Oriental Journal of Computer Science & Technology Vol. **4**(1), 159-164 (2011)

A Compression Method for PML Document based on Internet of Things

JAMAL MOHAMMAD AQIB

Research Scholars of Singhania University, 2/194 Vikas Nagar, Kursi Road,
Near Lekhraaj Panna Market, Lucknow - 226 022 (India).

(Received: April 25, 2011; Accepted: May 29, 2011)

ABSTRACT

In this paper we present a compression method for PML (Physical Markup Language) document based on Huffman Algorithm. The algorithm and Java code pieces are presented. And the test result shows the compression method has good compression ratio for the PML document. The Physical Markup Language (PML) is designed to be a general, standard means for describing the physical world including human and machines. How to save the PML document in internet server efficiently will become more and more important.

Key words: Physical Markup Language; Internet of Things; Huffman Code; compression.

INTRODUCTION

Internet of Things (IOT) is a concept that visualizes the vision for bringing the internet even to dummy things [1]. It is attributed to review of the Auto-ID Center infrastructure. In this architecture the objects surrounding us would all be part of a global infrastructure: the EPC (Electronic Product Code) Network. As a key technology of this network, Radio Frequency Identification (RFID) is arguably the ideal solution for object identification. RFID is one of the enabler technologies. It has successfully been used in a large variety of applications already, like enterprise supply chain management for inventorying, tracking and of course objects identification [2]. Though a RF reader, the EPC tag in standard [3] can be collected and transmitted to trace the good or assert.

Besides the EPC, which is the fundamental component of an EPC Network, the Physical Markup Language (PML) is also of great importance. As defined in[4] the "PML is a collection of vocabularies used to represent information related to EPC network enabled objects". The most important of its vocabularies is certainly the PML Core. Its objective is to provide a standardized format for information interchange within an EPC Network. It is used to model and encapsulate the data captured by the Auto-ID sensors (i.e. RFID readers or antennae), providing a generic markup-language.

Messages based on the PML Core schema can be exchanged between any two XML (Extensible Markup Language) [11] enabled systems in the EPC Network. Typically the information exchange based on the PML Core schema will occur between Savants [12] and the EPC Information Service and/or other enterprise applications[4]. How to save these PML data efficiently in server database is an important issue. A typical idea is to decrease the data/document size by compression.

PML SERVER

Designing the PML server is the purpose to store information of objects, batch orders and

manufacturing recipes and to make this information available to an Auto-ID enabled robotic manufacturing environment.

An example of PML services with EPC information exchange during a supply chain is given in Fig. 1:

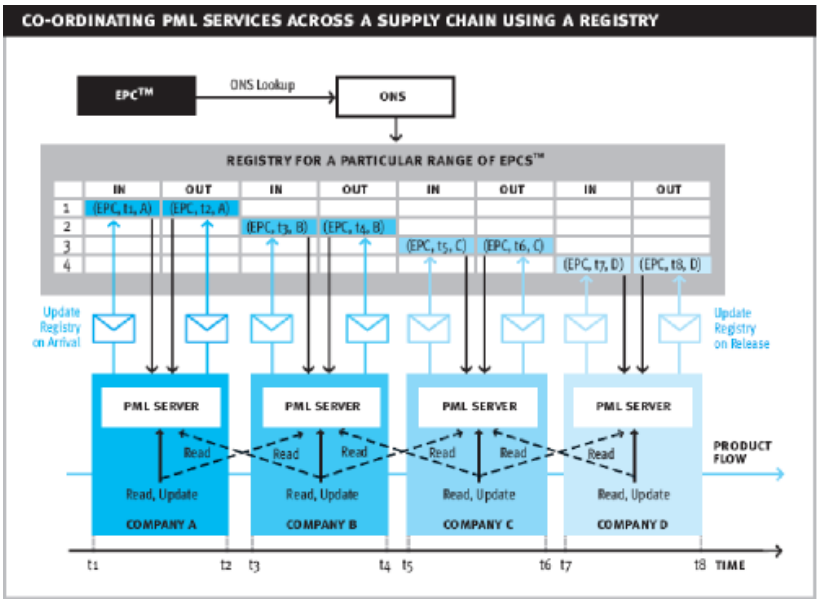

Fig. 1: PML services on a supply chain[9]

The ONS means Object Name Service, and it's used to provide URLs to authoritative information given an EPC, which works similarly to the well known Domain Name Service (DNS). More information about PML server development can be found in [9] and [10].

PML DATA FORMAT

Before decompression, we need to understand how the PML data is formatted. The PML Core language is based on an XML syntax validated by two XML-Schema:

PmlCore.xsd: which contains the Sensor related elements.

Identifier.xsd: which provides a validation schema for unique identifiers. It is worth noting that the use of the EPC scheme as the identifier within PML files is strongly encouraged but not mandatory.

Take PML Core for example, it defines a set of elements used to report the sensors' observations called the PML Core Sensor Data Model. A sample of sensor element is shown as below:

```
<pmlcore: Sensor>
 <pmluid:ID>urn:epc:1:4.16.36</p
 mluid:ID>
  <pmlcore :Observation>
   <pmlcore:DateTime>2011-02-
06T13 :04:34-06: 00</pmlcore
: DateTime>
    <pmlcore : Tag>
    <pmluid:ID>urn:epc:1:2.24.40
    0</pmluid:ID> </pmlcore :
    Tag>
    <pmlcore : Tag>
    <pmluid:ID>urn:epc:1:2.24.40
    1</pmluid:ID> </pmlcore :
    Tag>
  </pmlcore :Observation>
</pmlcore: Sensor>
```

Such PML data, large or small, with its associated schema, will be exchanged between

various applications running on diverse devices. Good storage methods will benefit the enterprise and the clients a lot.

From the PML sample we can see that the elements like "ID" and "Tag" appears for several times, which can be called "redundancy" when saving the data. They're not necessary to be saved every time when they appear. So we need a compression method to deal with this situation.

Compression method

Lossy and Lossless Compression

There are two basic lossy data compression [5] are:

1 Lossy compression means that some data is lost when it is decompressed. Lossy compression bases on the assumption that the current data files save more information than human beings can "perceive". Thus the irrelevant data can be removed.

2 Lossless compression means that when the data is decompressed, the result is a bit-for-bit perfect match with the original one. The name lossless means "no data is lost", the data is only saved more efficiently in its compressed state, but nothing of it is removed.

Obviously the Lossy compression is not out cake because the PML data such as EPC tag needs to be accurate to identify the object after decompression. Among the lossless compression techniques like LZW (Lempel-Ziv-Welch) compression, Burrows-Wheeler transform and Huffman code, we choose the Huffman code, which uses variable length codes based on frequency.

Huffman Coding

Huffman coding is an entropy encoding algorithm used for lossless data compression. It was developed by David A. Huffman in 1952, based on the frequency of occurrence of a data symbol. Its main principle is to use fewer bits to represent frequent symbols and use more bits to represent infrequent symbols. It's efficient when symbol probabilities vary widely. Its aim is to find a prefix-free binary code (a set of codeword) with minimum expected codeword length (equivalently, a tree with minimum weighted path length from the root), given a set of symbols and their weights (usually proportional to probabilities). It has a formalized description:

Input is

$A \wedge a1, a_2, ..., a_n$ ', which is the symbol set of size n.

Set

$W \wedge w1, w_2, ..., wn$ ', which is the set of the (positive) symbol weights (usually proportional to probabilities), where *W i weight ai* 1,. d *i* d *n*

Output is the code

C A, W ^ *c*1 , *c* 2 ,..., *cn* ' , which is the set of (binary)

codeword, where *ci* is the codeword for *ai*

i dd1 . *n* Its goal is to let

n

LC¦ u *w i lengthc*

i

i 1 be the weighted path length

of code C. Condition: *L C* d *LT* for any code *T A,. W*

Huffman Tree

The Huffman tree is a frequency-sorted binary tree. The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority.

Algorithm

Before doing the compression, we need to parse the PML document, and code the elements part in the document using Huffman code according to the frequency of the elements. Fig.2 describes the architecture of the compression method.

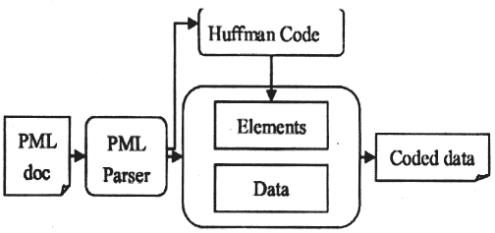

Fig. 2: Architecture of compression

The input and output of this mechanism are all static files. The input is a standard PML document. If to support the query after compression, the output result need to keep the parent and child relationship. When a PML document is input, it is parsed by a PML parser. The frequency information of the elements is

collected and the weight of each element is calculated to build the Huffman tree. The codeword are saved and the document can be compressed.

And when the document is decompressed, the tree is used to decode. Since the Huffman tree has the minimum weighted path length, the codeword length is shortest, which will make a good compression ratio.

The algorithm to build the Huffman tree is

1. Given n elements' weight, build a initial binary tree set *T ^ t*1 2 , *t ,..., tn ' ti,* every tree member just has one root node with weight , and its left and right child nodes are w i all null;
2. Choose 2 trees out of T which have minimum weight for the root node, and use them as left and right nodes to construct a new tree. The root node's weight of this new tree should be equal to the sum of the 2 trees' root node weight;
3. Remove the 2 trees in step 2 from the T set, and add the new generated tree into T;
4. Repeat step 2 and 3, until there is only one tree left, which is the final Huffman tree.

Code Realization

Before talking about the compression and decompression code detail we can firstly make a simple example. If we have an elements' weight table of a PML file as below,

Table 1: Elements weight table

PML Elements	Weight
<pmluid:ID>	1
<pmlcore:Tag>	2
<pmlcore:Data>	4
<pmlcore:DateTime>	5

Compression flow

1. Collect the frequency/weight of PML elements into a frequency table like a HASH table;
2. Reads elements and their weight from the table and creates the nodes;
3. Build the Huffman tree and traversing this tree to create the codeword;

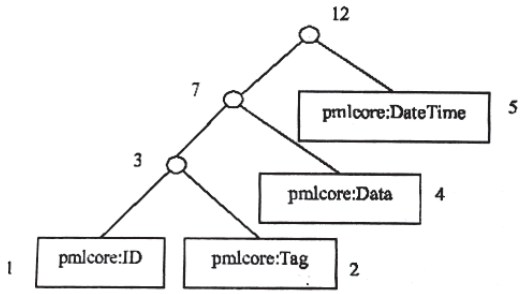

Fig. 3: Then the Huffman tree can be built

4. Encrypt the input PML file data with Huffman encoding. Code implementation in Java langages:

```
//Class Huffman node: class Node {
public int weight; public char c;
public Node leftchild; public Node rightchild; public
Node parent; public String code;
public Node (String v, String str) {
weight = Integer.valueOf(v);
c = str.charAt(0); leftchild = null; rightchild = null;
parent = null;
code = "";
}
}
//Build the Huffman tree:
private static ArrayList<Node> nodes = new
ArrayList<Node>();
Node root = new Node("0", "\0"); while (nodes.size()
- 1 > 0) {
if (nodes.size() == 1) {
root = nodes.get(0);
} else {
Node r = new Node("0", "\0"); Node x =
get min(nodes); Node y = get min(nodes); root = r;
r.leftchild = x;
r.rightchild = y;
r.leftchild.parent = r; r.rightchild.parent = r; r.weight
= x.weight + y.weight; insert (nodes, r);
}
}
```

2) Decompression flow:

Decompression is also the process of decode, which is to traverse the Huffman tree based on the code to decode into the original data. The code sample is as below:

```
//Decode using the Huffman tree:
public static void decode (Node root) { if (root.c != '\0') {
//write into data out stream } else {
if (get_first_bit() == '0') { decode (root. leftchild); } else {
decode (root.rightchild); }
}
}
```

Note: when creating the codeword to add prefix to the nodes, we assign '0' for left child and '1' for right child.

Verification Test

Since there's no standard PML data set for testing yet, so we edit by ourselves several different sizes of PML files to test. The result shows the compression ratio varies from 58% to 76%. Although these test files just have simple structure which cannot represent all the randomness of data, it does show this compression method based on Huffman coding is applicable.

Possible Improvement

Here we consider some possible improvements for this compression scheme, which can be the work direction of future work:

Query support after compression

To support the query of PML data after the compression, we need a good PML parser to parse the PML document well and save the relationship of the elements and data. And when explaining the query commands, the relationship will be checked and the required information can then be extracted out.

Compress the data part

Currently only the elements parts of PML document are compressed, the data parts are directly saved. We may consider other compression algorithms like LZMA(Lempel-Ziv-Markov chain-Algorithm)[7], which is a dictionary compression scheme somewhat similar to LZ77, to compress the data part and make the whole compression ratio higher.

Improve the Huffman code

There are some kinds of improved Huffman coding, such as the algorithm in[8], it uses overflow of code and a heap sort algorithm, and it accelerates the coding speed. It will be good when saving large number of files.

CONCLUSION

In future communication networks, the communicate forms will expand from current human-human to human-human, human-thing and thing-thing (also called M2M). And the PML which describe the objects in this network will be more and more important. In this paper we propose a compression method for the PML document storage in the Server database. The experimental result shows the compression performance is good. And this mechanism can be applied in future EPC network and the Internet of Things.

REFERENCES

1. M. P. Michael and M. Darianian, "Architectural Solutions for Mobile RFID Services for the Internet of Things," services, pp.71-74, 2008 IEEE Congress on Services - Part I (2008).
2. V. Kolias, I. Giannoukos, C. Anagnostopoulos, I. Anagnostopoulos, "Integrating RFID on event-based hemispheric imaging for internet of things assistive applications," In PETRA 2010 ACM International Conference Proceeding Series, 2707-2714 (2010).
3. EPCglobal, "EPC Tag Data Standards Version 1.1 Rev.1.24," Technical report, EPCglobal (2004).
4. C Floerkemeier, D. Anarkat, T. Osinski and M. Harrison, "PML Core Specification 1.0," Technical report, Auto-ID Center, 2003.
5. M. Sharma, "Compression Using Huffman Coding," IJCSNS International Journal of Computer Science and Network Security, vol. 10, May. 2010 pp. 133-141 .
6. C L. Brock, "The Physical Markup Language," Auto-ID White Paper, WH-003, Feb. 2001. http://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93M

arkov_chain_algorithm
7. F. L. Zhang, S. F. Liu, "Huffman*: improved huffman data compression algorithm.Computer Engineering and Applications," 2007, 43(2): pp. 73-74.
8. M. Harrison, H. Moran, J. Brusey, D. McFarlane, "PML Server Developments," Auto-ID White Paper, WH-015, (2003).
9. M. Harrison, D. McFarlane, "Development of a Prototype PML Server for an Auto-ID Enabled Robotic Manufacturing Environment", Auto-ID White Paper, WH-010, (2003). http://en.wikipedia.org/wiki/XML
10. S. Clark, K. Traub, D. Anarkat, T. Osinski, "Auto-ID Savant Specification 1.0 Version of 1", (2003).