to provide less than optimal efficiency for resource-constrained devices Kyusakov et al. [2011b]. Other requirements of the EXI processor implementation include a small footprint and an easy-to-use code base that executes quickly, and consumes as little RAM as possible while being portable across a wide range of embedded platforms. Although the main goal of the EXIP library directly follows from these requirements, detailed description and evaluation of the degree to which these requirements are met is out of the scope of this article. The reason for excluding these discussions is the low research value of the implementation technicalities that are involved in writing efficient and portable C code, a subject which is better presented by the EXIP library developers' documentation.[2] Instead, this section is focused solely on the grammar generation functionality that is an essential part of a number of use cases connected to dynamic/runtime exchange of schema information. The need for such runtime negotiation of the document structure is evident in supporting versioning of the schema documents and implementing generic Web services such as information logging and archiving, data visualization of uncategorized information, dynamic Web service composition, and peer-to-peer services. A concrete example where the XML Schema documents are processed during runtime to generate EXI grammars is the specification draft for using EXI over Extensible Messaging and Presence Protocol (XMPP) [Waher and Doi 2013].

The dynamic processing of XML schema information can also be employed in cases where no schema information is available to describe a particular set of XML documents. In such cases the XML schema can be inferred from the set of available XML examples, and used to enable more compact EXI encoding. Both the schema inference and the generation of EXI grammars can be done at runtime. Example approaches

[2]Available at http://exip.sourceforge.net/.

for schema inference include learning of deterministic regular expressions [Bex et al. 2010], as well as learning chain regular expressions, in the case of the Trang open source software library [Clark 2013].

3.2. Efficient EXI Grammar Generation

The standard way of generating EXI grammars from XML Schema is to rely on a generic XML Schema parser/validator such as the Apache Xerces library. The role of the XML Schema parser is to load the schema definitions into appropriate structures in the memory. These structures are then converted to EXI grammars based on the algorithms specified in the EXI specification. The EXIP library takes a different approach by including a dedicated EXI grammar generator without external dependencies on schema parsers, which uses a modified version of the algorithms described in the EXI specification.

Many embedded targets use EXI because XML processing is too heavy to support. In such cases, the dynamic generation of the EXI grammars cannot be achieved in a standard way, as it requires processing text-based XML schema definitions. One possible solution is to use proprietary encoding for the EXI grammars, which is against the principles of the Web, and will still require some loading code that expands the programming memory footprint.

The dedicated EXI grammar generator solves this problem by using two simple ideas. First, the XML Schema document is itself an XML document that can be represented in binary using EXI, thus reducing its size and improving the loading time. Second, once represented in EXI, the XML Schema document can be parsed by the EXI parser itself without the need of an external library for that; in other words, the EXI decoder code is reused to extract the XML schema definitions.

3.3. EXI Grammar Concatenation and Normalization

The EXI specification defines an algorithm for building a set of context-free grammars that directly correspond to the definitions in the W3C XML Schema specification. These grammars are called proto-grammars as they are intermediate representation which is only used during EXI grammar generation. The process of building proto-grammars is roughly as follow.

(1) A set of simple proto-grammars is defined that describes the content model for each atomic XML schema definition (attributes, simple types, element terms, wildcard terms).
(2) The proto-grammars for composite schema definitions are built by using the proto-grammars of their sub-components. For example, the <*sequence*> compositor equals to concatenation of the proto-grammars of its child elements and <*choice*> compositor equals to the union of the proto-grammars of its children.

The next step in the process of building EXI grammars is to normalize the proto-grammars such that all unit productions ($Z \rightarrow Y$ where both Z and Y are intermediate symbols) are removed and there are no ambiguities in the grammars. This essentially converts the proto-grammars to EXI grammars that are then used for processing EXI documents conforming to a schema.

The review of the algorithm for creating EXI proto-grammars from XML Schema definitions in section 8.5.4.1 EXI Proto-Grammars of the EXI specification leads to the conclusion that the only way for creating proto-grammars that contain unit productions, and hence are not regular, is as an output of the grammar concatenation operator (see 8.5.4.1.1 Grammar Concatenation Operator of the specification). However, all atomic grammars used as an input to the concatenation operator are regular and from the closure property of the regular languages under concatenation [Hopcroft and Ullman

1969] we know that the resulting output grammar can also be presented in a regular form.

This section defines an extended grammar concatenation operator that produces regular EXI grammars, thereby removing the need for additional normalization of the grammars by removal of unit productions. The extended operator depends on the following recursive definition.

Definition: Weak equality of grammar productions. The grammar production $A : Z_1 \rightarrow a_1 Y_1$ and the grammar production $B : Z_2 \rightarrow a_2 Y_2$ are *weakly equivalent* if:

(1) $a_1 \equiv a_2$ and $Y_1 \equiv Y_2$
OR
(2) $a_1 \equiv a_2$. Let the sets of productions in the EXI grammar that have Y_1 and Y_2 as a left-hand side be denoted as $\{Y_1\}$ and $\{Y_2\}$ respectively. The two sets have the same cardinality, and each production $P \in \{Y_1\}$ is *weakly equivalent* to a production in $\{Y_2\}$.

The grammar concatenation operator defined below is very similar to the one in the EXI specification in the sense that it creates a new grammar given two input grammars. The new grammar accepts any set of symbols accepted by the left operand followed by any set of symbols accepted by the right operand of the concatenation operator. The main difference is that the operator defined here produces regular EXI grammars, given its operators are also regular grammars.

Definition: Extended grammar concatenation operator. Given two EXI Grammars $L(N_l, T, S_l, P_l)$ and $R(N_r, T, S_r, P_r)$ where N_l and N_r are finite sets of non-terminals, T is the set of terminal symbols representing the EXI events, $S_l \in N_l$ and $S_r \in N_r$ are both designated initial symbols, and P_l and P_r are the sets of grammar productions in L and R respectively. All grammar productions in P_l and P_r are in one of the following two forms: $Z \rightarrow aY$ where $a \in T$ and $a \neq EE$ or $Z \rightarrow EE$ where $EE \in T$ is the terminating *end element* EXI event.

The result of applying the grammar concatenation operator to L and R, $L \bigoplus R$, is a new grammar $C(N_l \cup N_r, T, S_l, P_c)$ where the set of productions P_c is defined as follows: each production $l \in P_l$, where $l \neq Z \rightarrow EE$ for every $Z \in N_l$, is part of P_c; each production $r \in P_r$, where $r \neq S_r \rightarrow aY$ for every $a \in T$, and $Y \in N_r$ is part of P_c. For each production $e_l \in P_l$, where $e_l \equiv Z \rightarrow EE$ for every $Z \in N_l$, the following set of productions is also part of P_c: the set $\{Z \rightarrow aY\}$ where a production s_r of the form $S_r \rightarrow aY$ exists in P_r, and s_r is not *weakly equivalent* to any production in P_l that has Z as a left-hand side nonterminal symbol. There are no other productions in P_c besides those defined with these rules.

When the extended concatenation operator is used for XML Schema ⟨*sequence*⟩ definitions, the resulting regular grammar might contain productions with duplicate terminal symbols, that is, the result can be an ambiguous regular grammar. In this case the algorithm in section *8.5.4.2.2 Eliminating Duplicate Terminal Symbols* of the EXI specification should be further applied to the resulting concatenated EXI grammar. It is worth noting that these cases are extremely rare and can only occur when optional element particles are allowed to repeat more than once. Example content model that contains duplicate terminal symbols and leads to the creation of ambiguous regular grammar is the following.

```
<sequence maxOccurs=''2''>
  <element name=''a'' maxOccurs=''3''/>
  <element name=''b'' minOccurs=''0''/>
</sequence>
```

3.4. Efficient Representation of Schema Deviations

The EXI specification defines an algorithm that augments the EXI Grammars with additional grammar productions which are used to handle possible deviations from the XML schema. Such deviations are often used to add extensions to a particular protocol or handle cases that require additional information in the XML documents. Furthermore, certain XML events that are not explicitly declared in the schema may also occur in the instance documents without making them invalid (e.g., comments, processing-instructions, type casts using *type* attribute from `http://www.w3.org/2001/XMLSchema-instance` namespace).

One constraint that must be followed when adding productions to the normalized EXI grammars is that addition of productions allowing attribute deviations must only occur before the element content - otherwise the grammars describe a document which is not well formed. The algorithm as described in the EXI specification (see 8.5.4.4.1 Adding Productions when Strict is False [Schneider et al. 2014]) depends on a set of redundant productions in the normalized EXI grammars in order to fulfill this requirement. The redundant productions are a copy of the productions describing the possible states for starting the content of an XML element that has wildcard attributes or a mixed-content model. An example of such redundant productions is the EXI grammar describing element fragments (see 8.5.3 Schema-informed Element Fragment Grammar [Schneider et al. 2014]).

The algorithm described in this section augments the EXI grammars for accepting schema deviations without having a dependency on redundant productions in the input EXI grammar. The algorithm is presented by highlighting only the modifications and differences with comparison to the algorithm in the EXI specification. An example of applying the modified algorithm is given in Appendix A.

The algorithm depends on the definition of a *content* nonterminal symbol, and an index called content index for each input EXI grammar. The assignment of content index and content to a nonterminal symbol is identical to the process defined in the EXI specification, and a prose description of it is given here.

Definition. Content nonterminal symbol. The *content* nonterminal symbol is the symbol that indicates that all attributes (AT terminal symbols) are already encoded. The *content* nonterminal symbol represents all the states for starting the encoding of the content of a particular XML element.

Definition. Content index. Assign index numbers to all nonterminal symbols such that the designated initial symbol of the EXI grammar has index 0 and all other indexes are larger than 0. The index of the *content* nonterminal symbol, in other words, the *content index*, is then the smallest index that is larger than the indexes of all nonterminal symbols that are used as a left-hand side in grammar productions with AT terminals.

Definition. Grammar augmentation for schema deviations. Create a copy of all grammar productions that have the *content* nonterminal on the left-hand side if and only if there are AT productions that have the *content* nonterminal symbol on their right-hand side or the *content index* is 0. The copy of the *content* nonterminal symbol - *content2* if available, is inserted just before the *content* that is, it has index of (*content index* - 1). In the case when the *content index* is 0, that would mean that the *content2* is now the entry nonterminal symbol of the grammar. After the copying, there should be no productions with *content2* nonterminal on the left-hand side that have *content2* on their right-hand side - instead they should have only *content*. All AT productions that have a *content* nonterminal symbol on their right-hand side are changed to point towards *content2* instead.

Apply the procedure in 8.5.4.4.1 Adding Productions when Strict is False [Schneider et al. 2014] while applying the following modifications to the algorithm:

—The designated initial symbol of the EXI grammar is changed to *content2* when *content index* is 0.
—Change each occurrence of *content* with *content2* and vice versa, that is, each occurrence of *content2* with *content*.
—If there is no *content2* nonterminal, then do not perform the procedure for it and assume the *content2 index* is smaller than the *content index*, but larger than the indexes of all nonterminals that are used in AT productions.

4. PERFORMANCE EVALUATION

The goal of this section is to evaluate the performance of the dedicated EXI grammar generator implemented as part of the EXIP library. The grammar generator accepts EXI encoded XML Schema definitions as an input, and uses the extended grammar concatenation operator and the algorithm for efficient representation of schema deviations. The measurements in this section are indicative and aim to compare the execution time and memory usage of grammar generation on real-world data. As the core contribution of this work is in the grammar generation utility, this section does not evaluated the overall EXI processing performance. Measurements of the EXI parsing speed are included only to the extent needed to put the grammar generation evaluation in context.

Description of the Test Setup. A set of 5 XML schema documents were used for decoding 15 instances (XML examples that conform to the schema; 3 instances per each schema document) by 3 different EXI processors. Decoding in this experiment refers to converting a binary EXI file to its text-based XML representation. The EXI processors are EXIficient v0.9.1 Java [Peintner 2013], OpenEXI v1.0238.0 Java [Kamiya 2013], and EXIP v0.5.3 C [Kyusakov 2014]. At the time of writing this article - June 2014, there is one more open source EXI parser - WS4D-uEXI.[3] WS4D-uEXI is written in C and is designed for constrained embedded devices. It is not included in this comparison as it uses EXIficient library for building the EXI grammars at compile time and therefore does not support runtime grammar generation [Moritz et al. 2013]. Moreover, WS4D-uEXI implements a subset of the EXI specification and its current version (SVN r2) is unable to decode some of the EXI instances in this evaluation due to missing features.

The evaluation uses the following XML schema documents: netconf.xsd,[4] SenML.xsd,[5] sep.xsd,[6] OPC-UA-Types.xsd,[7] and XMLSchema.xsd.[8] All of them were accessed from the local hard-drive, including the imported XML schema files, so there were no dependencies on the network performance.

The tests were executed on a desktop PC (Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz, 4GB RAM @ 1067 MHz) running 32-bit Linux Ubuntu 13.10. The version of the Java Virtual Machine (JVM) used for running EXIficient and OpenEXI is Java HotSpot(TM) Server VM 1.7, and the C compiler used for EXIP is GCC 4.8.1.

Two distinct measurements of the execution time were performed for each EXI processor: (1) the time it takes for loading an XML Schema and converting it to EXI grammars, and (2) the time it takes to generate the EXI grammars as well as decode

[3]http://code.google.com/p/ws4d-uexi/.
[4]Network Configuration Protocol: https://www.iana.org/assignments/xml-registry/schema/netconf.xsd.
[5]Sensor Markup Language: http://tools.ietf.org/html/draft-jennings-senml-10.
[6]SEP2: http://www.zigbee.org/Standards/ZigBeeSmartEnergy/SmartEnergyProfile2.aspx.
[7]OPC-UA: http://opcfoundation.org/UA/2008/02/Types.xsd.
[8]Schema for XML Schema: http://www.w3.org/2001/XMLSchema.

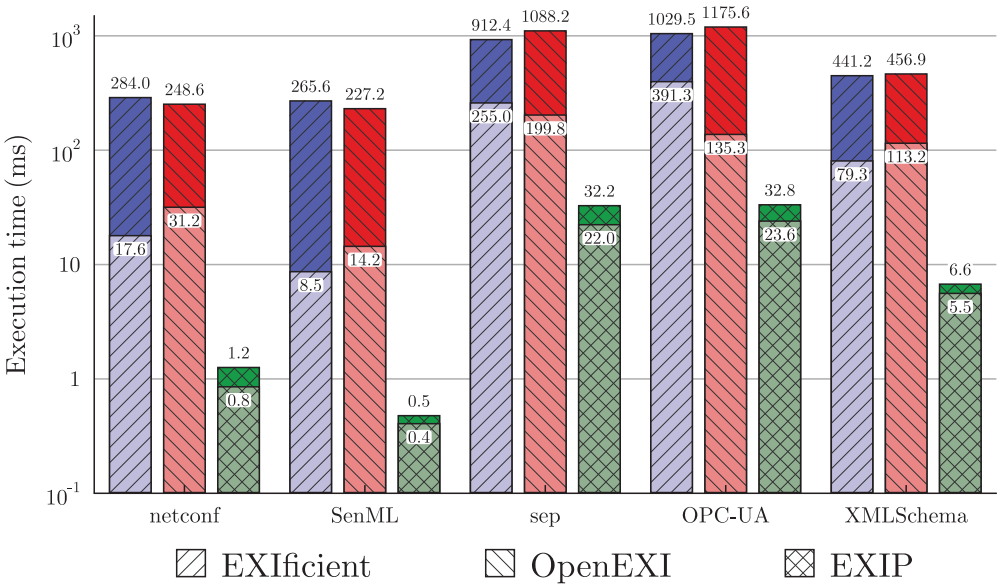

Fig. 3. Grammar generation execution times for each XML Schema test case. The averaged times per XML schema are given on the logarithmic Y axis for each of the tested EXI processors - EXIficient (leftmost column, forward slash hatching), OpenEXI (middle column, backslash hatching) and EXIP (rightmost column, grid hatching). Each bar in the chart represents the execution times when explicit optimizations are applied (lighter colored part of the bar) and when no optimizations are applied.

a sample XML instance. The time was measured using $System.nanoTime()$ in Java and $clock\_gettime()$ in C, in other words, we measured wall-clock time which can vary depending on the external load of the system. In order to get comparable results, the tests were executed ensuring similar conditions on the system load, and taking the mean value of 300 measurements. Moreover, the mean value is calculated for two distinct runs of the test framework - one with optimizations and one without applying optimizations. In the unoptimized case the Java processors run on a "cold" JVM that is, the code is executed for the first time on the VM and hence the classes for grammar generation and instance decoding are loaded at runtime. Also the "cold" JVM has smaller chance for applying runtime optimizations such as Just-In-Time (JIT) compilation. Conversely, the optimized case uses "warmed-up" JVM where the tests are run 5 times on the JVM before the measurement are taken. The EXIP processor is compiled with $-O0$ flag for unoptimized case and with $-O3$ for the optimized run.[9]

Figure 3 and Figure 4 show the averaged execution times per each XML schema test case with enabled and disabled optimizations. In Figure 3 the times are for grammar generation only while Figure 4 shows the execution times for both grammar generation and instance decoding. In both charts, the execution times on the Y axis are represented in logarithmic scale for enhancing the visual representation.

On average, among all test cases, the execution times for grammar generation and instance decoding are given in Table II. As shown in the table, EXIP generates the grammars about 9 times faster than OpenEXI and 14 times faster than EXIficient when compile time optimizations for the C code and runtime JVM optimizations for the Java code are enabled. This cannot be attributed solely to the performance difference

[9]The automated test framework for configuring and executing the evaluation is available open source at http://github.com/kjussakov/exip-eval.

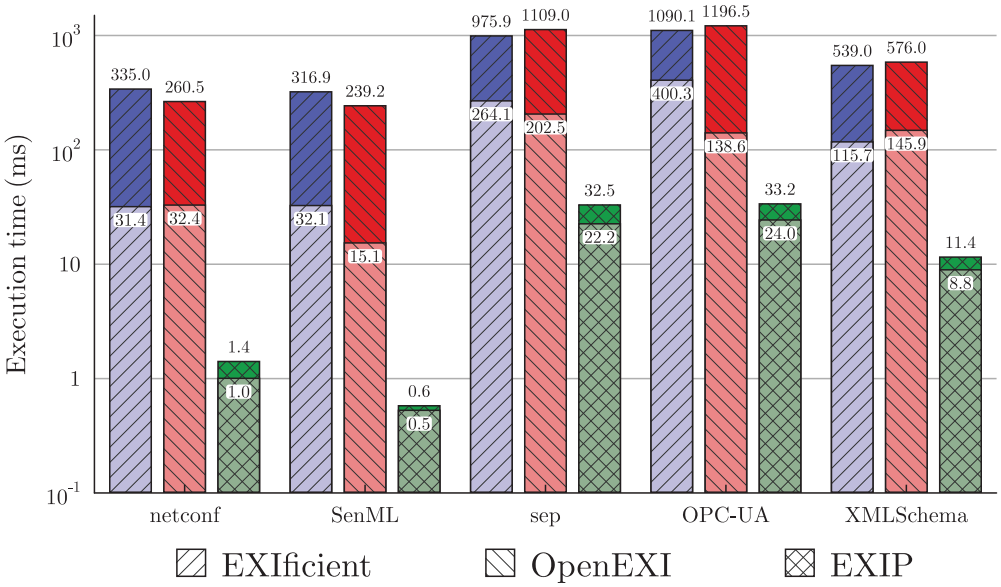

Fig. 4. Grammar generation and instance decoding execution times for each XML Schema test case. The averaged times per XML schema are given on the logarithmic Y axis for each of the tested EXI processors - EXIficient (leftmost column, forward slash hatching), OpenEXI (middle column, backslash hatching) and EXIP (rightmost column, grid hatching). Each bar in the chart represents the execution times when explicit optimizations are applied (lighter colored part of the bar) and when no optimizations are applied.

Table II. Averaged Execution Times (*ms*) for all XML Schema Test Cases

	Optimized		Unoptimized	
EXI Processor	Grammar	Grammar + Instance	Grammar	Grammar + Instance
EXIficient	150.3	168.7	586.5	651.4
OpenEXI	98.8	106.9	639.3	676.2
EXIP	10.5	11.3	14.7	15.8

in native code versus Java byte code execution where on average Java programs are somewhere between 50% faster to 4 times slower than their C counterparts.[10]

The superior performance of EXIP grammar generation is mainly due to the use of EXI-specific XML Schema parser that accepts EXI encoded XML Schema definitions as opposed to the use of general purpose XML Schema parser. By using the extended grammar concatenation operator (see Section 3.3), EXIP has to perform one less iteration over the set of all grammar rules which has noticeable benefits mainly in large XML Schemas such as SEP2 (sep.xsd). The grammar augmentation algorithm presented in Section 3.4 has no effect on processing efficiency, but slightly improves memory usage. Code optimizations, in terms of avoiding unnecessary loops and selecting appropriate searching and sorting algorithms (for example the use of a hash table for mapping element definitions to their globally defined types instead of iteration), have impact on the performance as well but are harder to quantify.

4.1. Memory usage

This section provides some insight into the memory consumption of EXIP, and EXI in general, as memory is often a bottleneck in embedded system applications. Section 2.1

[10]Source: http://benchmarksgame.alioth.debian.org/u32/java.php.

Table III.
Size of a SenML instance for different encoding modes and memory usage for EXIP and the light-weight XML parser library MiniXML on a Raspberry Pi system. The rows are ordered by document size.

Encoding mode	Size (bytes)	RAM/heap usage (kB)			
		EXIP		MiniXML	
		Encoding	Decoding	Encoding	Decoding
Plain XML	387	-	-	1.36	1.55
EXI Schema-less byte aligned	248	7.95	8.26	-	-
EXI Schema-less no value indexing	237	6.93	6.79	-	-
EXI Schema-less default options	200	7.90	8.26	-	-
EXI Schema mode no value indexing	137	1.93	2.27	-	-
EXI Schema mode default options	100	2.87	2.21	-	-
EXI Schema mode strict	98	2.89	2.23	-	-

already discussed that the dynamic memory usage for EXI processing can be controlled by some of the parameters defined in the EXI header. This is done by adjusting the extent of the content indexing used to detect and reduce redundancy in the data which also affects the compactness and processing speed. However, the mechanisms provided in the EXI specification cannot guarantee bounded runtime memory usage when deviations from the XML schema are present. For that purpose, an extension to these mechanisms are developed in a complementary specification called EXI Profile for limiting usage of dynamic memory [Fablet and Peintner 2014]. A subset of this profile is supported by EXIP but its impact on the memory consumption is not evaluated in this section as the tests presented here are restricted to a schema valid instance of the SenML standard. Table III shows the size and memory usage during encoding and decoding for a sample instance document borrowed from the SenML specification.[11]

The size and memory consumption are given for different encoding options. The platform used for testing is Raspberry Pi embedded computer with ARM-based system on chip including 700 MHz processor with 512 MB of RAM. The memory usage presented in Table III shows only the amount of dynamic memory (heap) usage for statically compiled EXI grammars and is measured using DHAT (dynamic heap analysis tool) that is part of the code profiling library Valgrind.

An interesting observation is that although the document is relatively small, turning off the indexing of repeating values (i.e., setting *valuePartitionCapacity* parameter to 0) substantially inflates the size of the resulting EXI representation. This is due to the high redundancy in the attribute values which has profound affect even in schema mode encoding. This simple example shows the high variation of compression and dynamic memory usage depending on the content of the documents and the encoding options in use.

The compile-time allocated RAM used by the EXIP library (calculated as the sum of .rodata, .data and .bss sections in the Executable and Linking Format (ELF)) is 23kB (of which 8kB EXI grammar definitions used for schema mode cases) while the light-weight XML parser MiniXML v2.8 requires only 3kB. EXIP SenML parser uses 79kB programming memory while MiniXML uses only 16kB. Additionally, as shown in Table III, MiniXML is more efficient in the use of dynamic memory compared to EXIP. These results indicate that EXI processing, and EXIP library in particular, require more RAM compared to highly optimized XML processing. The main reason for this is the use of content indexing and grammar information during EXI processing. Further optimizations of the RAM usage in EXIP are possible both for the size of the content index as well as the in-memory grammar representation. It should also be

[11]Available at: http://tools.ietf.org/html/draft-jennings-senml-10#section-7.

noted that schema-based EXI processing implicitly performs partial schema validation while MiniXML is a non-validating parser.

Enabling runtime EXI grammar generation from the SenML schema additionally requires 57kB of dynamic memory and 37kB of programming memory. These memory requirements show that the runtime grammar generation module fits easily in embedded devices such as Raspberry Pi but is too heavy for the most constrained platforms. As an example, the popular Stellaris LM4F120H5QR 32-bit ARM Cortex-M4F microcontroller (80MHz CPU frequency, 256KB flash and 32KB SRAM) does not have enough RAM for supporting runtime EXI grammar generation. Nevertheless, by using static grammars the EXIP library is capable of running on such platforms with averaged total RAM usage of about 20kB[12] and 60kB of programming memory for the SenML sample instance.

5. EXI DATA BINDING

The information contained in an XML/EXI document is often loaded into the memory for further processing and mapped to a hierarchy of data structures or objects that are maintained by the applications. For example, a status report by a device can include various hierarchical information such as network status (which in turn contains parameters like RTT, signal strength, connected peers etc.) or resource utilization (storage space, battery level etc.) that is mapped to a corresponding hierarchy of programming objects. The process of generating an XML/EXI document from a hierarchy of objects and vice versa is known as *XML/EXI data binding*. The process of building objects from an XML/EXI input document is called *unmarshalling* and the reverse, the generation of XML/EXI output document from objects, is called *marshalling*. The unmarshalling is implemented as a software module that connects to the parser API, and generates memory structures that correspond to the structure and content of the XML document. The marshalling is implemented as a module that transforms a set of objects in the memory to a sequence of calls to the serialization API.

The *XML/EXI* data binding code can be complex to write and maintain manually. For that reason, it is often automatically generated. There are two main approaches when generating the code and keeping it in sync with the XML/EXI documents; direct, and indirect mapping. In direct mapping, the source code is generated based on XML schema definitions or vice versa; the XML schema can be built based on the existing source code definitions. When no schema information is available or needed, the XML tree can be directly mapped to a memory representation, as in the case of the Document Object Model (DOM). The data binding frameworks that are based on direct mapping of the XML Information Set and the memory representation, are widely adopted in desktop and enterprise applications: examples include DOM, JAXB, XMLBeans, and others [Simeoni et al. 2003]. Their main advantage is that it is very easy to build and maintain the XML-to-source code mapping. An example of a pure XML direct mapping framework for embedded systems development is the gSOAP toolkit [van Engelen and Gallivany 2002]. A similar approach, but applied to EXI and targeted at highly resource-constrained embedded devices is the automatic EXI Processor generation reported by Käbisch et al. [2010].

The indirect mapping is a more flexible approach that allows discarding the unnecessary XML structures or reusing existing objects in the memory by defining a layer of indirection between the XML Information Set and the memory representation. Example libraries in this category include Castor and JiBX [Sosnoski 2014] - both only available in Java, and targeted at server/desktop applications. A comparison

[12] The RAM usage in schema mode is 20kB (1kB stack size + 2.5kB heap + 16.5kB .data and .bss) while the RAM usage in schema-less mode is 19kB (1kB stack size + 7.5kB heap + 10.5kB .data and .bss).

between the two approaches, that is, direct and indirect mapping, along with performance measurements, are presented by Sosnoski in IBM developerWorks article on data binding tools for Java/XML [Sosnoski 2003].

The EXI binding presented in this section falls into the category of indirect mapping, and it is targeted at embedded systems development. Its design is based on the following requirements.

—The mapping rules should have intuitive syntax and semantics.
—The binding definitions should be independent from the programming language in use - the same binding definition should work for programs written in C, Java, Python, and so on.
—The EXI binding should be efficient to use on embedded platforms.
—The mapping layer should allow for loading the binding definitions and building the objects in memory dynamically at runtime.

To optimally fulfill these requirements, we propose template-based binding definitions that are written in XML and converted to EXI before being used for code generation or loading at runtime. The binding templates are very similar to other frameworks for dynamic content delivery based on templates such as JavaServer Pages (JSP) technology. An in-depth overview of template-based code generation is presented by Arnoldus et al. [2012] where the authors describe the theoretical foundations of template systems and include comparison with other code generation techniques. The proposed EXI template framework is a heterogeneous code generator that follows the model-view-controller design pattern as suggested by Arnoldus et al. [2012].

Figure 5 shows a comparison of this approach to what is a commonly used method for defining such binding definitions. As depicted, the mapping between dynamic EXI content and programming constructs is done using a special character @ and semicolon notation. As such, the definitions are intuitive to define as well as simple to process by the loading code. As with other such approaches based on templates, these special characters must be escaped when used in a static content. As an example, the value for a static attribute *email* within an EXI binding definition should be defined as *example@@com* to escape the special character that indicates the beginning of dynamic content mapping.

6. COAP/EXI/XHTML WEB PAGE ENGINE

This section presents a prototype implementation of a dynamic Web interface for an embedded sensor platform based on CoAP/EXI/XHTML technologies. The implementation is developed using the EXIP framework, and consists of an experimental Java browser running on a laptop PC that connects to a wireless sensor device (Mulle version 3.2 [Eliasson et al. 2008]) over Bluetooth. The laptop user can navigate to the device Web interface using mDNS/DNS-SD or CoAP built-in discovery capabilities - multicast service discovery [Shelby et al. 2013a], or CoRE Resource Directory [Shelby et al. 2013b]. In our simplified test setup, the network address of the sensor device is predefined so the discovery process was not implemented.

The EXI encoded XHTML page is dynamically generated on the sensor platform on a CoAP GET request, and it contains an *iframe* tag with a link to an external observable CoAP resource.

```
...
<p>Current temperature is:</p>
<iframe src="coap://192.168.150.10:5683/temp"/>
...
```

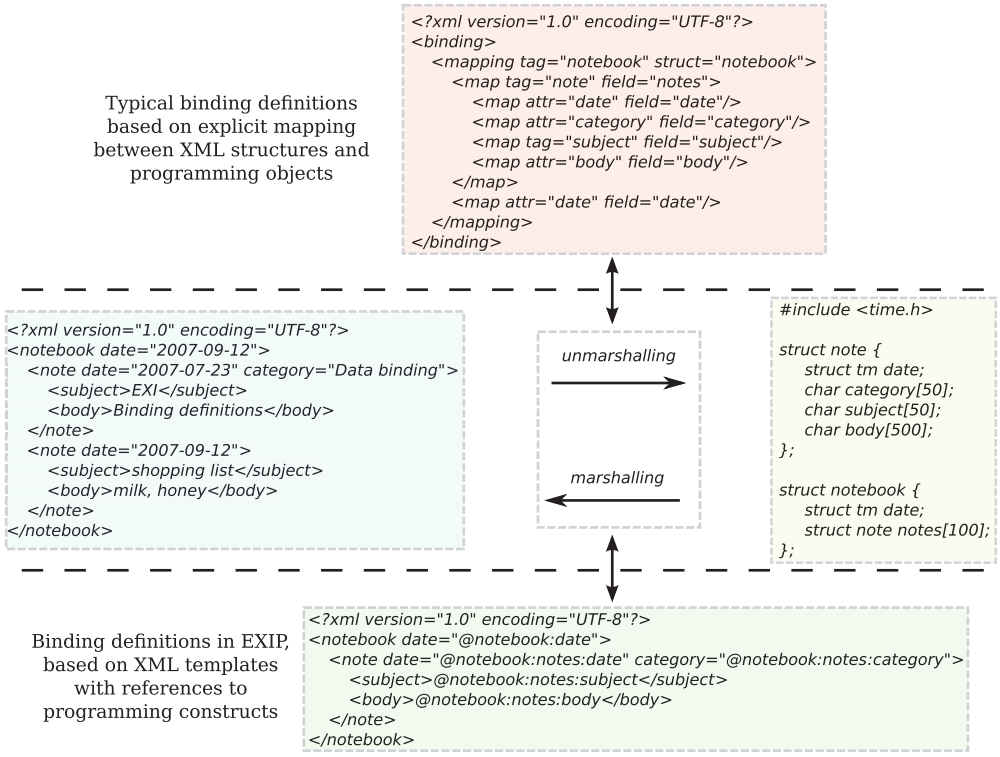

Fig. 5. Comparison between typical binding definitions and the EXIP templates.

The observable CoAP resource is linked to a temperature sensor on the wireless device, and is updated whenever there is a change in the measured air temperature.

As shown in Figure 6, upon user request, the Java browser[13] sends a CoAP GET request to the sensor device using the *Californium* library. The wireless device receives the request and generates a CoAP response using the *libcoap* library. The CoAP version 13 payload is a dynamically generated EXI/XHTML Web page using the EXIP framework. Once the packet is transmitted back to the Java browser, the EXI document is decoded by the *OpenEXI* library, and the *iframe* link is resolved by initiating an additional CoAP request to fetch the temperature. By using CoAP Observe [Hartke 2013], the Java client subscribes to changes in the temperature without requiring additional periodic polling requests. The temperature is represented in plain text, and visualized on the browser window each time a CoAP notification is received.

This prototype demonstrates how the newly emerging binary Web protocols can be employed to enable a dynamic Web interface for highly resource-constrained embedded devices. The Web interface can be used in a wide range of mobile applications, as suggested by Puñal Pereira et al. [2013]. The approach of using the *iframe* tag with CoAP Observe enables very lightweight event-based content delivery that is suitable for low-power radio communications such as IEEE 802.15.4 (6LoWPAN, ZigBee), Z-Wave, or Bluetooth low energy. Example application domains for the EXIP framework include, but are not limited to: industrial process monitoring and control, eHealth and elderly care, wearable electronics, home automation, and energy management.

[13]Based on Flying Saucer XHTML/CSS 2.1 renderer: https://code.google.com/p/flying-saucer/.

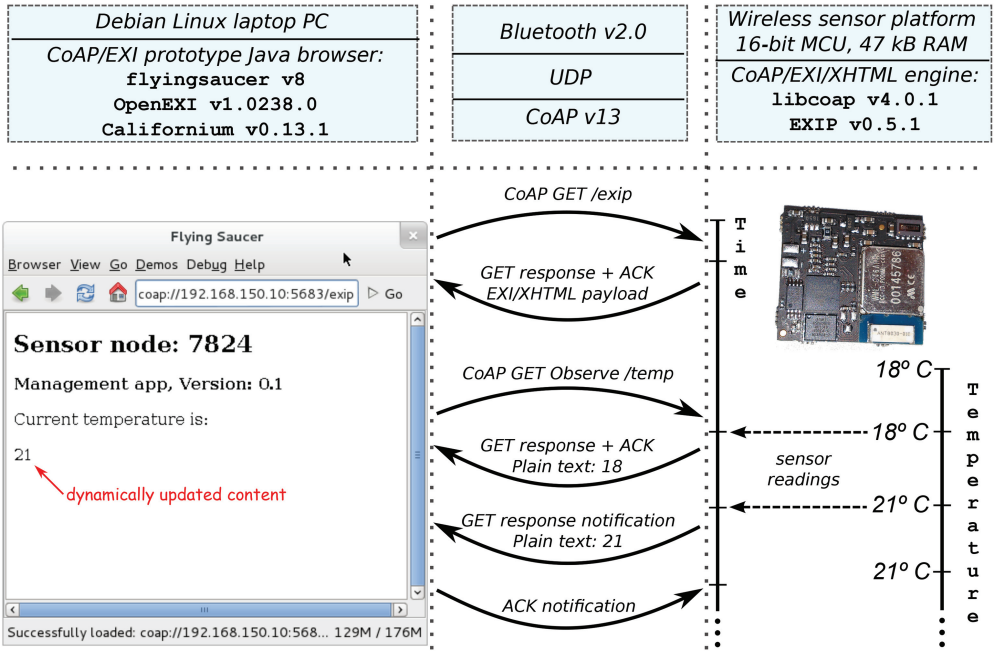

Fig. 6. CoAP/EXI/XHTML dynamic Web interface demonstration.

Data visualization technologies based on XML encoding such as SVG[14] and X3D[15] can be readily included in the CoAP/EXI/XHTML engine to efficiently represent graphical indicators (e.g., battery level, signal strength) and visualize measurements and configuration parameters. An evaluation of EXI encoding for SVG in rich media applications for embedded systems presented by [Peintner et al. 2009] shows that EXI significantly increases the efficiency of the SVG format. Also shown in this work is an approach using the EXI header option *datatypeRepresentationMap* to further optimize the compression of graphics formats for embedded Web applications.

The presented CoAP/EXI/XHTML Java browser always tries to subscribe to the *iframe* CoAP links - if the resource is not observable, the subscription is not established. When the resource is observable but should be treated statically for display in the browser (for example representing a snapshot of dynamic data), the embedded server should reject the subscription request by the browser. This approach can be too limited in certain scenarios, in which case different ways to indicate whether the browser should subscribe to changes on the *iframe* resource can be employed; adding an extra boolean argument *observe* to the *iframe* tag as an XHTML schema deviation, or requesting the resource description in CoRE Link Format before sending the subscription request. Similarly, in more complex scenarios the use of plain text encoding for the *iframe* resource might be too limiting. In such cases a structured format such as EXI can be used instead of plain text. The definition of the data format (including parameters and schema if available) for particular *iframe* can be defined as XHTML schema deviation or read from the CoRE Link Format as suggested for the observe use case.

[14]Scalable Vector Graphics (SVG): http://www.w3.org/TR/SVG11/.
[15]X3D Specification for 3D Graphics: www.web3d.org/x3d.

Implementation Details. The information provided hereafter gives more insight into the actual implementation, and is useful for reproducing the test setup. The Mulle sensor platform has a 16-bit Renesas M16C/62P microcontroller and Mitsumi Bluetooth 2.0 wireless module. The application runs on bare metal, in other words, without an OS, on top of a port of *lwIP* TCP/IP stack and a *libcoap* v4.0.1 library. The EXI/XHTML generation is done in schema-less mode using *EXIP* v0.5.1.

The laptop PC is running Debian Linux, and is equipped with a USB Bluetooth 2.0 adapter. Debian packages *bluez-compat v4.99-2*, *bridge-utils v1.5-6*, and *isc-dhcp-server* were installed and configured on the system to enable TCP/IP communication over Bluetooth.

The size of the EXI/XHTML Web page is 239 bytes, and is generated directly in binary (EXI) form without transition to plain XML. If converted to text XHTML, the size is 427 bytes. The temperature notifications are in plain text, and account for 14 bytes of CoAP packet size (UDP payload) in total, assuming 2 bytes for the plain text temperature value.

7. CONCLUSIONS

The newly emerging transport and data representation protocols based on binary encoding—CoAP and EXI—provide an efficient way to connect embedded systems to the Web across scenarios as diverse as mobile computing, home automation, and smart grid. As the translations between CoAP $\Leftrightarrow$ HTTP and EXI $\Leftrightarrow$ XML are well defined, the integration of these binary protocols to the existing Web infrastructure is standardized and conforms to the well-established programming interfaces. For example, EXI processors often provide the same API as XML processors, and CoAP/HTTP proxies are simple to deploy and are transparent for the Web users.

The work presented in this article shows that the use of CoAP/EXI stack and the EXIP Web development toolkit enables reuse of the existing pool of Web technologies and developers' skills, even on very resource-constrained embedded platforms. The development process and especially the integration with existing systems is much faster and easier to maintain as compared to the use of handcrafted communication protocols.

Moreover, the presented EXI processor design, and the EXI grammar generation algorithms in particular, provide superior processing performance compared to the methods described in the EXI specification with order-of-magnitude speed-up in some of the test cases. This could enable exchange patterns supporting dynamic XML schema negotiations even for embedded hosts. The use cases for such an approach include support for schema versioning, generic Web services, and runtime service composition.

Finally, the presented prototype of dynamic Web interface for sensor platforms demonstrates the possibility to use event-based Web content delivery with a very low overhead in terms of network bandwidth and processing power. The development of the Web interface or Web service exchange can be automated by using the template-based EXI data binding. As the data binding creates indirect mapping between the EXI document and the programming constructs, the memory structures and programming objects can be reused when generating or decoding the EXI streams.

Possible extensions of this work include in-depth memory consumption evaluation and trade-off analysis as well as developing a formal specification of the EXIP data binding, and implementing prototypes in C and Java to evaluate the proposed approach against existing XML data binding frameworks. Providing support for light-weight client-side scripting as part of the CoAP/EXI/XHTML embedded Web programming is also an interesting and important topic for future investigation. It is also worth analyzing the application of CoAP/EXI, and the EXIP framework in particular, for mobile platforms and even for desktop applications that are not resource-constrained.

Lowering the network traffic and CPU cycles for Web content delivery on mobile phones and tablet PCs could potentially increase the battery life for these devices, lower the networking cost for both operators and users, and even lead to energy savings if applied on a global scale.

APPENDIX

Grammar Augmentation Algorithm Example

This appendix gives an example of how the augmentation procedure is applied to the wildcard XML schema type *anyType* which is the base type definition for all other XML schema types. A minimal (without redundant productions) EXI grammar that describes the content model according to the process of creating proto-grammars is as follows.

```
anyType−0:
        AT(∗)   anyType−0
        SE(∗)   anyType−1
        EE
        CH      anyType−1

anyType−1:
        SE(∗)   anyType−1
        EE
        CH      anyType−1
```

There is no need to copy the content grammar productions (the ones with anyType-1 on the left-hand side) because there are no AT productions that points to it and the *content index* is 1.

Then apply the procedure in 8.5.4.4.1 Adding Productions when Strict is False [Schneider et al. 2014]:

As there is EE production already do not add additional one. Adding the AT(xsi:type) and AT(xsi:nil) productions produces the following.

```
anyType−0:
        AT(∗)           anyType−0
        SE(∗)           anyType−1
        EE
        CH              anyType−1
        AT(xsi:type)    anyType−0
        AT(xsi:nil)     anyType−0

anyType−1:
        SE(∗)           anyType−1
        EE
        CH              anyType−1
```

"For each nonterminal $Element_{i,j}$, such that $0 \leq j \leq content$ ..."
becomes:
"For each nonterminal $Element_{i,j}$, such that $0 \leq j \leq content2$ ..."
Because there is no *content*2 we apply that rule only to anyType-0.

```
anyType−0:
        AT(∗)                   anyType−0
        SE(∗)                   anyType−1
        EE
        CH                      anyType−1
```

```
                AT(xsi:type)            anyType−0
                AT(xsi:nil)             anyType−0
                AT(*)                   anyType−0
                AT(*)[untyped value]    anyType−0

anyType−1:
                SE(*)                   anyType−1
                EE
                CH                      anyType−1
```

After adding the NS and SC productions, we have the following.

```
anyType−0:
                AT(*)                   anyType−0
                SE(*)                   anyType−1
                EE
                CH                      anyType−1
                AT(xsi:type)            anyType−0
                AT(xsi:nil)             anyType−0
                AT(*)                   anyType−0
                AT(*)[untyped value]    anyType−0
                NS                      anyType−0
                SC Fragment

anyType−1:
                SE(*)                   anyType−1
                EE
                CH                      anyType−1
```

"Add the following productions to each nonterminal $Element_{i,j}$, such that $0 \leq j \leq content$."
becomes
"Add the following productions to each nonterminal $Element_{i,j}$, such that $0 \leq j \leq content2$."
The result of applying this rule is as follows.

```
anyType−0:
                AT(*)                   anyType−0
                SE(*)                   anyType−1
                EE
                CH                      anyType−1
                AT(xsi:type)            anyType−0
                AT(xsi:nil)             anyType−0
                AT(*)                   anyType−0
                AT(*)[untyped value]    anyType−0
                NS                      anyType−0
                SC Fragment
                SE(*)                   anyType−1
                CH[untyped value]       anyType−1

anyType−1:
                SE(*)                   anyType−1
                EE
                CH                      anyType−1
```

"Add the following productions to $Element_{i,content2}$ and to each nonterminal $Element_{i,j}$, such that $content < j < n$, where n is the number of nonterminals in $Element_i$."
becomes
"Add the following productions to $Element_{i,content}$ and to each nonterminal $Element_{i,j}$, such that $content2 < j < n$, where n is the number of nonterminals in $Element_i$."
The final grammar is as follows.

```
anyType−0:
        AT(∗)                    anyType−0
        SE(∗)                    anyType−1
        EE
        CH                       anyType−1
        AT(xsi:type)             anyType−0
        AT(xsi:nil)              anyType−0
        AT(∗)                    anyType−0
        AT(∗)[untyped value]     anyType−0
        NS                       anyType−0
        SC Fragment
        SE(∗)                    anyType−1
        CH[untyped value]        anyType−1

anyType−1:
        SE(∗)                    anyType−1
        EE
        CH                       anyType−1
        SE(∗)                    anyType−1
        CH[untyped value]        anyType−1
```

ACKNOWLEDGMENTS

The authors would like to thank Takuki Kamiya, Yusuke Doi, Daniel Peintner, and the other members of the W3C EXI working group for the fruitful discussions and valuable feedback. We are very grateful to the contributors and the users of the EXIP project for their feedback, code review, and source code contributions. We acknowledge the valuable discussions and support by the partners of the EU projects IMC-AESOP and Arrowhead.

REFERENCES

J. Arnoldus, M. van den Brand, A. Serebrenik, and J. J. Brunekreef. 2012. *Code Generation with Templates*. Atlantis Press.

J. Berstel and L. Boasson. 2000. XML grammars. In *Mathematical Foundations of Computer Science 2000*, Springer, 182–191.

G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. 2010. Learning deterministic regular expressions for the inference of schemas from XML Data. *ACM Trans. Web* 4, 4, Article 14, DOI:http://dx.doi.org/10.1145/1841909.1841911

G. Borriello and R. Want. 2000. Embedded computation meets the World Wide Web. *Commun. ACM* 43, 5, 59–66. DOI:http://dx.doi.org/10.1145/332833.332839

C. Bournez. 2009. Efficient XML interchange evaluation. Tech. Rep. W3C. http://www.w3.org/TR/exi-evaluation/.

A. Brüggemann-Klein and D. Wood. 2004. Balanced context-free grammars, hedge grammars and pushdown caterpillar automata. In *Extreme Markup Languages*.

G. Bumiller, L. Lampe, and H. Hrasnica. 2010. Power line communication networks for large-scale control and automation systems. *IEEE Commun. Mag.* 48, 4, 106–113. DOI:http://dx.doi.org/10.1109/MCOM.2010.5439083

D. Caputo, L. Mainetti, L. Patrono, and A. Vilei. 2012. Implementation of the EXI schema onwireless sensor nodes using Contiki. In *Proceedings of the 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. 770–774. DOI:http://dx.doi.org/10.1109/IMIS.2012.79

A. P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi. 2011. Web services for the internet of things through CoAP and EXI. In *Proceedings of the IEEE International Conference on Communications Workshops*. 1–6. DOI:http://dx.doi.org/10.1109/iccw.2011.5963563

A. Charland and B. Leroux. 2011. Mobile application development: web vs. native. *Commun. ACM* 54, 5, 49–53. DOI:http://dx.doi.org/10.1145/1941487.1941504

B. Chidlovskii. 2000. Using regular tree automata as XML schemas. In *Proceedings of the IEEE Computer Society Technical Committee on Digital Libraries*. IEEE, 89–98. DOI:http://dx.doi.org/10.1109/ADL.2000.848373

K. Chiu and W. Lu. 2004. A compiler-based approach to schema-specific XML parsing. In *Proceedings of the 1st International Workshop on High Performance XML Processing*.

J. Clark. 2013. Trang: Multi-format schema converter based on RELAX NG. http:relaxng.org.

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. Tree. http://www.grappa.univ-lille3.fr/tata.

J. Cowan and R. Tobin. 2004. XML Information Set (Second Edition). http://www.w3.org/TR/xml-infoset/.

Y. Doi, Y. Sato, M. Ishiyama, Y. Ohba, and K. Teramoto. 2012. XML-less EXI with code generation for integration of embedded devices in web based systems. In *Proceedings of the 3rd International Conference on the Internet of Things*. 76–83. DOI:http://dx.doi.org/10.1109/IOT.2012.6402307

S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. 2009. Smews: Smart and mobile embedded web server. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'09)*. 571–576. DOI:http://dx.doi.org/10.1109/CISIS.2009.29

J. Eliasson, P. Lindgren, and J. Delsing. 2008. A Bluetooth-based sensor node for low-power ad hoc networks. *J. Computers* 3, 1–10.

Y. Fablet and D. Peintner. 2014. Efficient XML interchange (EXI) profile for limiting usage of dynamic memory. http://www.w3.org/TR/exi-profile/.

R. T. Fielding and R. N. Taylor. 2002. Principled design of the modern Web architecture. *ACM Trans. Internet Technol.* 2, 115–150. DOI:http://dx.doi.org/10.1145/514183.514185

R. Gossweiler, C. McDonough, J. Lin, and R. Want. 2011. Argos: Building a web-centric application platform on top of Android. *IEEE Pervasive Comput.* 10, 4, 10–14. DOI:http://dx.doi.org/10.1109/MPRV.2011.64

S. A. Greibach. 1965. A new normal-form theorem for context-free phrase structure grammars. *J. ACM* 12, 1, 42–52. DOI:http://dx.doi.org/10.1145/321250.321254

K. Hartke. 2013. Observing Resources in CoAP. http://tools.ietf.org/html/draft-ietf-core-observe-08.

J. E. Hopcroft and J. D. Ullman. 1969. *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

D. A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proc. IRE* 40, 9, 1098–1101.

S. Käbisch, D. Peintner, J. Heuer, and H. Kosch. 2010. Efficient and flexible XML-based data-exchange in microcontroller-based sensor actor networks. In *Proceedings of the IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*. 508–513. DOI:http://dx.doi.org/10.1109/WAINA.2010.95

S. Käbisch, D. Peintner, J. Heuer, and Harald Kosch. 2011. Optimized XML-based Web service generation for service communication in restricted embedded environments. In *Proceedings of the 1 IEEE 16th Conference on Emerging Technologies Factory Automation*. 1–8. DOI:http://dx.doi.org/10.1109/ETFA.2011.6059002

T. Kamiya. 2013. OpenEXI. http://openexi.sourceforge.net/.

Q. Kang, H. He, and H. Wang. 2006. Study on embedded web server and realization. In *Proceedings of the 1st International Symposium on Pervasive Computing and Applications*. 675–678. DOI:http://dx.doi.org/10.1109/SPCA.2006.297507

A. J. Korenjak and J. E. Hopcroft. 1966. Simple deterministic languages. In *Conference Record of the 7th Annual Symposium on Switching and Automata Theory*. 36–46. DOI:http://dx.doi.org/10.1109/SWAT.1966.22

M. Kovatsch, S. Duquennoy, and A. Dunkels. 2011. A low-power CoAP for Contiki. In *Proceedings of the IEEE 8th International Conference on Mobile Adhoc and Sensor Systems*. IEEE, 855–860.

M. Kovatsch, M. Lanter, and Z. Shelby. 2014. Californium: Scalable cloud services for the internet of things. In *Proceedings of the 4th International Conference on the Internet of Things (IoT'14)*.

M. Kovatsch, S. Mayer, and B. Ostermaier. 2012. Moving application logic from the firmware to the cloud: Towards the thin server architecture for the internet of things. In *Proceedings of the 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS'12)*.

M. Kovatsch, M. Weiss, and D. Guinard. 2010. Embedding internet technology for home automation. In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*. 1–8. DOI:http://dx.doi.org/10.1109/ETFA.2010.5641208

K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg. 2011. Implementation of CoAP and its application in transport logistics. In *Proceedings of the Workshop on Extending the Internet to Low Power and Lossy Networks*.

R. Kyusakov. 2013. EXIP User Guide. Tech. Rep. EISLAB. http://exip.sourceforge.net/exip-user-guide.pdf.

R. Kyusakov. 2014. Efficient XML Interchange Processor. http://exip.sourceforge.net/.

R. Kyusakov, J. Eliasson, and J. Delsing. 2011b. Efficient structured data processing for web service enabled shop floor devices. In *Proceedings of the IEEE International Symposium on Industrial Electronics*. 1716–1721. DOI:http://dx.doi.org/10.1109/ISIE.2011.5984320

R. Kyusakov, J. Eliasson, J. van Deventer, J. Delsing, and R. Cragie. 2012. Emerging energy management standards and technologies: Challenges and application prospects. In *Proceedings of the IEEE 17th Conference on Emerging Technologies Factory Automation*. 1–8. DOI:http://dx.doi.org/10.1109/ETFA.2012.6489674

R. Kyusakov, H. Mäkitaavola, J. Delsing, and J. Eliasson. 2011a. Efficient XML interchange in factory automation systems. In *Proceedings of the 37th Annual Conference of the IEEE Industrial Electronics Society*. 4478–4483. DOI:http://dx.doi.org/10.1109/IECON.2011.6120046

G. Moritz, F. Golatowski, C. Lerche, and D. Timmermann. 2013. Beyond 6LoWPAN: Web services in wireless sensor networks. *IEEE Trans. Ind. Inf.* 9, 4, 1795–1805. DOI:http://dx.doi.org/10.1109/TII.2012.2198660

M. Murata, D. Lee, M. Mani, and K. Kawaguchi. 2005. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol.* 5, 4, 660–704.

F. Neven. 2002. Automata theory for XML researchers. *ACM Sigmod Record* 31, 3, 39–46.

D. Peintner. 2013. EXIficient. http://exificient.sourceforge.net/.

D. Peintner, H. Kosch, and J. Heuer. 2009. Efficient XML Interchange for rich internet applications. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME'09)*. 149–152. DOI:http://dx.doi.org/10.1109/ICME.2009.5202458

D. Peintner and S. Pericas-Geertsen. 2009. Efficient XML Interchange (EXI) Primer. Tech. Rep. W3C. http://www.w3.org/TR/2009/WD-exi-primer-20091208/.

A. Petrick and S. Van Ausdall. 2013. Smart Energy Profile 2.0. http://www.zigbee.org/Standards/ZigBeeSmartEnergy/Version20Documents.aspx.

P. Puñal Pereira, J. Eliasson, R. Kyusakov, J. Delsing, A. Raayatinezhad, and M. Johansson. 2013. Enabling Cloud Connectivity for Mobile Internet of Things Applications. In *Proceedings of the IEEE Seventh International Symposium on Service-Oriented System Engineering*. 518–526. DOI:http://dx.doi.org/10.1109/SOSE.2013.33

J. Schneider, T. Kamiya, D. Peintner, and R. Kyusakov. 2014. Efficient XML Interchange (EXI) Format 1.0. http://www.w3.org/TR/exi/.

Z. Shelby. 2010. Embedded web services. *IEEE Wirel. Commun.* 17, 6, 52–57. DOI:http://dx.doi.org/10.1109/MWC.2010.5675778

Z. Shelby, K. Hartke, and Bormann C. 2013a. Constrained Application Protocol (CoAP). http://tools.ietf.org/html/draft-ietf-core-coap-18.

Z. Shelby, S. Krco, and C. Bormann. 2013b. CoRE Resource Directory. http://tools.ietf.org/html/draft-ietf-core-resource-directory-00.

V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, and M. Welsh. 2004. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd International Conference on Embedded networked sensor systems (SenSys'04)*. ACM, New York, 188–200. DOI:http://dx.doi.org/10.1145/1031495.1031518

F. Simeoni, D. Lievens, R. Conn, and P. Mangh. 2003. Language bindings to XML. *IEEE Internet Comput.* 7, 1, 19–27. DOI:http://dx.doi.org/10.1109/MIC.2003.1167335

D. Sosnoski. 2003. XML and Java Technologies: Data binding. IBM developerWorksXML or Java Technology.

D. Sosnoski. 2014. JiBX: Binding XML to Java Code. http://jibx.sourceforge.net/.

V. Trifa, S. Wieland, D. Guinard, and T. M. Bohnert. 2009. Design and implementation of a gateway for web-based interaction and management of embedded devices. In *Proceedings of the 2nd International Workshop on Sensor Network Engineering (IWSNE'09)*.

R. Van Engelen. 2004a. Code generation techniques for developing light-weight XML Web services for embedded devices. In *Proceedings of the ACM Symposium on Applied Computing*. ACM, 854–861.

R. A. Van Engelen. 2004b. Constructing finite state automata for high performance web services. In *Proceedings of the IEEE International Conference on Web Services*.

R. A. van Engelen and K. A. Gallivany. 2002. The gSOAP toolkit for web services and peer-to-peer computing networks. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid*. 128.

P. Waher and Y. Doi. 2013. XEP-0322: Efficient XML Interchange (EXI) Format. http://xmpp.org/extensions/xep-0322.html.

G. White, J. Kangasharju, D. Brutzman, and S. Williams. 2007. Efficient XML Interchange Measurements Note. Tech. Rep. W3C. http://www.w3.org/TR/exi-measurements/.

D. Wood. 1995. Standard generalized markup language: Mathematical and philosophical issues. In *Computer Science Today*, Springer, 344–365.

Received October 2013; revised June 2014; accepted August 2014