Efficient Management of Data Models in Constrained Systems by Using Templates and Context Based Compression

Jorge Berzosa[1(✉)], Luis Gardeazabal[2], and Roberto Cortiñas[2]

[1] IK4-Tekniker, Eibar, Spain
jorge.berzosa@tekniker.es
[2] University of the Basque Country UPV/EHU, San Sebastián, Spain
{pedrojoseluis.gardeazabal,roberto.cortinas}@ehu.eus

Abstract. Data communication is at the heart of any distributed system. The adoption of generic data formats such as XML or JSON eases the exchange of information and interoperability among heterogeneous systems. However, the verbosity of those generic data formats usually requires system resources that might not be available in resource-constrained systems, e.g., embedded systems and those devices which are being integrated into the so-called IoT. In this work we present a method to reduce the cost of managing data models like XML or JSON by using templates and context based compression. We also provide a brief evaluation and comparison as a benchmark with current implementations of W3C's Efficient XML Interchange (EXI) processor. Although the method described in this paper is still at its initial stage, it outperforms the EXI implementations in terms of memory usage and speed, while keeping similar compression rates. As a consequence, we believe that our approach fits better for constrained systems.

Keywords: IoT · XML · JSON · Template · Context · Compression · EXI

1 Introduction

The current trend to integrate heterogeneous systems into a big network like the Internet of Things (IoT) demands interoperable communication and data models. Since many systems are composed of resource-constrained devices, a big effort is being done to provide those systems with protocols and tools adapted to their limitations. The general approach is to tackle the challenge at different layers. For example, we can find IEEE 802.15.4 [6] for the media access control layer, 6LoWPAN [10] in the case of the network layer and the Constrained Application Protocol (CoAP) [11] at the application layer.

This work considers the representation of data, which could be located at the presentation layer. Data can be represented by using many different formats. In this work we will focus on semi-structured data models and, more precisely,

C.R. García et al. (Eds.): UCAmI 2016, Part II, LNCS 10070, pp. 332–343, 2016.
DOI: 10.1007/978-3-319-48799-1_38

W3C's XML (Extensible Markup Language) as a main reference, even though we could also consider other options such as JSON (JavaScript Object Notation). XML is widely extended and is the basis for many web services and related protocols, e.g., SOAP [5]. Roughly speaking, XML has been designed to be human readable, with tokens and attributes codified as strings. Nevertheless, this makes XML too large to be efficiently managed by resource-constrained devices like embedded systems. Additionally, parsers need to deal with large amount of string data, thus involving too much processing for energy and processor constrained devices.

The efficient management of formatted data models would ease the native use of Web Services and high level protocols and data models in resource-limited IoT devices. This would enhance the application of self-* services family (self-discovery, self-configuring, etc.) as well as overall interoperability. The benefits for all IoT domains in general are clear but they are specially important for consumer electronics targeted domains (domotics, entertainment, etc.) where improved interoperability across services/vendors and increasing complexity and richness of the interactions with the "smart" devices/environment would enhance user experience.

Efficient XML Interchange, EXI [9], adopted as a recommendation by the World Wide Web Consortium (W3C), relies on a binary representation of XML Information Set. It is designed to provide a considerable reduction on the size of the information in XML format (70–80 % as shown in [12]) and a high performance when encoding/decoding (6.7 times faster decoding and 2.4 times faster encoding according to [4]). In EXI a XML document is represented by an EXI stream, which is composed of a header (containing encoding information) and a body (representing the data). Data are represented based on formal grammars to model redundancy. Interestingly, when the schema of the XML is available, EXI achieves better results.

In this work we propose an approach based on templates. Roughly speaking, templates are extracted from managed data model schema documents so that their representation can be replaced in the data model instance documents with a minimum number of references. The documents are then compressed (by using lossless compression) following a context based grammar. The templates are registered at running time and made available to the rest of the system. In order to manage the data model documents, nodes will go through a preliminary discovery phase, where required templates and identifiers are downloaded from its location.

Outline. The paper is structured as follows; Sect. 2 presents the basis for template based compression. Then, Sect. 3 discuss about the dissemination of the templates in the system. A brief performance study compared to EXI is presented in Sect. 4. Finally, Sect. 5 presents the conclusions of the paper and future work.

2 Template Based Compression

Formatted data models can be represented by a tree graph encapsulating the links between the elements and the attributes of those links (such as the cardinality of the children showed in Fig. 1). In this work we define a grammar that is able to describe the links between the elements and templates that compose a formatted data model as well as the rules to follow in order to apply the grammar for efficient encoding and decoding. The proposed method is intended to be generic and not tied to a specific data description format (such as XML or JSON).

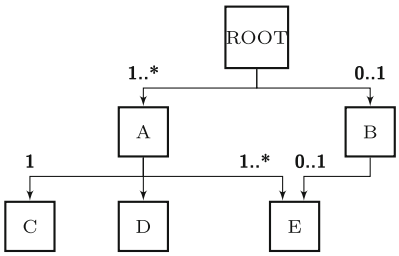

Fig. 1. Formatted data model tree example. The numbers in the links denote the cardinality: "1" one child, "1..*" one to many children, "0..1" none or one child (optional).

Each data model is described by a specific schema. This schema is used to extract the generic tree graph that is independent of the schema's original representation format. The tree graph is represented by a *Schema Context* that contains all the relevant schema information including elements and links. The generic grammar is used to process the *Schema Context* and execute the encoding and decoding processes. A set of generic rules is defined to map a schema into a *Schema Context*. However, these rules have to be specifically implemented for each data format type as the mapping of the schema to a *Schema Context* is data format specific.

2.1 Context Table and Schema Context

A *Context Table* contains all the information of the data model schemas known to the node. Each entry of the *Context Table* is a *Schema Context* that contains the information related to the elements and templates used in the schema, links between elements and, in summary, all the information needed to process a data model described by the schema. Figure 2 shows a detail of the Context Table of the example shown in Fig. 1.

A *Schema Context* is identified by the *URI* and *SchemaId* attributes. The *URI* attribute must be unique and is used to globally identify the *Schema Context*. The *SchemaId* attribute is assigned in the node bootstrapping phase (as

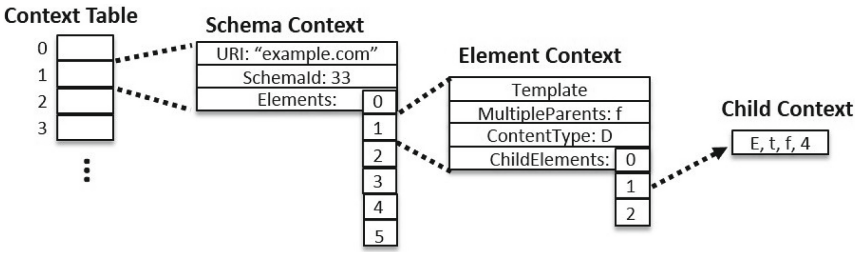

Fig. 2. Example context table detail.

described later) and must be unique within the (sub-)network the *Schema Context* is used.

A *Schema Context* contains the list of *Element Contexts* that describe the elements of the schema. An *Element Context* has the following attributes:

- *Id*: the identifier of the element in the schema, which is denoted by its index in the *Schema Context* element list.
- *Template*: a reference to the entry in the *Template Table* that contains the template for this element.
- *MultipleParents*: *true* if the element is a child of more than one parent. *false* otherwise.
- *ContentType*: the value of this attribute depends on the order the children may appear in a data model instance. If the order of the children is fixed and coincides with the order in which they are defined in the schema, the value is *fixed*. If the order is random, the value is *dynamic*. Finally, if only one single children can appear (among all the ones defined in the schema), the value is *choice*.
- *ChildElements*: it contains the *Child Context* list.

Finally, *Child Context* is a tuple composed of the following attributes:

- *Type*: data type of the element. The data type can be either a basic type or an *element* type. For the basic data type case the following types (inherited from the EXI [9] specification) are supported: *binary*, *boolean*, *decimal*, *float*, *integer*, *date-time* and *string*.
- *IsOptional*: *true* in case the cardinality of the child is $0..m$, where $m > 0$, and *false* otherwise.
- *IsArray*: *true* in case the child is an array, i.e., those children which have cardinality $n..m$, where $m > n \geq 0$.
- *ElementId*: if the *Type* attribute is *element*, *ElementId* contains the *Id* (i.e., the index) of this child's *Element Context*.

Table 1 shows the Element Contexts associated to the data model of the example in Fig. 1.

Table 1. *Element Contexts* example. Each column represents an Element Context. The Type of all *Child Contexts* is *element*. The content of the *ChildElements* row represents the tuple (*IsOptional*, *IsArray*, *Id*). *F* denotes *fixed*, *D dynamic*, *f false* and *t true*.

Atribute	Id					
	root(0)	A(1)	B(2)	C(3)	D(4)	E(5)
MultipleParents	f	f	f	f	f	t
ContentType	F	D	F	F	F	F
ChildElements	(f,t,1) (t,f,2)	(f,f,3) (t,f,4) (f,t,5)	(t,f,5)	()	()	()

2.2 Template Table

The *Template Table* contains the list of templates known by the node. Basically, templates are represented by using a custom format. They are strings which use the `@` symbol to denote place-holders. Each place-holder represents a child of the element represented by the template.

In addition to the information about the template, each row also contains context information in order to ease the matching between the original format and the template, thus, improving and optimizing the searching and codification processes.

2.3 Template Grammar

The *Template Grammar* allows processing data model elements by using the information available in the *Context Table*.

Figure 3 shows the grammar, where $\text{Child}_0 \ \ldots \text{Child}_{k-1} \in C$, being C the group of children of an element and k the number of children of the element. The index i, $0 \leqslant i < k$, denotes any children of C.

```
Element ::= FixedChildren
  | DynamicChildren
  | ChoiceChildren

FixedChildren ::= Child_0 ... Child_{k-1}

DynamicChildren ::= ϵ
  | Child_i DynamicChilds

ChoiceChildren ::= Child_0
  | ...
  | Child_{k-1}

Child ::= ϵ
  | ChildElement Child
```

```
ChildElement ::= binary
  | boolean
  | decimal
  | float
  | integer
  | date-time
  | string
  | Element
```

Fig. 3. Template grammar. k denotes the number of children ($0 \leqslant i < k$).

The production used for the left-hand *Element* depends on the value of the attribute *ContentType*. If the value is *fixed*, the right-hand production *FixedChildren* is used. If instead the value is *dynamic*, the right-hand production used will be *DynamicChildren*. Finally, if the value of *ContentType* is *choice*, the right-hand production *ChoiceChildren* will be used.

The production for left-hand *Child* depends on the value of *IsOptional* and *IsArray* attributes. In case both are *false*, only "ChildElement ϵ" is accepted. The *root* element always should have *IsOptional* and *isArray* attributes to *false*.

Finally, the left-hand *ChildElement* contains the terminal symbols of the basic types as well as the non-terminal symbol *Element*.

2.4 Context Table and Template Table Creation

As explained before, the *Context Table* contains the list of all the *Schema Context* structures known by the node. Each *Schema Context* is created by processing the data model schema. Also, as the schema is processed, each *Element Context* is created strictly following the order in which the elements are defined.

When processing an element, the process checks whether it already exists in the *Schema Context*. In case it does not exist, a new *Element Context* is created, together with an unique *Id* identifier, and *MultipleParents* attribute is set to *false*. In case the element already exists, *MultipleParents* is set to *true*. Then, its attribute *Template* is assigned with the identifier of the associated template in the *Template Table*.

In case (1) the order of appearance of children is fixed and (2) the appearance matches the order defined in the schema, *ContentType* attribute is set to *fixed*. If the appearance order of the children can be different, *ContentType* attribute is set to *dynamic*. Finally, in case only one of the children can appear, *ContentType* attribute is set to *choice*.

Once *Element Context* attributes have been set, children links are processed. If the cardinality related to the child is *1*, *IsOptional* and *IsArray* attributes are set to *false*. If the child has cardinality $n..m$, where $n = 0$, *isOptional* is set to *true*. If $m > 1$ then *isArray* is also set to *true*. Finally, the child type is added to *Type* attribute. In case the child type is *element*, *ElementId* attribute is set with the *Id* identifier of the associated Element Context.

Once all the schema has been processed and the *Schema Context* has been created, a process called *Context Collapsing* is performed in order to reduce the number of *Element Contexts* and *Child Contexts* without any loss of information. Starting from the root node, if two or more siblings (nodes that share the same parent) (1) are neither optional nor arrays (i.e., *IsOptional=false* and *IsArray* = *false*), (2) they only have one parent (i.e., *MultipleParents* = *false*) and (3) the order of the siblings is fixed (i.e., *ElementContent* attribute of the parent is *fixed*), then the *Element Contexts* of the children are merged together. In a similar way, if a parent and a child fulfill those same conditions, then the *Element Context* of the child is merged with the *Element Context* of the parent.

Context Table and Template Table Example. We present an example of an *Element Context* and *Template Table* generated from an XML Schema. To this end,

we use the *Notebook* XML Document example proposed by Peintner et al. [8]. Figure 4 shows the original XML Schema of *Notebook* example.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                      elementFormDefault="qualified">
  <xs:element name="notebook">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="note" type="Note"/>
      </xs:sequence>
      <xs:attribute ref="date"/>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="Note">
    <xs:sequence>
      <xs:element name="subject" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
    <xs:attribute ref="date" use="required"/>
    <xs:attribute name="category" type="xs:string"/>
  </xs:complexType>
  <xs:attribute name="date" type="xs:date"/>
</xs:schema>
```

Fig. 4. *Notebook* XML document schema.

Figure 5 shows the Template Table generated before *collapsing* (see Fig. 5a) and after *collapsing* (see Fig. 5b). Also, Table 2 shows the Context table generated after collapsing[1].

Table 2. *Element Context* example, after collapsing. The *ContentType* attribute of all the elements is *fixed*. The content of the ChildElements row corresponds with the tupla (*Type*, *IsOptional*, *IsArray*, *ElementId*). S denotes *string*, E *element*, D *date-time*, t *true* and f *false*.

Atribute	Id				
	root(0)	notebook(1)	date(2)	note (3)	category (4)
Template	-	0	1	2	3
MultipleParents	f	f	t	f	f
ChildElements	(E,f,f,1)	(E,t,f,2) (E,f,t,3)	(D,f,f,-)	(E,t,f,4) (E,f,f,2) (S,f,f,-) (S,f,f,-)	(S,f,f,-)

[1] Context table before *collapsing* is available at [3].

```
--------------------------------------- 0
<?xml version="1.0" encoding="UTF-8"?>
--------------------------------------- 1
<notebook @>
 @
</notebook>
--------------------------------------- 2
date=@
--------------------------------------- 3
<note @ @>
 @
 @
</note>
--------------------------------------- 4
category=@
--------------------------------------- 5
<subject>@</subject>
--------------------------------------- 6
<body>@</body>
```

(a) No collapsing

```
--------------------------------------- 0
<?xml version="1.0" encoding="UTF-8"?>
<notebook @>
 @
</notebook>
--------------------------------------- 1
date=@
--------------------------------------- 2
<note category=@ @>
 <subject>@</subject>
 <body>@</body>
</note>
--------------------------------------- 3
category=@
```

(b) With collapsing

Fig. 5. Schema example template table.

3 Template Management Configuration

3.1 Schema Register

Nodes need to know the templates (and their identifiers) associated with the data models they are using. This information is made available in an initial dissemination phase, in which *Template Tables* and *Context Tables* of the data models are distributed.

When a node joins the network for the first time, it can start a schema registration process. Schemas are registered in a centralized schema repository, usually located at the gateway. Nodes use the URI of the data model schema to register. When the gateway receives a registration request, it first checks whether that schema is already registered. In that case, the associated *SchemaId* is returned. If the URI is not registered yet, the gateway generates a new *schemaId*.

When registering a schema, an associated URL is provided so that its data model schema can be accessed and downloaded. Schemas can be stored at a node (see Fig. 6a) or at an external server (Fig. 6b). Once the gateway has downloaded a schema, it generates the *Context Table* and *Template Table*. As an efficiency improvement, the schema repository could also pre-load a set of standard schemas or download already pre-compiled *Context Tables*.

Note that constrained nodes only need to store the schemas of the data models they actually use. Moreover, if the schemas can be accessed from an external server, they can be totally stripped from the node.

In case a central schema repository were not available, nodes could select proper identifiers based on point-to-point agreement. Optionally, distributed system convergent algorithms could be used.

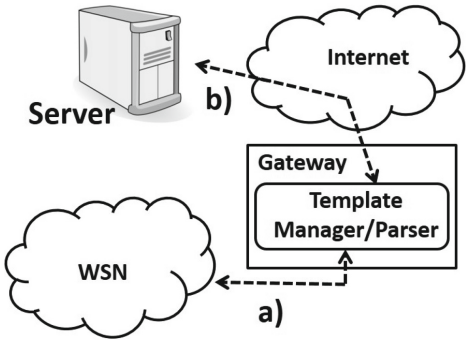

Fig. 6. Template location. (a) at the Node, (b) at an external server.

After the registration process, the *schemaId* and *Context Table* are available in the schema repository so that nodes can access them in their bootstrapping phase.

4 Performance Evaluation

In this section we present two performance tests in order to compare context based compression and EXI. In the first test, a set of XML instances are compressed by using (a) EXIficient [2], an implementation of EXI, and (b) a preliminary implementation of the templates and context based compression approach. In the second test, we study the performance of the decoding process using as input the streams obtained from the previous test. In order to decode EXI streams, another EXI implementation, EXIP [7], is used.

The set of XML documents used in the tests is composed of the *netconf* and *SenML* instances (three documents each) used in the EXIP evaluation paper [7], as well as the notebook XML instance used as an example in the EXI Primer web page [8].

4.1 First Test: Encoding

In this test we compressed the XML instances using the EXIficient [2] implementation of EXI. In order to ensure fairness in the comparison, the encoding was performed setting the *schema strict* option to *true* and all the *preserve* options to *false*. The options were not included in the header of the encoded stream. For the Context Based compression we created the Context and Template Tables from the schemas and performed the encoding using the grammar presented in this work. The results in terms of size are shown in Table 3.

Observe that results are very similar. It is interesting to note that in the case of the *SenML-01* document, EXI shows better compression results. The reason lies in the fact that our proposal is not able to compress occurrences of strings outside the schema, while EXI does not differentiate between strings belonging to data and schema space.

Table 3. XML document compression comparative. Numbers are in bytes.

XML	original	EXIP	CB
notebook	297	59	60
netconf-01	395	21	20
netconf-02	660	51	49
netconf-03	423	3	2
SenML-01	448	97	129
SenML-02	219	61	59
SenML-03	173	45	43

4.2 Second Test: Decoding

For the second test we decoded the EXI streams produced in the previous encoding test using the EXIP v5.4 [1] implementation of EXI. Only decoding time has been measured (grammar generation time has not been considered). In the case of the context based approach, we used a preliminary implementation of the Templates and Context Based compression to decode the same streams compressed in the previous test.

We performed 1000 runs for each stream encoding in similar system load conditions. Table 4 shows the results. Two columns are presented for each case. The first one contains the non-optimized program (compiled with -O0) measurements while the second one is the optimized program (compiled with -O3).

Table 4. XML document compression time comparative. Numbers are in milliseconds.

XML	EXIP		CB	
notebook	25.3	23.1	15.0	13.8
netconf-01	28.6	24.4	7.9	6.8
netconf-02	38.4	32.0	15.4	13.1
netconf-03	30.1	25.7	7.5	6.4
SenML-01	38.7	33.6	23.7	19.7
SenML-02	25.6	22.2	16.1	14.3
SenML-03	20.5	18.3	14.0	12.5

Observe that decoding compressed streams is significantly faster[2] in the case of Context Based Compression. This is a direct result from using the simpler grammar and structure of the Context and Template Tables compared to the EXI specification.

[2] This is very relevant for resource-constrained devices as sorter processing cycles imply lower energy consumption.

Regarding memory requirements, performed tests showed that RAM usage by EXIP [7] was 23KB. In the case of Context Based Compression, the RAM allocations of the Context and Template Tables for the *notebook.xsd*, *senml.xsd* and *netconf.xsd* schemas were 565, 969 y 3611 bytes respectively. Also note that the Context and Template tables are constant data so, for the most resource-constrained devices, they could be stored and accessed from flash in order to save RAM. Finally, the programming memory used by the core Context based implementation was 7KB, while EXIP reported 79KB.

Results from the tests show that Context Based Compression performed better in terms of processing time (31.3 %-75.2 % time reduction) and memory usage (78.2 % less RAM and 89.9 % less program memory).

5 Conclusions

In this paper we have presented a context based compression technique that can be applied to formatted data models. Context based compression allows a better performance in the representation of data model structures, keeping independence from the original format. Additionally, since templates are used, different *Template Tables* can be applied to the same *Context Table* in order to manage different representations, such as XML or JSON or even the same XML document with different extensions (e.g. with or without comments).

We have shown that Templates and Context based compression provides good performance results compared to EXI implementations. Context Based Compression achieves better performance than EXI implementations in terms of speed and memory usage, while keeping a similar efficiency in terms of space in its better case (*Schema Strict*). Additionally, since templates are used, the binding between the *Schema Table* and the original document is also achieved in an speed efficient way.

On the other hand, the preliminary version of Templates and Context based compression does not offer support for deviations from the data model schema. As a consequence, data models can not be extended (for instance, for nesting multiple data models) and extensibility attributes of the native formats (e.g., `<any>` and `<anyAttribute>` XML elements) are not currently supported, so they are ignored in the parsing process.

As a future work, we are considering to improve Context based compression in order to enable Schema Context extensions and nested data models. This mechanism would also allow assigning templates to dynamic parameters, improving the compression of repeated values in the data space. We are also studying the representation in the Context Table of constraints described in the schema. This would improve the compression and the addition of partial validations of the coded streams. Finally, one of our main objectives is to develop a modular library which allows to be configured and customized to the resources of a given system. This library will provide an API and required functions to manage the formatted data in an efficient way. Additionally, we are also planning to develop a toolbox which will automatically generate Context Tables and *Template tables* from standard schema formats (such as XML and JSON schemas).

Acknowledgements. Research partially supported by the European Union Horizon 2020 Programme under grant 680708/HIT2GAP, by the Spanish Research Council, grant TIN2013-41123-P, and the University of the Basque Country UPV/EHU, grant UFI11/45.

References

1. Embeddable EXI processor in C (2016). http://exip.sourceforge.net/. Accessed June 2016
2. Exificient (EXI procesor) (2016). http://exificient.github.io/. Accessed June 2016
3. Berzosa, J., Cortiñas, R., Gardeazabal, L.: Efficient management of data models in constrained systems by using templates and context based compression. Technical report, University of the Basque Country UPV/EHU, Computer Science Faculty (Donostia, San-Sebastian) (2016). http://go.ehu.eus/BerzosaGC16-TR-05-16.pdf
4. Bournez, C.: Efficient XML interchange evaluation. Technical report, W3C (2009). http://www.w3.org/TR/exi-evaluations/. Accessed June 2016
5. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Frystyk Nielsen, H.: SOAP version 1.2 part 1: messaging framework. Recommendation, W3C, June 2003. http://www.w3.org/TR/2003/REC-soap12-part1-20030624
6. Gutierrez, J.A., Callaway, E.H., Barrett, R.: IEEE 802.15.4 Low-rate wireless personal area networks: enabling wireless sensor networks. IEEE Standards Office, New York, NY, USA (2003)
7. Kyusakov, R., Punal Pereira, P., Eliasson, J., Delsing, J.: EXIP: a framework for embedded web development. TWEB **8**(4), 23:1–23:29 (2014). doi:10.1145/2665068
8. Peintner, D., Pericas-Geertsen, S.: Efficient XML interchange (EXI) primer (2014). https://www.w3.org/TR/exi-primer/. Accessed June 2016
9. Schneider, J., Kamiya, T., Peintner, D., Kyusakov, R.: Efficient XML interchange (EXI) Format 1.0 (2nd edn.). Technical report, W3C (2014). http://www.w3.org/XML/EXI. Accessed June 2016
10. Shelby, Z., Bormann, C.: 6LoWPAN: The Wireless Embedded Internet. Wiley Publishing, Hoboken (2010)
11. Shelby, Z., Hartke, K., Bormann, C.: The constrained application protocol (CoAP). Technical report 7252, RFC Editor, Fremont, CA, USA, June 2014. http://www.rfc-editor.org/rfc/rfc7252.txt
12. White, G., Kangasharju, J., Brutzman, D., Williams., S.: Efficient XML interchange measurements note. Technical report, W3C (2007). http://www.w3.org/TR/exi-measurements/. Accessed June 2016