EFFICIENT FILTERING OF BINARY XML IN RESOURCE RESTRICTED EMBEDDED NETWORKS

Sebastian Käbisch[1,2], Richard Kuntschke[1], Jörg Heuer[1] and Harald Kosch[2]
[1]*Siemens AG, Corporate Technology, Communication Systems & Control Networks, 81730 Munich, Germany*
[2]*University of Passau, 94032 Passau, Germany*

Keywords: XML, EXI, Filtering, Embedded Networks.

Abstract: Existing XML-based filter and publish-subscribe solutions are based on plain-text XML. Due to the computational overhead and memory consumption of parsing and processing textual XML, these approaches are generally not applicable to embedded devices such as microcontrollers. However, XML-based communication in embedded networks is a desirable paradigm to ease the development of applications on top of diverse heterogeneous nodes by leveraging existing XML-based development processes and tools. In this paper, we present an approach using the W3C Efficient XML Interchange (EXI) format for efficiently filtering XML data against a number of XPath subscriptions with low computational effort and memory usage. Thus, XML-based messaging can be brought to resource limited embedded devices while at the same time gaining performance compared to technologies based on plain-text XML.

1 INTRODUCTION

Sensor/actor networks in the embedded domain, e.g., in home or industrial automation, can be very heterogeneous, containing wired and wireless nodes with different kinds of resources and service capabilities such as sensing, acting, and processing. This is especially true if the sensor/actor network evolves over time, adding new components or removing old ones. Since proprietary interfaces drastically limit the amount of available compatible devices or cause huge integration effort, research in recent years has focused on extending generic XML-based technologies such as Web services to the embedded domain.

By means of using XML-based communication, it is possible to create interoperable service communication between heterogeneous machines on today's Internet. However, textual XML comes with a huge penalty for parsing and processing XML data. Thus, binary XML has emerged and with the W3C Efficient XML interchange (EXI) format (Schneider and Kamiya, 2011), a binary XML standard has been established that enables very efficient usage and seamless adoption of XML-based protocols on embedded devices with limited resources such as microcontrollers (Käbisch et al., 2011). EXI eliminates the overhead of parsing and processing textual XML, reducing both, memory usage and computational effort to a degree that makes XML applicable in the embedded domain.

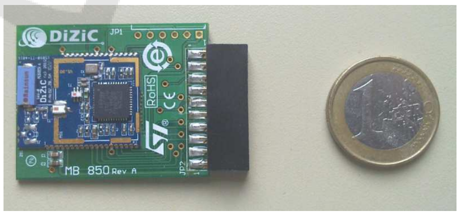

Figure 1: Typical microcontroller for embedded devices.

Figure 1 gives an impression of the kind of embedded devices that we target with the approach presented in this paper. The Figure shows an STMicroelectronics[1] ARM Cortex-M3 microcontroller with a 24 MHz CPU, 8 kilobytes of RAM, and 128 kilobytes of flash memory integrated into a typical interface board.

Figure 2 illustrates an example embedded network with eight embedded devices. The arrows indicate that data emitted by device 1, e.g., sensor readings, is routed to receiving devices 4, 5, and 7 via devices 2, 3, and 6, respectively. In such a scenario, the combined subscriptions of devices 4, 5, and 7 can already be evaluated at device 1. The resulting stream of data can then be routed to devices 4, 5, and 7 where it can

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

Käbisch S., Kuntschke R., Heuer J. and Kosch H..
EFFICIENT FILTERING OF BINARY XML IN RESOURCE RESTRICTED EMBEDDED NETWORKS.
DOI: 10.5220/0003939001740182
In *Proceedings of the 8th International Conference on Web Information Systems and Technologies* (WEBIST-2012), pages 174-182
ISBN: 978-989-8565-08-2

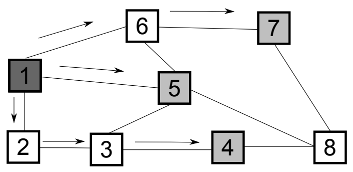

Figure 2: Service communication in embedded network.

again be processed to yield the actual data subscribed at the respective devices. In this paper, we focus on technologies for efficiently evaluating a batch of subscriptions against a potentially continuous stream of data at a single device such as, e.g., device 1 in the example above.

Our main contribution in this paper is the presentation of an approach using the W3C Efficient XML Interchange (EXI) format (Schneider and Kamiya, 2011) for efficiently filtering XML data against a number of XPath subscriptions with low computational effort and memory usage, thus making the approach applicable to resource limited embedded devices. In detail, we make the following contributions:

- We shortly introduce the W3C Efficient XML Interchange (EXI) format (Schneider and Kamiya, 2011) and describe how we realize an efficient EXI processor for embedded devices (Käbisch et al., 2010) (Section 2).
- We describe a naive approach for EXI-based XML filtering based on SAX events as a baseline and then go on to introduce in detail our solution for efficient XML filtering for embedded devices based on EXI (Section 3).
- Finally, we present performance results of our solution, comparing it to the naive approach as well as to YFilter (Diao and Franklin, 2003) as a representative of filtering based on textual XML (Section 4). Furthermore, a demo setup of an embedded network is presented that provides some numbers of memory usage of the approaches.

2 BINARY XML WITH EXI

The Efficient XML interchange (EXI) format (Schneider and Kamiya, 2011) is a very compact representation of the XML Information Set[2] (Cowan and Tobin, 2004) that is intended to simultaneously optimize performance and utilization of computational resources. Since March 2011 EXI is a W3C recommendation. In the following, a brief introduction of the functionality of EXI is given. The next subsection describes an approach how an efficient *EXI Processor* can be created that is suitable for small embedded devices.

[2]The XML Information Set is a W3C specification describing an abstract data model of an XML document in terms of a set of information items (e.g., elements and attributes). XML and EXI respectively are implementations thereof.

2.1 Basic Functionality of EXI

The EXI format uses a relatively simple grammar-driven approach that achieves very efficient encodings for a broad range of use cases (Peintner et al., 2009) (Bournez, 2009). The EXI specification defines a predefined process how schema information is to be transformed to EXI grammars. The reason for doing so is that EXI grammars are much simpler to process, compared to XML Schema information, while still describing in an accurate way what is expected to occur at any given point.

Figure 3 shows an example of an EXI grammar based on the XML schema shown in Listing 1. In general, EXI grammars correspond to deterministic finite automata (DFA) where each automaton represents a complex type in an XML schema. States represent a particular element. The transitions declare which successor states/elements may arrive. For states that allow multiple transitions, the grammar uses an event code (*EV*) to indicate which path in the automaton has been chosen.

Listing 1: XML Schema example snippet

```
<?xml version="1.0" encoding="UTF-8"?>
<schema>
    <element name="A">
        <complexType>
            <sequence>
                <element name="e" minOccurs="0"/>
                <element name="d"/>
            </sequence>
        </complexType>
    </element>
    <element name="B"> ... </element>
    <element name="C"> ... </element>
</schema>
```

The event code is represented with an *n*-bit stream ($n = \lceil log_2 m \rceil$, where *m* is the number of transitions at the state). E.g., the start state of the root grammar has three transitions. Thus, only 2-bits are required to signal *A* (=*EV(00)*), *B* (=*EV(01)*), or *C* (=*EV(10)*) as possible successor states. A transition is considered to be implicit if there is only one possible next state. In that case no event code is required (*EV(-)*).

To understand how such a grammar is used to encode and decode EXI streams, we discuss a simple example: Lets assume there is a plain XML document

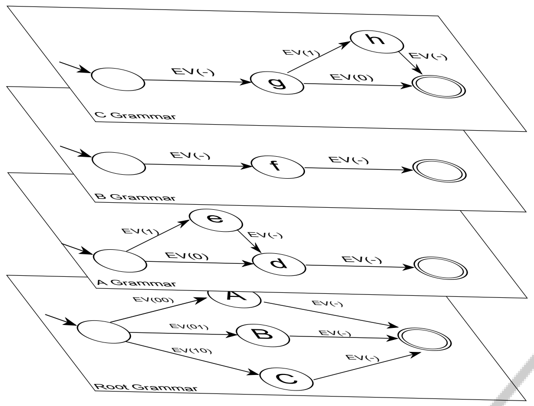

Figure 3: EXI grammar example.

as shown on the left hand side in Listing 2.

Listing 2: Two XML instances.

```
<A>                 <B>
  <e>123</e>          <f>abc</f>
  <d>abc</d>        </B>
</A>
```

For that case, the equivalent EXI stream would look like:

$EXI\_Stream_1 =$`00 1 '123' 'abc'`

This EXI stream is produced by traversing the corresponding transitions starting from the start state of the *Root Grammar*: The root element *A* in the XML document is met in this grammar by following the transition with the event code *00*. Signalling this, 2 bits are written to the stream and the *A* state is visited next. Since the *A* element is a complex type the *A Grammar* is processed. There, the event code *1* (1 bit) is written to the stream since the *e* element is present in the XML document. The *e* element itself is a simple type and at this point the value of the element is encoded (only sketched here). After that, no event code is written to the stream since there is no choice of transitions. Therefore, the *d* state is visited next without writing any bits ex ante. The *d* element is also a simple type and the value is encoded at this point. This will also close the stream since no event codes have to be written based on the grammar and no other element is present anymore in the example XML document. The decoding process is realized in the same manner. Only the EXI stream is read. Thereby, the event codes navigate through the grammar and for each simple typed state the decoding process is triggered.

The EXI stream for the XML instance on the right hand in Listing 2 is encoded in a similar way. The corresponding EXI stream has the following content:

$EXI\_Stream_2 =$`01 'abc'`

2.2 Efficient EXI Processor

In general, there is no prescribed nor a standardized way how an EXI Processor can be made aware of a certain set of EXI grammars. There are two apparent possibilities, but none of them seem to be suitable for restricted devices. Neither sharing the XML Schema documents itself nor pre-parsed grammar files are suitable. The requirement for a processor to provide all *possible* EXI functionalities, datatypes and such in each case (even if not required) is not feasible for microcontrollers. In (Käbisch et al., 2010) we present a solution to this problem. The main idea can be broken down into three automated processing steps:

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

II Based on the domain specific functionalities and datatypes the EXI grammar set *G* is generated.

III Those EXI grammars form the basis for creating the source code of an *EXI Processor* based on SAX eventing.

Summing up, we produce an *EXI Processor* with a SAX interface (startDocument, endDocument, startElement, endElement, etc.) that uses a minimal code footprint and complexity where the runnable code implicitly contains all required grammar information without any external dependencies. This resulting processing unit is able to encode the schema equivalent XML information to an EXI stream. Vice versa, an EXI stream or respectively XML information items can be decoded.

3 EFFICIENT FILTERING APPROACHES

In the following, we present two approaches to evaluate a number of given XPath requested (e.g., given by client nodes) on EXI streams. The goal is to get feedback if the present stream provides information that is requested and which queries have a match on the stream. In general, for both approaches we make the following assumptions:

- In this work, we focus on queries that are written in a subset of XPath, similar as described in (Diao and Franklin, 2003). That excludes, e.g., all dynamical operators that have to be executed at run time.
- We have the complete knowledge of the datamodel behind the XML-based messages which are evaluated by having the underlying XML

Schema definition. This enables a type aware predicate evaluation (e.g., $//e[text() > 100]$) and the immediate rejecting of XPath queries which can never be fulfilled.

- A XPath query which uses relative paths such as the descending-or-self axis ('//') and/or the wildcard ('*') operator is normalized based on the underlying XML Schema information (e.g., $//e[text() > 100] \rightarrow /A/e[text() > 100]$). This may result in one or more XPath queries when there are multiple paths to the addressed element.

We will start to introduce a basic approach that works on the top of an EXI grammar to evaluate XPath queries on binary XML streams. This approach will be called *BasicEXIFiltering*. A more sophisticated and optimized approach is presented in the next subsection that affects the EXI grammar directly. This approach is called *OptimizedEXIFiltering*.

3.1 Basic Binary XML Filtering

In general, this approach is based on the XPath evaluation concept with SAX events and follows the mechanism given by the the Knuth-Morris-Pratt algorithm (Cormen et al., 2001). Instead of the usage of a generic plain XML SAX parser we are using an *EXI Processor* with a SAX interface to read binary XML streams.

The XPath evaluation process is simple: Based on the SAX events that are released by reading the EXI stream we keep track of the current step *i* of each given XPath query. If an encountered *startElement* event matches the next step of one or more of the queries, the current step index of all affected XPath queries is incremented ($i = i+1$). If there is a mismatch, the mechanism will fall back to the last step in the corresponding query that matches the tags we have seen before and test again if a match was found. This process is continued for each XPath query until the last element was found (XPath match) or the *endDocument* event is reached which results to a no XPath query match.

In the following, an example is given based on the EXI grammar as shown in Figure 3. Thereby, the *EXI_Stream_1* (see section 2.1) is checked if the XPath query $/A/d$ matches:

EXI Stream	SAX Event	Current Step *i*
	startDocument()	0
00	startElement('A')	1
1	startElement('e')	1
'123'	character('123')	1
	endElement()	1
	startElement('d')	2
'abc'	character	2

Here, *i* is only incremented if a *startElement* provides the requested XPath node at the corresponding position. There is a match at the time $i = 2$ since two addressing elements are defined in the XPath query. Consequently, the result can be provided: *'abc'*.

Generally, this simple mechanism takes also predictions into account which are evaluated at the corresponding time when they occur. If one predicate evaluation is negative, this would result to a general query mismatch.

Since this *BasicEXIFiltering* is only seen as baseline we are not going to discuss it any further in detail in this paper and concentrate on the next presented binary XML filtering approach.

3.2 Optimized Binary XML Filtering

This proposed mechanism is realized by two main steps for determining the filter grammar and one optional code generation step. The details of each step are described separately in the following subsections.

3.2.1 Determine Accepting and Predicate States

For better clarification of the following processing steps lets consider the following 3 simple example queries that can successfully be applied on XML-based instances which are created based on the EXI grammar shown in Figure 3:

- $Q_1 = /C/h$
- $Q_2 = /A[e =' 123']/d$
- $Q_3 = //h$

Q_1 addresses the element *h* that is nested in *C*. Q_2 returns the value of element *d* if *e* is present and the value of *e* is 123 ($e =' 123'$). The last query selects all *h* elements of an XML-based document. Here, only one will be present, thus the result of Q_3 will be the same as that of Q_1.

Based on the set of queries, an analyzing step is performed to identify the so called *Accepting States (AS)* and *Predicate States (PS)* in the EXI Grammar. An AS represents the state or element that is requested by the query. A PS is a state where a predicate evaluation has to be done that is indicated by the query. Such kind of states can be found by following the addressed elements in each XPath query. E.g., Q_1 addresses at first the *C* element. In the EXI grammar the transition is followed that leads to the *C* state. As next the *h* element is searched within the *C* grammar. Since this is the requested element of Q_1 the *h* state is marked as *AS*. The general search mechanism in this process can be done by classical search algorithms such as Depth First Search (DFS) or Breadth First Search (BFS).

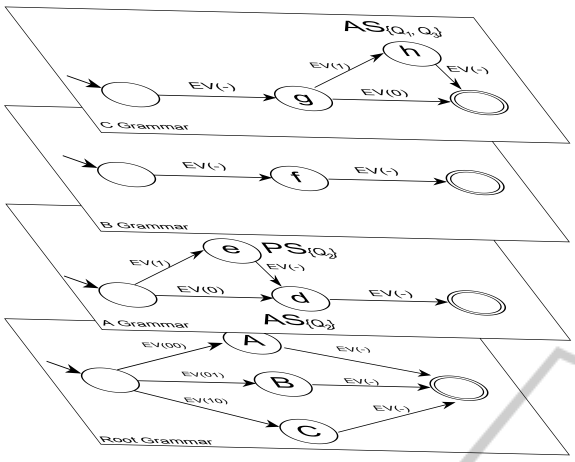

Figure 4: *AS* and *PS* in the grammar.

Query Q_2 contains a predicate. In that case, the corresponding *e* state is marked as *PS* which means a predicate evaluation has to be done at this point. Only if the predicate evaluation is positive then the requested *d* element is desired. The *d* state itself is also marked as *AS*.

The last query Q_3 uses the descending-or-self axis to find all *h* elements that occur in an XML-based instance. In such a case all paths that lead to an *h* state have to be identified. Since only one *h* state exists and we would prune out the descending operator the same result as Q_1 is expected.

Figure 4 shows the result of the identification process of *AS* and *PS* based on the given queries Q_1, Q_2, and Q_3.

3.2.2 Determine Filter Grammar G_f

After determining for each query the involved states in the EXI Grammar the filter grammar G_f is built. G_f is a subset of G and contains all necessary states and transitions for the given queries.

To create such a grammar the following steps are performed:

I Remove all states and transitions which are not required to reach *AS* and *PS*

II Remove all states and transitions that would skip a *PS* and would lead to an *AS* of the same query.

III *AS* will be an end state if there is no successor *AS* at this point

Since we are only interested in states that are required to evaluate each given query we remove all states and transitions which do not lead to at least an *AS* and a *PS*. If there are one or more predicates given in an XPath query the paths are removed which would skip the evaluation of such predicates. If we consider the query Q_2 and the *A* grammar (see Figure 4) it will explain this aspect: The start state of the *A* grammar allows two sucessor transitions, namely to the *d* ($EV(0)$) state and to the *e* ($EV(1)$) state. Q_2, however, calls only for *e* values where the predicate $e ='123'$ is true. EXI stream instances which do not address this condition by not containing the *e* value and using the $EV(0)$ transition at this point will not fulfil the query Q_2.

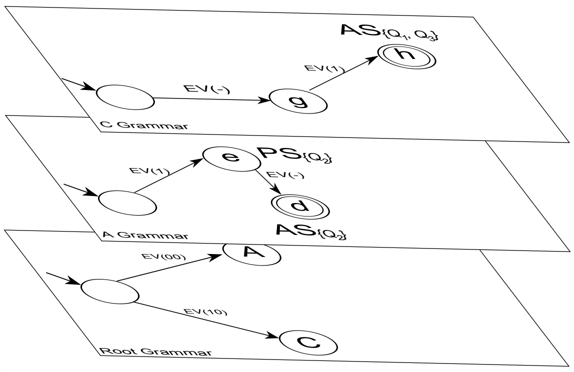

Figure 5: Filter grammar G_f.

Figure 5 shows the result of the elimination process which leads to the filter grammar G_f. It can be seen that the number of states and transitions has been reduced. E.g., all grammar components which are involved by and within *B* are removed. Thus, the complete *B* grammar is removed as well as the automaton fragments which lead to and from the *B* state in the *root* grammar.

In general, this grammar will only read an EXI stream that is really requested by one or more XPath queries. Applying the streams which was shown in section 2.1 $Stream_1$ would successfully traverse through the grammar and would match Q_2. However, $Stream_2$ would be discarded since the $EV(01)$ is not present in the *root* grammar of G_f. This shows the benefit that we are able to evaluate EXI streams at the very beginning stage of the decoding process if there is at least one query that matches this stream.

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

$$|G_f| \leq |G|$$

Thereby, the absolute value ($|..|$) gives the total number of used states and transition of the corresponding grammar. This equation is justified by the fact that G_f is a subset of G ($G_f \subseteq G$) and may contain as maximum all states and transitions when there is a demand of one or more XPath queries.

3.2.3 Generate Filter Code

Based on the filter Grammar G_f that contains the predicate evaluation functionality and the accepting state we are already able to use a generic EXI interpreter for the evaluation of an EXI message. However, as mentioned in section 2.2, such a solution is not suitable for small embedded devices because of the highly restricted memory and processing capacity.

Based on this motivation we extended our code generation tool as presented in section 2.2 for the filtering propose:

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

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

(b) For each query determine AS and PS of G (s. section 3.2.1)

(c) Build G_f by removing all states and transitions which do not lead to a AS or PS (s. section 3.2.2)

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

Mainly, step II and III are modified. The step II only integrates the mechanism as described in the former two subsections. The novelty is the step III where only the code is generated for only decoding EXI stream messages as well as the evaluation methods for the predictions. The motivation for not generating the code for encoding EXI stream is based on the simple fact, that we want to filter EXI streams to identify, if the requested information is present or not.

4 EXPERIMENTAL EVALUATION

In this section, we are going to evaluate the applicability of the presented binary XML filtering approaches. The evaluation section considers two aspects. First, the performance of the approaches is tested in general. To get an estimation how these approaches perform compared to an existing XML-based filtering mechanism, YFilter (Diao and Franklin, 2003) is involved in this test. The second aspect provides some numbers of code footprint and RAM usage size of a demo embedded network scenario that uses the binary XML filtering approaches.

Both approaches are implemented in the Java programming language and use the open source W3C EXI reference implementation[3]. For the described code generation mechanism we modified our existing implementation (see section 2.2) to realize the described filtering functionality in a code generated way. So far, the generator produces source code in the C and Java programming languages which can be used with platforms such as ContikiOS[4] and Java Micro Edition CLDC 1.1, respectively. The functionality to normalize XPathes if relative paths are used is given by the implementation of XPathOverSchema[5] library.

4.1 Performance

To evaluate the performance of both approaches we also involved the performance results of an implementation of YFilter [6] for the same data set. YFilter is one of the well known fast approaches for XML filtering that is based on non-deterministic finite automata (NFA). The XML-based documents which are used for the evaluation are based on an embedded device profile. That includes information such as addressing, status of device, time of data, temperature (with different scales), humidity, voltage, and status of the LEDs. The example document has a size of 875 bytes. Based on the underlying XML Schema of this device profile we serialized the document in binary XML by means of EXI that produces a size of 13 bytes. Since we are not able to process plain XML on small embedded devices such as microcontrollers, especially not the YFilter algorithm, these performance experiments were conducted on an Intel Core 2 Duo with 2.10GHz and 3GB RAM.

To avoid the overhead of the result collection process in our measurement we disabled that in our approaches and in the YFilter (this was done by the commands: *–result=NONE* and *–nfa_opt=0*). However, to have a fair comparison the parsing time of the XML-based documents is always included in the measurement results. This is based on the fact that - regarding our presented approaches - there is no preprocessing of the present binary XML stream and the XPath query evaluation is done at the same time as the binary XML stream is parsed.

In general, to have a very good shape the average time of 1000 rounds is determined. The XPath sets are based on queries with different kinds of requests of device status, values, times, and addressing. In the

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

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

[5]http://xpath-on-schema.sourceforge.net/

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

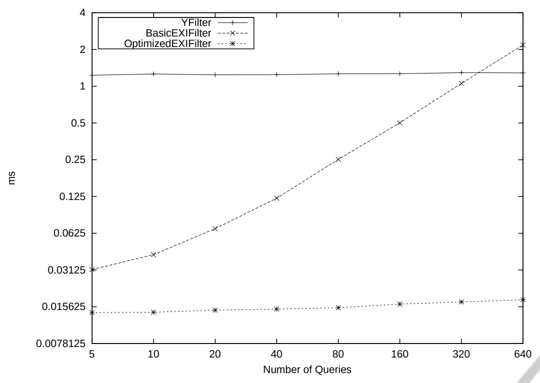

Figure 6: Filtering performance test based on an 875 bytes plain XML document (YFilter) and the equivalent 13 bytes binary XML representation (EXI filter approaches).

context of this paper the number of executed XPath query sets is small and more oriented at the usage in the embedded domain.

Figure 6 shows the result of our performance experiments with 8 different XPath query sets in milliseconds (with logarithmic scaling). Considering the first query sets both binary XML filtering approaches always performs much better as the YFilter implementation. In the case of 5 queries the *BasicEXIFiltering* approach is 40 times faster and the *OptimizedEXIFiltering* approach is 70 times faster. This shows the benefit of the opportunity to operate directly on the binary XML document without the need to transform it to a plain text representation. Furthermore, we are able to evaluate the XPath queries at the same time of this decoding process. YFilter separates this process (XML document parsing and XPath evaluation by the constructed NFA) which results to slower performances.

The *BasicEXIFiltering*, however, losses in performance exponentially when the number of XPath queries increases. This is due to the fact, that for each start element or attribute SAX event that occurs during the decoding process all queries are checked if the current step index can be incremented or not. This processing overhead is getting dominant and results in lower performance when the query set is getting larger. Here, at the time of around > 320 queries the *BasicEXIFiltering* starts to perform slower compared to YFilter.

Comparing the *OptimizedEXIFiltering* with the YFilter one sees that both perform almost constantly in their time level, even the number of queries increases. This is explained by the fact that all XPath queries are represented as automata, and duplicated queries (these occurrences arise if the number of query sets increases) do not affect the automata size and thus the automata processing. Only the query registration for the predicate evaluation and the accepting

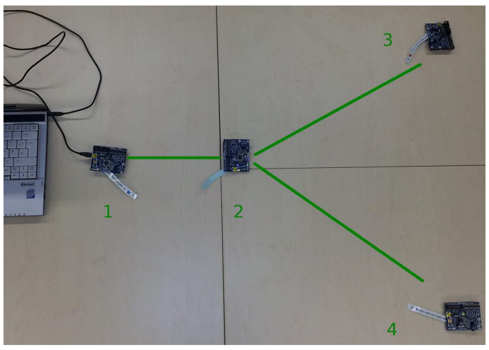

Figure 7: Demo embedded network.

state is required.

4.2 Demo Network

To evaluate the applicability of the binary XML filtering approaches for the embedded domain we set up a small embedded network with four wireless battery powered evaluation boards from STmicroelectronics that embeds the ARM Cortex-M3 microcontroller (24 MHz CPU, 8 kilobytes of RAM, and 128 kilobytes of flash) as presented in the introduction section. Our demo network and its topology can be seen in Figure 7.

Each node running the ContikiOS and the communication is based on IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) (Montenegro et al., 2007). Node 1 provides (in 1 seconds tact) its device profile information represented as binary XML format (compare previous subsection 4.1). For this test, some values of the device profile are randomized. Node 3 and 4 are the subscriber of node 1. However, node 3 wants only this information if the following query is fulfilled:

$$//Temperature[@Scale = 'C']/Value[text() > 27.2]$$

The precondition of node 4 is given by:

$$//StatusOfMote[text() = 'ErrorOccured']$$

Based on the XML Schema definition of the device profile and the provided XPath queries we created our binary XML filter mechanisms (*BasicEXIFiltering* and *OptimizedEXIFiltering*) with the code generation variants. In our network node 2 runs the generated filter. There, the implementation is realized in such a way that it receives the device profile documents from node 1, evaluates the content by using the filter, and only forwards the message to the corresponding node (3 and/or 4) if there is a match.

The demo could be successfully executed. At runtime we used a laptop that is connected to node 1 to monitor the device profile values which are sent to node 2. If there is a match a LED is showing that event. Also a LED is switching on on the nodes 3 and 4 if they received a message that was requested.

Table 1 gives an overview of the automata size and memory usage of the filtering set-up on node 2 for the cases the *BasicEXIFiltering* or *OptimizedEXIFiltering* implementation was used. It can be seen that the grammar size (transitions and states) of the *OptimizedEXIFiltering* is much smaller compared to the *BasicEXIFiltering* approach. This is due to the fact that the *BasicEXIFiltering* operates on the top of the full EXI decoder grammar (compare section 3.1). Meanwhile, the *OptimizedEXIFiltering* operates directly within the grammar and states and the transitions are removed which are not required to evaluate the XPath queries. Consequently, the ROM exhibits a smaller size since the generated code does not contain this grammar information. The numbers for this use case are not impressive since the generic EXI decoder mechanism dominates the grammar implementation part.

For the *BasicEXIFiltering* variant, besides the predicate data structure, an extra data structure is required that represents all XPath queries with a tracking position. Such a kind of data structure is not required for the *OptimizedEXIFiltering* since the filter grammar is already a representation of all XPath queries. Thus, the RAM usage shows better results compared to the *BasicEXIFiltering*. Since only two XPath queries are used the difference is relatively small. It will increase, however, when more XPath queries are taken into account.

Table 1: Grammar size (number of states / transitions) and memory usage (in bytes).

	BasicEXIFilt.	OptEXIFilt.
Grammar Size	31 / 45	21 / 22
ROM	6280	6104
RAM	580	412

5 RELATED WORK

Binary XML has been a research topic for a long time and related projects started shortly after XML had been introduced. These efforts were primarily driven by the desire to employ well-established and interoperable XML technologies on devices with limited resources such as, e.g., cell phones and cameras. More sophisticated schema-based techniques followed shortly after, examples of which include BiM (Heuer et al., 2002), FastInfoset (Sandoz et al., 2004) and ASN.1 (ITU, 2002). However, these technologies never achieved broad acceptance in the XML community. This was mainly due to the fact that many solutions were proprietary and/or tailored to a particular application domain. These approaches were thus not generic and lacked the necessary flexibility. In addition, some had the potential issue of license fees compared to a royalty free W3C standard. The W3C Efficient XML Interchange (EXI) format (Schneider and Kamiya, 2011) resolves these issues by defining a generic, flexible, efficient, royalty free W3C Binary XML standard that fits seamlessly into the portfolio of existing W3C XML standards.

Efficient filtering, processing, and dissemination of data has been an area of active research for many years. This includes work on data management in embedded networks, such as TinyDB (Madden et al., 2005). Sinced TinyDB is based on SQL, it is focused on structured data. With the advent of semi-structured data models, especially XML, efforts in the direction of XML filtering, XML transformation, and Internet-scale XML-based publish/subscribe systems emerged. These include, among others, XFilter (Altinel and Franklin, 2000), YFilter (Diao and Franklin, 2003), and ONYX (Diao et al., 2004). Since these solutions aim at processing textual XML, they require systems with adequate amounts of memory and processing power. Embedded devices with extremely limited resources such as the one introduced in Chapter 1 are unable to use such approaches.

Another field of related work are data stream management systems (DSMSs) such as Aurora (Abadi et al., 2003), Borealis (Abadi et al., 2005), TelegraphCQ (Chandrasekaran et al., 2003), and StreamGlobe (Kuntschke et al., 2005). The main focus of such systems is on efficient processing of potentially infinite data streams against a set of continuous queries. In contrast to publish/subscribe systems, continuous queries in DSMSs can be far more complex than simple filter subscriptions. Also, in distributed DSMSs such as StreamGlobe and Borealis, network-aware stream processing and operator placement are important issues. These are also relevant issues in distributed embedded networks such as the one illustrated in Figure 2. However, these questions are not in the focus of this paper which deals with solutions for efficient filtering of binary XML at a single node. Most DSMSs, such as TelegraphCQ for example, are based on relational data. StreamGlobe, however, focuses on XML data streams. Today, there are no DSMSs that are based on binary XML. Consequently, the nodes used for distributed data stream processing in systems such as StreamGlobe and Borealis generally need to be far more powerful than the microcontrollers for embedded devices we aim at in this paper. Solutions for direct processing of binary XML as proposed in this paper help to bring XML-based publish/subscribe and data stream management systems to the embedded domain.

6 CONCLUSIONS AND OUTLOOK

In this paper, we proposed mechanisms to enable XML-based filtering for resource restricted embedded networks. Both approaches use the W3C EXI format to evaluate XPath queries. The *BasicEXIFilter* approach works on the top of the EXI grammar, meanwhile, the more sophisticated *OptimizedEXIFilter* approach maps all XPath queries directly in the EXI grammar and removes all states and transitions which are not required for the evaluation. This results in a high performance filtering processor with very low resource usage that makes it also applicable on small embedded devices such as microcontrollers.

Topics for future work include the further optimization of the presented approaches as well as the development of publish-subscribe and optimized data dissemination systems for service communication within embedded networks. Especially, the challenge of updating XPath query sets at runtime is an important issue for further work.

ACKNOWLEDGEMENTS

We would like to thank Daniel Peintner for all the advices corresponding to EXI. Furthermore, we would like to thank Li Chen for the inspiring discussions and great support of this topic in general.

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. B. (2005). The design of the borealis stream processing engine. In *CIDR*, pages 277–289.

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. B. (2003). Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139.

Altinel, M. and Franklin, M. J. (2000). Efficient filtering of xml documents for selective dissemination of information. In Abbadi, A. E., Brodie, M. L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., and Whang, K.-Y., editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 53–64. Morgan Kaufmann.

Bournez, C. (2009). Efficient XML Interchange Evaluation. http://www.w3.org/TR/exi-evaluation/. W3C Working Draft 7 April 2009.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. (2003). Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*.

Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

Cowan, J. and Tobin, R. (2004). XML Information Set (Second Edition). http://www.w3.org/TR/xml-infoset/. W3C Recommendation 4 February 2004.

Diao, Y. and Franklin, M. J. (2003). High-performance xml filtering: An overview of yfilter. *IEEE Data Eng. Bull.*, 26(1):41–48.

Diao, Y., Rizvi, S., and Franklin, M. J. (2004). Towards an internet-scale xml dissemination service. In Nascimento, M. A., Özsu, M. T., Kossmann, D., Miller, R. J., Blakeley, J. A., and Schiefer, K. B., editors, *VLDB*, pages 612–623. Morgan Kaufmann.

Heuer, J., Thienot, C., and Wollborn, M. (2002). *MPEG-7 Binary Format*, chapter 2.3. MPEG-7. Introduction to MPEG-7: Multimedia Content Description Interface.

ITU, T. S. S. (2002). Abstract Syntax Notation One (ASN.1) Specification of Basic Notation. ITU-T Rec. X.680.

Käbisch, S., Peintner, D., Heuer, J., and Kosch, H. (2010). Efficient and Flexible XML-based Data-Exchange in Microcontroller-based Sensor Actor Networks. In *5th International IEEE SOCNE Workshop on Service Oriented Architectures in Converging Networked Environments*.

Käbisch, S., Peintner, D., Heuer, J., and Kosch, H. (2011). Optimized XML-based Web Service Generation for Service Communication in Restricted Embedded Environments. In *16th IEEE International Conference on Emerging Technologies and Factory Automation*.

Kuntschke, R., Stegmaier, B., Kemper, A., and Reiser, A. (2005). Streamglobe: Processing and sharing data streams in grid-based p2p infrastructures. In Böhm, K., Jensen, C. S., Haas, L. M., Kersten, M. L., Larson, P.-Å., and Ooi, B. C., editors, *VLDB*, pages 1259–1262. ACM.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2005). Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173.

Montenegro, G., Kushalnagar, N., Hui, J., and Culler, D. (2007). Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard).

Peintner, D., Kosch, H., and Heuer, J. (2009). Efficient XML interchange for rich internet applications. In *Multimedia and Expo, 2009. ICME 2009*, pages 149–152.

Sandoz, P., Triglia, A., and Pericas-Geertsen, S. (2004). Fast Infoset. On Sun Developer Network.

Schneider, J. and Kamiya, T. (2011). Efficient XML Interchange (EXI) Format 1.0. http://www.w3.org/TR/exi. W3C Recommendation 10 March 2011.