Efficient Structured Data Processing for Web Service Enabled Shop Floor Devices

Rumen Kyusakov, Jens Eliasson and Jerker Delsing
Department of Computer Science, Electrical and Space Engineering
Luleå University of Technology S-97187 Luleå, Sweden

Abstract—**State of the art factory automation systems are now using Service Oriented Architecture (SOA) in order to increase flexibility and lower complexity of process monitoring and control. However, the service technology has not yet penetrated into the lower levels of plant-wide automation processes i.e. throughout shop floor devices such as programmable controllers, embedded sensors and actuators. Different techniques to adapt the web service technology to the specific requirements of embedded systems domain are intensively investigated by researchers and yet real-time properties, limited resources and wireless links are still posing immense challenges. The most promising solutions proposed are based on a newly emerging structured data format - Efficient XML Interchange (EXI). It is designed to compensate for the inefficiency of widely used throughout service implementations XML format. This paper investigates the design of EXI processor targeted at highly resource constrained embedded devices found at the shop floor level. The EXI processor is proposed as an alternative to the XML parsers and serializers currently used in web service implementations. Among the results presented are a novel low level processor interface and measurements quantifying the gained efficiency compared to traditional XML-like interfaces. Reference open source implementation equipped with the new interface is also provided.**

I. Introduction and Background

Today's global competitive environment provides new opportunities for manufacturing enterprises capable of keeping pace with the volatile markets and rapidly changing business demands. According to IBM 2008 and 2010 Global CEO studies [1], [2], the biggest challenges faced by manufacturing enterprises are the constant demands to change their processes and products and still be able to manage the inherent complexity in all levels of their production environment. In order to provide the IT support needed to cope with this challenges, a new way of designing automation software systems is required.

Service Oriented Architecture is seen as a promising approach to handle the complex interactions in highly heterogeneous and interdependent environment found in the process industry [3]. As a consequence, factory automation providers are now adopting SOA approach in their integrated solutions [4]. However, the currently used SOA implementations are based on web service technology which in turn relies on XML for communicating the service request and response messages. Using structured data formats like XML has the benefit of providing great flexibility to application developers and enhance interoperability but also brings high overhead in terms of memory, CPU, latency and power [5]. This makes the application of SOA on resource constrained devices problematic.

In order to take advantage of the SOA approach but avoid the unacceptable performance impact on the embedded devices, middleware software systems acting as a mediator between SOA capable enterprise systems and low-level networked embedded devices are employed. These systems integrate the shop floor devices such as supervisory control and data acquisition/distributed control systems (SCADA/DCS) with Manufacturing Execution Systems (MES), Enterprise Resource Planning (ERP), Enterprise Asset Management (EAM) systems etc. In such scenario, the expected gain from SOA deployment is limited to information access, system decomposition and application integration on enterprise system level but not on device level [6].

Different techniques to bring the XML-based service technology to highly resource constrained devices has been proposed. These include usage of compact tags and avoid SOAP binding for service messages [7] or the use of application specific XML parsers and serializers, transmission on the fly and data aggregation. Nevertheless, all these techniques come short to meet the real-time requirements and resource constraints for industrial machinery applications. Complementary to the aforementioned techniques, higher efficiency of SOA implementations can be achieved by using a binary structured data format instead of text-based XML. In order to preserve interoperability and flexibility and meet the resource constraints, it is required that the format is completely compatible with XML, provides higher processing efficiency and compactness and has smaller footprint implementation. The Efficient XML Interchange format [8] was developed by W3C Working Group to fulfill the above mentioned requirements and bring the benefits of XML to the resource constrained devices. On the other hand, the current EXI implementations are only supporting XML-like interfaces. This is necessary so that the legacy systems working with text-based XML will be able to use EXI without any modifications. However, we claim that for shop floor devices, where there are no legacy XML applications deployed, this is unnecessary and only decrease the EXI efficiency. The rest of this paper is devoted to analyzing the requirements for efficient EXI processor Application Programming Interface (API) for deeply embedded devices and propose an alternative to the currently used interfaces. The gained efficiency is also presented with concrete measurements.

The remainder of this paper is structured as follows - Section II presents the related work in the area. Section III discuss different implementation scenarios for EXI web services for shop floor devices. Section IV analyzes the EXI API requirements from embedded systems perspective and describes our low level interface. Section V summarizes the gained efficiency and shows performance measurements for our proposed solution. In Section VI, we give the possible improvements and extensions to our work, and Section VII concludes the paper.

II. Related work

Unifying the data exchange between different applications and systems under an open and standardized format could lead to huge savings on software development and maintenance especially for heterogeneous distributed systems. So far, XML has proven successful in wide range of applications due to its flexibility and is also established as a structured data carrier for most of the SOA implementations. This is the reason for having so many attempts to provide binary representation capable of bringing XML to embedded systems domain. The comparison between different binary structured data formats has been studied by both - researchers [9] and W3C binary XML working groups [10], [11]. The results from these analyses are giving preference to EXI over other formats as it provides higher compression ratio and faster processing that supports schema and schema-less encoding and decoding. The measurements are done on variety of test data sets covering different applications. A study which is specifically devoted to the use of binary XML for embedded web services is presented by Moritz et al. [12]. It shows the compactness of Devices Profile for Web Services (DPWS) [13] messages when different XML encodings are applied. Once again the most compact representation is achieved using EXI.

The aforementioned studies are concentrated on proving the applicability of Efficient XML Interchange to embedded systems domain and showing its superiority over other binary structured data formats. However, a detailed investigation on the optimal EXI processor interface has not been carried out yet. Similar studies for text-based XML, however, exist [14] and show the effects of different API on application programming and system maintainability. The analysis presented in this paper shows that for EXI, the design of the Application Programming Interface can have a significance impact on performance as well.

Figure 1 depicts the software components included in an EXI enabled web service implementation. As depicted, the network component provides an EXI encoded input/output using network API - which in many cases is TCP/IP Berkeley Software Distribution (BSD) socket API. Then the web service data model interacts with the EXI processor through its interface for encoding an EXI stream from data objects or decoding an EXI stream to data objects. In such way, the SOA application works on the data objects using the operations defined with Web Service Description Language (WSDL).

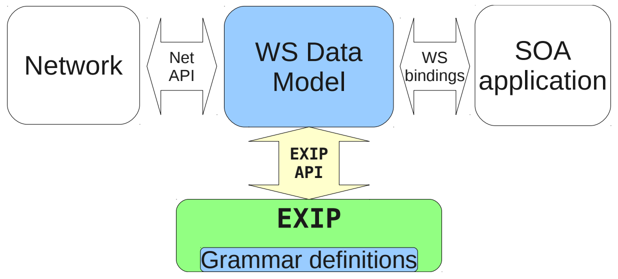

Fig. 1. Using EXI as a message carrier for web service implementations. The *WS Data Model* and *Grammar definitions* components can be automatically generated from the WSDL service definitions.

III. Efficient XML Interchange for web services

The bijection relationship between XML and EXI representations of the XML Information Set [15] provides for seamless integration of EXI with web service technology. In scenarios such as factory automation systems, where existing XML-based web services deployed on enterprise level systems must co-operate with EXI web services running on shop floor devices, the interoperability is from great importance. When implementing such a system, the following requirements must be taken into account:

1) The introduction of EXI services must be as transparent as possible. This means that modifications on existing systems should be avoided.
2) The complexity of the systems must not increase.
3) The overhead of deploying EXI services on networked embedded devices must be kept low. This implies that performance characteristics of the devices are preserved without upgrading their hardware.

Three implementation strategies can be identified. The first one is to use a gateway between embedded EXI services and enterprise services running on MES, ERP and EAM systems. The messages coming from one side of the gateway will be re-written and send to the other side in an appropriate representation - either binary or text-based. In this way, the modifications on the enterprise level are avoided. This approach however, introduces additional software and possibly hardware component to the system that will need additional maintenance efforts especially when new versions of the web service protocols are deployed. Moreover, it accounts for increased latency and must be redundant in order to avoid single point of failure.

The second strategy is to deploy XML parser and serializer on the shop floor devices in addition to the EXI ones. In this way, the device-to-device services will be executed over EXI and interaction between enterprise systems and embedded devices will be carried out using XML. This approach again does not require any modifications on the existing systems, but put additional overhead to resource constrained devices.

The third strategy is to plug additional EXI processor in the enterprise level service implementations. This modification can easily be implemented if the web service stack supports adding new transport protocols as plugins. Example of such stack is JAX-WS [16]. The main benefits of this approach are the optimal performance and low resource requirements for the embedded devices. The use of XML compliant EXI interface for the enterprise services is required in order to keep the modifications to a minimum. However, for embedded services the use of the same XML-like interface is unnecessary and only leads to degrading the device performance.

The next section discuss the requirements for efficient EXI interface for shop floor devices and gives overview of our approach.

IV. XML INFOSET PROCESSOR INTERFACES

There are two programming models for working with XML Information Sets: streaming and in-memory model. The later one stores the entire document tree and the complete infoset state in memory. This provides for easy to use and intuitive interface, but is not appropriate when working with large documents or memory constrained devices. Example of such interface is DOM (Document Object Model) [17]. On the other hand, in the streaming model, the XML infoset is processed in one direction from the start to the end of the document without preserving any state in memory. Instead, the streaming processor generates events when passing over information set objects in the document. This provides for small memory and processing footprint. As an example, Simple API for XML (SAX) [18] and Streaming API for XML (StAX) [19] are both streaming interfaces. The main difference between the two is the way the streaming events are delivered to the application. In the case of SAX, they are pushed to registered callback event-handling functions hence ”push parsing”. StAX however, uses the opposite approach - the application controls when the events will be fired i.e ”pulls” them from the parser.

As of the time of writing this paper there are only two EXI parser implementations. One is provided with commercial license - *Efficient XML* [20] and the other is open source - *EXIficient* [21]. As our attempt to receive evaluation copy of *Efficient XML* was not successful, our implementation is only evaluated against *EXIficient* processor which uses SAX and DOM interfaces.

A. Use cases

The requirements for our EXI API are derived from the target domain for our EXI implementation - resource constrained embedded devices. In such scenario the following features are of great importance:

1) Ability to turn on and off features depending on the needs of particular application i.e. application specific EXI processor
2) Support for web service code generation based on WSDL definitions as described by Kätibisch et al. [22]
3) Efficiency

Moreover, the observations of real-world data exchanged between shop floor devices, lead us to the following conclusions:

1) The amount of text data is very low as opposed to business domain transactions. The most prevalent is numeric data - integers and floating point numbers. However, raw binary data is also used in some occasions. As an example, besides the low level sensor data, the process monitoring can include CCD cameras for image capturing and subsequent processing for detecting defects in products or malfunctioning equipment.
2) Frequent exchange of small messages with strict timing constraints

B. Design for efficiency

The use of SAX or DOM for EXI processing has several advantages. First, any application working with XML data can start using binary EXI streams with little or no modifications. This means that the XML web service stacks in use today can be reused rather than replaced by new ones. Second, since both APIs are widely used, no new knowledge is required from programmers. Third, DOM is W3C standard and SAX is industry standard which leads to compatible and easier to maintain source code. Nevertheless, both SAX and DOM are designed to work on text-only data. When using them on top of an EXI processor, two extra transformation are needed - from native types to string data and then back from string data to native types. As an example, an EXI encoded unsigned integer [23] must be extracted as native type (32 bits or higher), casted to character string so that it conforms to the XML SAX or DOM API and then when the application starts working with the data it again must be casted from character string back to native unsigned integer representation. This is also the case for other data types. Another example is the raw binary data that require additional Base64 or HEX encoding and decoding. These transformations are unlikely to affect enterprise applications running on powerful servers with low real-time requirements. So in this scenario the benefits of using standard XML APIs outweigh the small performance cost. For programmable controllers, embedded sensors and actuators, however, this performance cost is too high, which requires new interface design meeting the requirements highlighted in our use cases - Subsection IV-A.

As our target platforms are memory and CPU constrained, the streaming programming model was selected for working with EXI encoded XML infoset objects. Both, pushing and pulling event processing styles have been investigated for applying to our interface. We have not found any differences between the two from performance perspective. However, they differ from usability point of view. Based on our personal impression from the experiments we did, the pushing streaming style was applied to EXI parsing while more control over streaming events is given for EXI serialization (pull streaming style).

The listings below show excerpts from our low level EXI interface. Full access to the source code is available from Efficient XML Interchange Processor (EXIP) project web page [24].

a) EXIP Serializer API: Encoding EXI streams

```
struct EXISerializer
{
  // For handling the meta-data (document structure)
  errorCode (*startDocumentSer)(EXIStream* strm);
  errorCode (*endDocumentSer)(EXIStream* strm);
  errorCode (*startElementSer)(EXIStream* strm,QName qname);
  errorCode (*endElementSer)(EXIStream* strm);
  errorCode (*attributeSer)(EXIStream* strm, QName qname);

  // For handling the data
  errorCode (*intDataSer)(EXIStream* strm, int32_t int_val);
  errorCode (*booleanDataSer)(EXIStream* strm,
       unsigned char bool_val);
  errorCode (*stringDataSer)(EXIStream* strm,
      const StringType str_val);
  errorCode (*floatDataSer)(EXIStream* strm,
      double float_val);
  errorCode (*binaryDataSer)(EXIStream* strm,
      const char* binary_val, size_t nbytes);
  errorCode (*dateTimeDataSer)(EXIStream* strm,
      struct tm dt_val, uint16_t presenceMask);
  errorCode (*decimalDataSer)(EXIStream* strm,
      decimal dec_val);

  // EXI specific
  errorCode (*exiHeaderSer)(EXIStream* strm,
      EXIheader* header);

  // EXIP specific
  errorCode (*initStream)(EXIStream* strm, char* buf,
      size_t bufSize, IOStream* ioStrm,
      struct EXIOptions* opts, ExipSchema* schema);
  errorCode (*closeEXIStream)(EXIStream* strm);
};
```

These functions are used to write to an EXI stream using native data types directly. The stream is constructed sequentially with immediate error reporting. As the pulling style is used, multiple streams can be created simultaneously in the same programming thread.

b) EXIP Parser API: Decoding EXI streams

```
struct ContentHandler
{
  // For handling the meta-data (document structure)
  char (*startDocument)(void* app_data);
  char (*endDocument)(void* app_data);
  char (*startElement)(QName qname, void* app_data);
  char (*endElement)(void* app_data);
  char (*attribute)(QName qname, void* app_data);

  // For handling the data
  char (*intData)(int32_t int_val, void* app_data);
  char (*booleanData)(unsigned char bool_val,
       void* app_data);
  char (*stringData)(const StringType str_val,
       void* app_data);
  char (*floatData)(double float_val, void* app_data);
  char (*binaryData)(const char* binary_val,
       size_t nbytes, void* app_data);
  char (*dateTimeData)(struct tm dt_val,
       uint16_t presenceMask, void* app_data);
  char (*decimalData)(decimal dec_val, void* app_data);

  // For error handling
  char (*warning)(const char code,
       const char* msg, void* app_data);
  char (*error)(const char code,
       const char* msg, void* app_data);
  char (*fatalError)(const char code,
       const char* msg, void* app_data);

  // EXI specific
  char (*exiHeader)(const EXIheader* header,
        void* app_data);
};
```

When decoding/parsing an EXI stream the events are pushed to the application. This requires the event-handler functions with signatures shown in the listing above be registered with the EXI processor. Error reporting is also delivered as an event. This API is very similar to SAX with the main difference being the use of native data types instead of character strings.

V. Performance measurements

EXIP interface evaluation: In this section we quantify the overhead of using text-based API, such as SAX or DOM, for an EXI processor as compared to our low-level interface on a resource constrained device. That is, we measure how much time it takes for the conversion of native data types to string data and then back to native data. The measurements are highly dependent on the hardware platform as well as the converting algorithms employed, the parameters passed to the compiler and the compiler itself. The testing platform for the experiments was Mulle sensor node [25]. It is equipped with a Renesas M16C/65 microcontroller running at 10 MHz with 47 kB RAM and 512 kB ROM. The Mulle sensor platform has Bluetooth or IEEE 802.15.4 radio transceiver. Numeric data conversions were implemented by using the standard functions provided with GNU C Library (glibc) - *sprintf()*, *atoi()*, *atol()* and *atof()*. As the only exception from that rule is when converting floating point numbers to character strings. In this case, the *sprintf()* function performance was very slow so we used our own implementation that executes more than five times faster. Binary data were character represented with Base64 encoding which creates 33% larger data block as compared to 50% larger when HEX encoding is applied. For the Base64 encoding/decoding routines, Bob Trower's open source implementation was employed [26]. The source code was built with GCC 4.4.1 cross-platform compiler for Renesas M16C microcontroller with no optimization parameters.

Native data type	Conversion time [ms]
Boolean	0.028
Signed 16 bit int	1.3
Unsigned 32 bit int	2.2
Floating point	11
Binary (1000 bytes of random data)	58

TABLE I
CONVERSION TIME FOR DIFFERENT DATA TYPES ON MULLE SENSOR PLATFORM

The results in Table I shows that for a Mulle sensor node to process an EXI encoded service request containing two signed 16 bits integers, a boolean and a floating point number it will take 13.628 ms longer if XML-like interface was used for the EXI parser running on Mulle. As another example, if a CCD camera is attached to the sensor node and a 10 kB JPEG image is sent from the sensor node to a processing unit using EXI web services, it will take 580 ms longer and will require additional memory when text-based EXI API is used. This is because the raw image received from the camera first needs to be Base64 encoded to fit into the SAX or DOM API and then in order for the EXI processor to encode the data in an EXI stream the Base64 encoded image must be decoded back to binary form.

EXIP implementation: Our EXI implementation is written in C and currently it only supports the default EXI encoding and decoding options. Hence, it cannot use schema to optimize the message size and the processing speed. Nevertheless, we performed some preliminary measurements to compare its performance to application specific XML parser available in gSOAP [27] that uses schema to predict the structure of the XML documents. We used a gSOAP port for our Mulle sensor platform and few simple web service messages. These XML messages were then converted to EXI representation using the default encoding options. Although, the size of the EXI messages is about 2 times smaller, the EXIP parser processing takes longer than the gSOAP parsing. This comes to show the importance of schema for optimizing performance, but also a number of areas for improvement in our processing algorithms employed in the implementation. Nevertheless, if we take into account the latency of a IEEE 802.15.4 radio-link induced by the larger messages our suboptimal EXI implementation is still providing better overall performance than text based XML and schema-enabled gSOAP parser.

As an example, for a 6LoWPAN (IEEE 802.15.4 radio) link, if we take an average TCP throughput of 15.2 kbits/s [28] and message size between 418 bytes and 2089 bytes (test set of DPWS messages [12]), the latency for a single hop transmission accounts for between 220 ms and 1099 ms respectively. While, when the same messages are binary encoded using default EXI options their size is between 234 bytes and 1118 bytes which leads to transmission time of 123 ms and 588 ms over 6LoWPAN TCP connection. When we took the smallest DPWS message the gain from EXI encoding is 97 ms which is much higher than the difference in processing between our EXIP parser and gSOAP which is around 40 ms for sample 415 bytes message. If we assume the presence of channel interference [29], then the TCP throughput is even lower and hence the benefits of our EXIP solution higher.

A more accurate statement of the current EXIP stack processing speed is shown in Table II. The measurements are acquired by setting up a test environment using two Mulle nodes with IEEE 802.15.4 radios, that run on TinyOS and use Berkeley Low-power IP stack (BLIP) as a 6LoWPAN network layer implementation. During the experiment, the first node requested a temperature readings from the second node. The request and response messages were formated according to the SOAP-over-UDP [30] standard and encoded and decoded in a binary EXI form in schema-less mode rather that plain XML.

Message	Parsing time [ms]	Serialization time [ms]
SOAP request (EXI schema-less) 451 bytes	182	252
SOAP response (EXI schema-less) 651 bytes	280	425

TABLE II
PROCESSING TIME FOR EXI ENCODED SOAP MESSAGES

VI. FUTURE WORK

Having efficient EXI processor API is not enough to guarantee the performance of web service executions required by programmable controllers, embedded sensors and actuators. Different parameters of the Efficient XML Interchange format can be used to further increase the compactness of the messages or even the processing efficiency. The most important are the use of schema-enabled processing and compression. Currently our EXI implementation only supports the default parameters, but providing support for all options as defined in the specification is in progress. Important area for improvement is the optimization of processing algorithms in our implementation.

A feature that can be exploited to increase efficiency is the use of Datatype Representation Maps. The application of this technique is described by Peintner et al. [31] where it is shown that in some scenarios it significantly increases the compactness of the messages.

VII. CONCLUSION

The introduction of Service Oriented Architecture in factory automation systems brings new opportunities but also new challenges. The web service standards adopted in business applications cannot meet the real-time requirements and resource constraints of the shop floor devices. The Efficient XML Interchange structured data format is seen as promising alternative to the text-based XML that could bring the efficiency needed by embedded systems. However, the early EXI implementations are not designed for deployment on highly resource constrained devices. In this paper we presented our EXI processor targeted at programmable controllers, embedded sensors and actuators. The measurements presented shows the performance gain achieved by using our low level processor interface on a small sensor node as compared to standard SAX or DOM solutions.

ACKNOWLEDGMENT

The authors would like to thank the European Commission and the partners of the EU FP7 project IMC-AESOP (www.imc-aesop.eu) for their support.

REFERENCES

[1] (2008) The enterprise of the future. IBM Global Business Service. [Online]. Available: http://www.ibm.com/ibm/ideasfromibm/us/ceo/20080505/

[2] (2010) Capitalizing on complexity. IBM Global Business Service. [Online]. Available: http://www-935.ibm.com/services/us/ceo/ceostudy2010/index.html

[3] R. Credle, V. Akibola, V. Karna, D. Pannerselvam, R. Pillai, and S. Prasad, *Discovering the Business Value Patterns of Chemical and Petroleum Integrated Information Framework*, I. Redbooks, Ed. IBM Redbooks publication, 2009, no. 0738433136. [Online]. Available: http://www.redbooks.ibm.com/abstracts/sg247735.html?Open

[4] S. Castro, "SAP Manufacturing Service Oriented Architecture (SOA) Current and Future Strategy," SAP Labs, LLC, Tech. Rep., 2009. [Online]. Available: http://www.sdn.sap.com/irj/scn/index?rid=/library/uuid/b0a3b05a-105f-2c10-8abe-a79220d8eb40

[5] C. Groba and S. Clarke, "Web services on embedded systems - a performance study," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, march 2010, pp. 726 –731.

[6] A. Duca, N. Freeman, and S. Drews, "SOA What? Demystifying SOA for the Process Industry," Honeywell International Inc, Tech. Rep., 2008. [Online]. Available: www.instrumentation.co.za/papers/C9206.pdf

[7] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, "Tiny web services: design and implementation of interoperable and evolvable sensor networks," in *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*. New York, NY, USA: ACM, 2008, pp. 253–266.

[8] *Efficient XML Interchange (EXI) Format 1.0*, W3C Std. [Online]. Available: http://www.w3.org/TR/2011/REC-exi-20110310/

[9] S. Sakr, "XML compression techniques: A survey and comparison," *J. Comput. Syst. Sci.*, vol. 75, pp. 303–322, August 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1530897.1531090

[10] (2005) XML Binary Characterization Working Group. W3C. [Online]. Available: http://www.w3.org/XML/Binary/

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

[12] G. Moritz, D. Timmermann, R. Stoll, and F. Golatowski, "Encoding and compression for the devices profile for web services," in *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*, 2010, pp. 514 –519.

[13] *Devices Profile for Web Services Version 1.1*, OASIS Std. [Online]. Available: http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.pdf

[14] F. Simeoni, D. Lievens, R. Conn, and P. Mangh, "Language bindings to XML," *Internet Computing, IEEE*, vol. 7, no. 1, pp. 19 – 27, 2003.

[15] J. Cowan and R. Tobin. XML Information Set (Second Edition). W3C. [Online]. Available: http://www.w3.org/TR/xml-infoset/

[16] J. Kotamraju, *The Java API for XML-Based Web Services (JAX-WS) 2.2*, Sun Microsystems, Inc. Std. [Online]. Available: http://jcp.org/en/jsr/detail?id=224

[17] (2010) Document Object Model (DOM). W3C. [Online]. Available: http://www.w3.org/DOM/

[18] (2010) Simple API for XML (SAX). [Online]. Available: http://www.saxproject.org/

[19] (2010) Streaming API for XML (StAX). Java Community Process. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr173/index.html

[20] Efficient XML. AgileDelta Inc. [Online]. Available: http://www.agiledelta.com/

[21] D. Peintner. (2010) EXIficient. [Online]. Available: http://exificient.sourceforge.net/

[22] S. Kändbisch, D. Peintner, J. Heuer, and H. Kosch, "XML-based Web service generation for microcontroller-based sensor actor networks," in *Factory Communication Systems (WFCS), 2010 8th IEEE International Workshop on*, May 2010, pp. 181 –184.

[23] (2010) EXI Unsigned Integer. [Online]. Available: http://www.w3.org/TR/2011/REC-exi-20110310/#encodingUnsignedInteger

[24] R. Kyusakov. (2010) Efficient XML Interchange Processor. LTU. [Online]. Available: http://exip.sourceforge.net/

[25] J. Johansson, M. Völker, J. Eliasson, Å. Östmark, P. Lindgren, and J. Delsing, "Mulle: A minimal sensor networking device - implementation and manufacturing challenges," in *Proceedings IMAPS Nordic*, 2004, pp. 265–271.

[26] B. Trower. (2010) Base64 implementation. [Online]. Available: http://base64.sourceforge.net/

[27] R. A. van Engelen and K. A. Gallivany, "The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks," in *2nd IEEE International Symposium on Cluster Computing and the Grid*, 2002. [Online]. Available: http://www.cs.fsu.edu/~engelen/ccgrid.pdf

[28] W. Guo, "Performance Analysis of IP over IEEE 802.15.4 Radio using 6LoWPAN," Washington University in St. Louis, Tech. Rep., 2008. [Online]. Available: http://www1.cse.wustl.edu/~jain/cse567-08/ftp/7lowpan.pdf

[29] J. Delsing, J. Eliasson, and V. Leijon, "Latency and packet loss of an interferred 802.15.4 channel in an industrial environment," in *Sensor Technologies and Applications (SENSORCOMM), 2010 Fourth International Conference on*, 2010, pp. 33 –38.

[30] *SOAP-over-UDP*, specs.xmlsoap.org Std. [Online]. Available: http://specs.xmlsoap.org/ws/2004/09/soap-over-udp/

[31] D. Peintner, H. Kosch, and J. Heuer, "Efficient XML Interchange for rich internet applications," in *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*, 282009-july3 2009, pp. 149 –152.