Online Internet Communication using an XML Compressor

Tomasz Müldner
Jodrey School of Computer Science Acadia University
Wolfville, B4P 2A9 NS, Canada
e-mail: Tomasz.muldner@acadiau.ca

Jan Krzysztof Miziołek
IBI AL University of Warsaw, Poland
e-mail: jkm@ibi.uw.edu.pl

Christopher Fry
Jodrey School of Computer Science Acadia University
Wolfville, B4P 2A9 NS, Canada
e-mail: chrisfry99@gmail.com

Abstract—Online communication and various other Web applications, such as collaborative systems using XML as a data representation often suffer from performance problems caused by the verbose nature of XML. In this paper, we present an XML-conscious compressor designed to alleviate these problems, by using it online and evaluating queries using lazy decompression. Our XML compressor not only decompresses the data whenever enough data are available, but it also compresses them online, it is updateable (i.e., it works with dynamic XML documents), and its implementation can be parallelized thereby significantly increasing performance on multi-core machines.

Keywords - Internet communication; XML; compression.

I. INTRODUCTION

Online communication is increasingly using the eXtensible Markup Language (XML) [1] as a data format. Unfortunately, the XML markup results in increased size of this representation, often by as much as ten times as large as alternative formats. This overhead is particularly concerning for communications involving large data sets and for memory-constrained devices participating in online communication. For applications passing communicating over the Internet, the network bandwidth is the main bottleneck and this is why decreasing the size of the information passed is essential. There has been considerable research on XML-conscious compressors, which unlike general data compressors can take advantage of the XML structure, e.g., [2], [3], [4]. Most recently, there has been research on queryable XML compressors for which queries can be answered using lazy decompression, i.e., decompressing as little as possible, see, e.g., [5], [6]. Also, there has been research for updateable XML compressors, for which updates can be saved without full decompression, see, e.g., [7], [8]. Online XML compressors are typically defined as compressors, which decompress chunks of compressed data whenever possible rather than doing it offline when the entire compressed file is available, see [9], [10]. These compressors are particularly useful for Internet applications, used on networks with limited bandwidth. Clearly, for a compressor to be online means that it must have only one pass through the document is required to compress it. In this paper, we present an online compression based on XSAQCT, an XML compressor, see [11]. There are other online compressors, e.g., TREECHOP [12], but XSAQCT has a number of distinctive features, as it is queryable using lazy decompression (i.e., with minimal decompression) and updateable [7], and finally it can be parallelized to execute faster on multi-core machines [13]. Various possible educational applications of XSAQCT are described in [14]. Similarly to TREECHOP, XSAQCT supports both the compression where the decompressor's output is exactly the same as the original input (including the white space), and generating a canonicalized [15] XML document.

Contributions. We present a novel online XML compressor suitable for improving online communication on Internet. Our initial experiments indicate that XSAQCT's performance is comparable to TREECHOP, but unlike TREECHOP, XSAQCT not only decompresses the data whenever enough data are available, but it also compresses them online, which is essential for the case of a network node N1 receiving streamed XML data from one or more sources, which are to be stored in a compressed form. In such cases, instead of waiting for the entire set of XML data, N1 may compress incoming data online thereby increasing the efficiency of the compression. In addition, XSAQCT is updateable (i.e., it works with dynamic XML documents) and its implementation can be parallelized thereby significantly machines. increasing performance on multi-core This paper is organized as follows. Section II gives a short introduction to the design and functionality of the previous version of XSAQCT, which is offline, and Section III describes its current extension, i.e., the online XSAQCT. Section IV describes applications of XSAQCT to an online communication, and finally, Section V provides conclusions and describes future work.

II. OUTLINE OF OFFLINE XSAQCT

Given an XML document D, we perform a single SAX (specifically using Xerces [16]) traversal of D to

 ISBN: 978-1-61208-200-4

encode it, thereby creating an annotated tree $T_{A,D}$ n which all similar paths (i.e., paths that are identical, possibly with the exception of the last component, which is the data value) are merged into a single path and each node is annotated with a sequence of integers; see Fig. 1. When the annotated tree is being created, data values are output to the appropriate data containers. Next, $T_{A,D}$ is compressed by writing its annotations to one container and finally all containers are compressed using a selected back-end compressors, e.g., gzip [17]. For more details on XSAQCT, see [11] and [7].

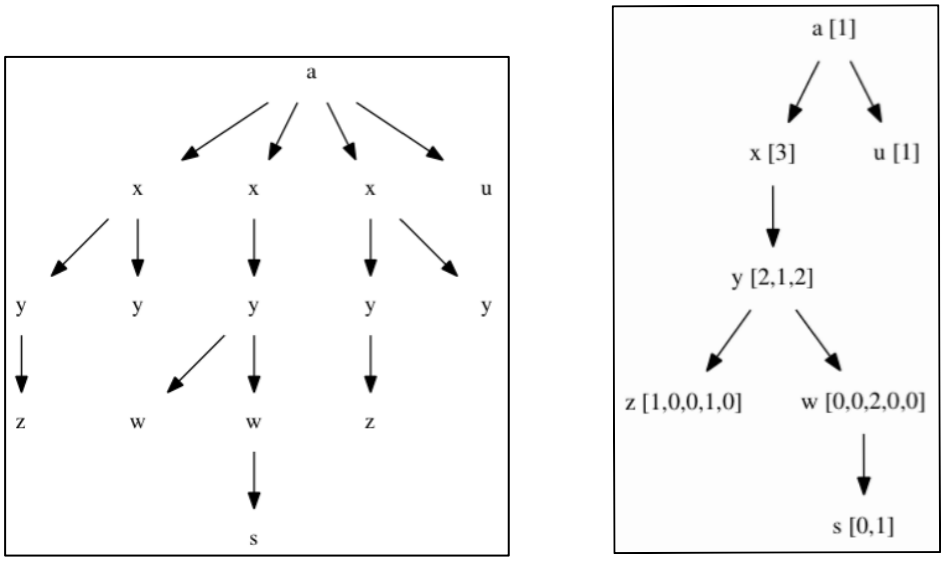

Fig.1 XML document (a) and its annotated tree (b)

III. ONLINE XSAQCT

A. Notations and Assumptions

In this version, we assume that a leaf of an XML tree stores exactly one text child. By SN we denote a sending node and by RN we denote a receiving node. SN and RN communicate using message passing; here SN is a producer and RN is a consumer using receive(packet); a synchronization is taken care of by these procedures. A packet may have binary contents or it can be a sequence of integer values. By the skeleton tree T_D we denote the tree labeled by tag names, and by we denote an annotated tree. By ANN we denote the sequence of all annotations. Annotations for a node of $T_{A,D}$ may be stored with this node, or the node may store a (logical) pointer to ANN (e.g., the offset within ANN). We assume that an annotated tree $T_{A,D}$ is implemented so that following functions are available:

- Node ADD_RC(Node n, Tag p, annotation a) creates and returns a new rightmost child of n with the tag p and the annotation a;
- Node create_Root(Tag p) creates a new root with tag p;
- Node get_LC(Node n) returns the leftmost child of n or a null node;
- Node get_RS(Node n) returns the right sibling of n or a null node;
- bool function is_Text(Node n) returns true iff n is a special tree node to store text;
- Node get_Parent(n) returns the parent of n; text get_Tag(n) returns the tag of n;
- text get_Text(n) returns the only text child of n.

We also assume that a data structure Path stores tags or text value, with the operations append_Path(Path p, Node n) which appends n to the path p, clear_Path(Path p) which sets the path p to empty, length_Path(Path p) which returns the length of p, and set_Path(Path p) which stores length_Path(p) as the first element of p. Finally, we use the following notations:

- a(n) annotation of the node n
- a(n)+=j increase the last annotation of n by j
- a(n)+="0" add ", 0" to the annotation of n; e.g. if a(n)=[1] then it becomes [1,0]
- $[0^{a(m)},1]$ if a(m) is [1], then $[0^{a(m)},1]$ is [0,1] otherwise $[0^{a(m)},1]$ is [0,…,0, 1] where $0^{a(m)}$ is the sum of all annotations in a(m), minus 1; e.g., if a(m) = [2,1], then $[0^{a(m)},1]$ is [0, 0, 1].

B. Online Compression

SN parses XML data using the SAX parser, and sends packets to RN, which first creates an annotated tree (as described below) and then follows the compression process as in XSAQCT [11]. At the same time, the parser creates a dictionary of tags. Each packet is of the form: (integer k, followed by N indices into the dictionary, followed by the uncompressed text) where

 ISBN: 978-1-61208-200-4

N>=0. We assume that when the SAX parser terminates (i.e., completes parsing) it sends the packet (-2, dictionary). Therefore, RN creates an annotated tree labeled by indices rather than tags. For the sake of readability, in the description provided in this paper, we consider sending and receiving tags rather than indices but our implementation operates on indices.

```
int k = -1;
Path p;
//initially stores only the tag of the root of the XML tree
void SN_send(Node n) {
        if (n is a leaf) {
                // a leaf must have a text child
                append_Path(p, getText(n));
                set_Path(p);
                send(p);
                clear_Path(p);
                k=0;
        } else
                for every child m of n {
                        append_Path(p, getTag(m));
                        SN_send(m);
                        k++;
                }
} // SN_send()
```

Now, we'll explain the actions executed by RN. By the *leftmost path* of the labeled tree T rooted at node n_1 we mean a path of labels $(p_1,\ldots,p_k)$ s.t. get_Tag(n_1) = p_1, and for i = 1,…,k-1 get_LC(n_i) = n_{i+1}, get_Tag(n_i) = p_i, get_LC(n_k) is null. The *first time* send() is called, it will send a packet (-1, the path of the leftmost path rooted at the root of the tree). At this time the value of the *current node* c will be set to n_k. To explain the meaning of sending the consecutive packet (k, $p_1,\ldots,p_N$) let's assume that SN_send() is visiting nodes (which are initially non-visited) in a dfs-order and c is current node. Then the value of k is found as follows. Let n_1, …,n_k be the shortest path of nodes s.t. n_1 = c, for i=1,… k-1 get_Parent(n_i) = n_{i+1} and there exists a child m of n_k which has not been visited. Next, let $m_1,\ldots,m_N$ be a path of nodes s.t. m_1=m, for i=1,…,N-1 m_{i+1} is the leftmost unvisited child of m_i, m_N has only a text child. Then, for i=1,…,N get_tag(m_i)=p_i.

```
void RN_receive(Node n) {
  bool flag;
  Node c; // current node
  Node m;
  Text t;
  receive(k, p1,…,pN , t);
  if(k==-1) { // initialization
      c = create_Root(p1);
      for(i=2; i<N; ++i)
              c = ADD_RC(c, pi, [1]);
  }
  while (true) { // until the final packet
       receive(k, p1,…,pN , t);
       if(k == -2)
           return; // done
          // move current based on the value of c
        for(i=1; i<=k; ++i) // set the current
           c = get_Parent(c);
        // check every tag in the received path
        for(i=1; i<=N; ++i) {
            flag = false;
            for (every child m of c)
                if(get_Tag(m) == pi) {
                    a(m)+=1;
                    c = m;
                    flag = true;
                    for (every child m of c)
                        a(m) += ",0";
                    break;
                } // end of if
            // for every child
            if(!flag)
                c = ADD_RC(c, pi, [0^a(c),1]);
            add text to the container for c;
        } // for i=1…
    } // while(true)
} // RN_receive()
```

C. Online Decompression

We assume that the sending node SN can decompress all annotations, restore the skeleton tree and send it to RN, then re-annotate it as well as run a procedure SN_dfs(AnnotationTreeNode) shown below. As far as the receiving node RN is concerned, we assume that it can run a procedure RN_restore(SkeletonTreeNode) shown below. We also assume that RN implements the AA Abstract Data Type (ADT), which stores sequences of annotations with the following operations (initially, the annotations for every node are un-initialized):

- void AA_delete(Node n) removes the first element of the annotations for n
- void AA_store(Node n, sequence of integers seq) stores seq as the annotations for n
- void AA_init(Node n) initializes the annotations for n
- bool AA_isInit(Node n) returns true iff the annotation for n has been initialized
- int AA_getFirst(Node n) returns the first element from the annotations for n
- AA_get_Text(Node n, binary b) where b contains a compressed text, performs the following actions: b is decompressed, stored into a container, and then the iteration AA_nextIter(Node n) is started, this iteration returns the next text in the container
- bool AA_hasReceivedText(Node n) returns true iff the text for n has been received

D.Initialization

SN restores the skeleton tree T_D and then the annotated tree $T_{A,D}$ (but it does not decompress text containers), then it sends the *skeleton* tree to RN:

SN: send(T_D)
RN: receive(T_D)

After the initialization, RN runs the following procedure:

```
SN_dfs(AnnotationTreeNode f) {
    for (every child c of f)
        if(isText(c))
            send(c, text of c);
        else {
            send(ANN(c));
            SN_dfs(c);
        }
} // SN_dfs()
```

For the RN, we show the recursive version:

```
RN_restore_recursive(SkeletonTreeNode f) {
   for (every child c of f)  //from left to right siblings
      if (is_Text(c)) {
         if (!AA_hasReceivedText(c))
            AA_getText(c);
         output(AA_nextTextIter(c));
         return;
      } else {
         if (!AA_isInit(c)) {
            receive(ann);
            AA_init(c);
            AA_store(c,ann);
         }
         while (AA_getFirst(c) > 0) {
            output("<" + tag(c) + ">");
            a(c)+=-1;
            RN_restore_recursive(c);
            output("</" + tag(c) + ">");
         }
         AA_delete(c);
   }
} // RN_restore()
```

For the XML file from Fig. 1 (a), in Fig. 3 we show packets that will be sent by SN_send() and the state of the annotated tree after each packet has been processed by RN_restore(), (the current node is bold, un-annotated nodes have annotation [1]). Note the last state (in the right bottom corner) shows the same annotated tree as in Fig 1 (b).

E. Querying Strategy

For queries formulated using a subset of XPath, the network node receiving a compressed data can query this data as it is being processed. Specifically, XSAQCT decompresses the skeleton tree, and annotations, and then decorates the tree with annotations. Depending on a type of the query, it can be immediately answered (for exact-match queries involving only tag names, e.g. /a/b/) or (for example to find the location of some text data) XSAQCT finds the location of the data, decompresses the appropriate data container, and completes the evaluation of the query. This type of lazy decompression makes the evaluation of queries more efficient.

IV. EXAMPLES OF APPLICATIONS

To consider possible applications of online XSAQCT, see Fig. 2, consider the network node N1, which produces XML data, to be sent to the network node N2, where they are compressed online by XSAQCT. N2 stores compressed data, which can be queried by the network node N3 sending queries. They can also be decompressed online by XSAQCT and sent to the network node N4. This node can either store uncompressed data, or they can be piped into any WWW application. Therefore, this figure shows the general architecture of our system. For example, for online decompression, input data may be piped into the compressor and sent over the Internet. On the receiving end, the data may be piped into two programs; one that collects the entire compressed document, and a second program which performs on-the-fly decompression. The decompressed data can be piped into any WWW application, such as a SOAP processor. The complete compressed data can be stored, and queried without having to decompress it.

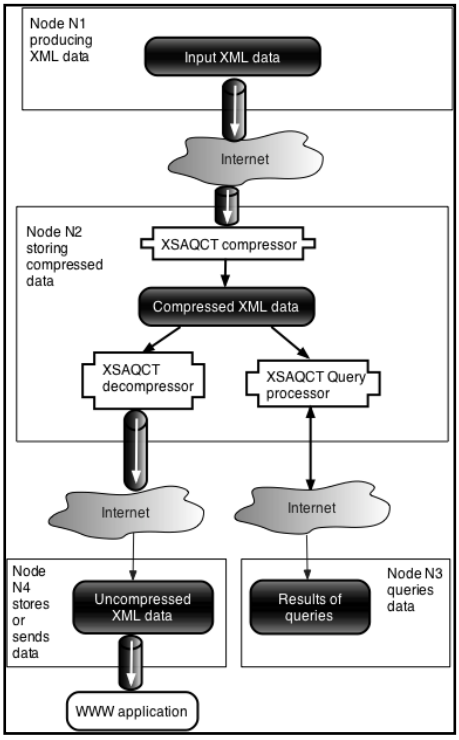

Fig. 2. Applications

V. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an online XML compressor/decompressor XSAQCT suited for efficient

implementation of online communication. We provided the brief outline of the implementation and results of the implementation.

In this version we did not attempt to handle XML mixed content or cycles, e.g., nodes with the consecutive children b, c and b. In the future version, we will remove these limitations. In addition, the future version will add more querying and updating facilities. Finally, we will add parallelization to the online compressor, based on [13]. To evaluate the effectiveness of online XSAQCT; specifically its compression and decompression and compression ratios, we will use three files of varying sizes: shakespeare.xml, dblp.xml and 1gig.xml. The first two files are taken from the Wratislavia corpus [18], while the last file is a randomly generated XML file, using xmlgen [19]. We will test our code by recording: (a) time to send a single uncompressed XML file D over the network from node N1 to node N2 and then compressing offline in N2, and (b) time to compress D online while sending from N1 to N2. Similarly, we will record (a) time to send a single compressed XML file D over the network from node N1 to node N2 and then decompressing offline.

Example**.**

For the XML file from Fig. 1 (a), in Fig. 3 we show packets that will be sent by SN_send() and the state of the annotated tree after each packet has been processed by RN_restore(), (the current node is bold, un-annotated nodes have annotation [1]). Note the last state (in the right bottom corner) shows the same annotated tree as in Fig 1 (b).

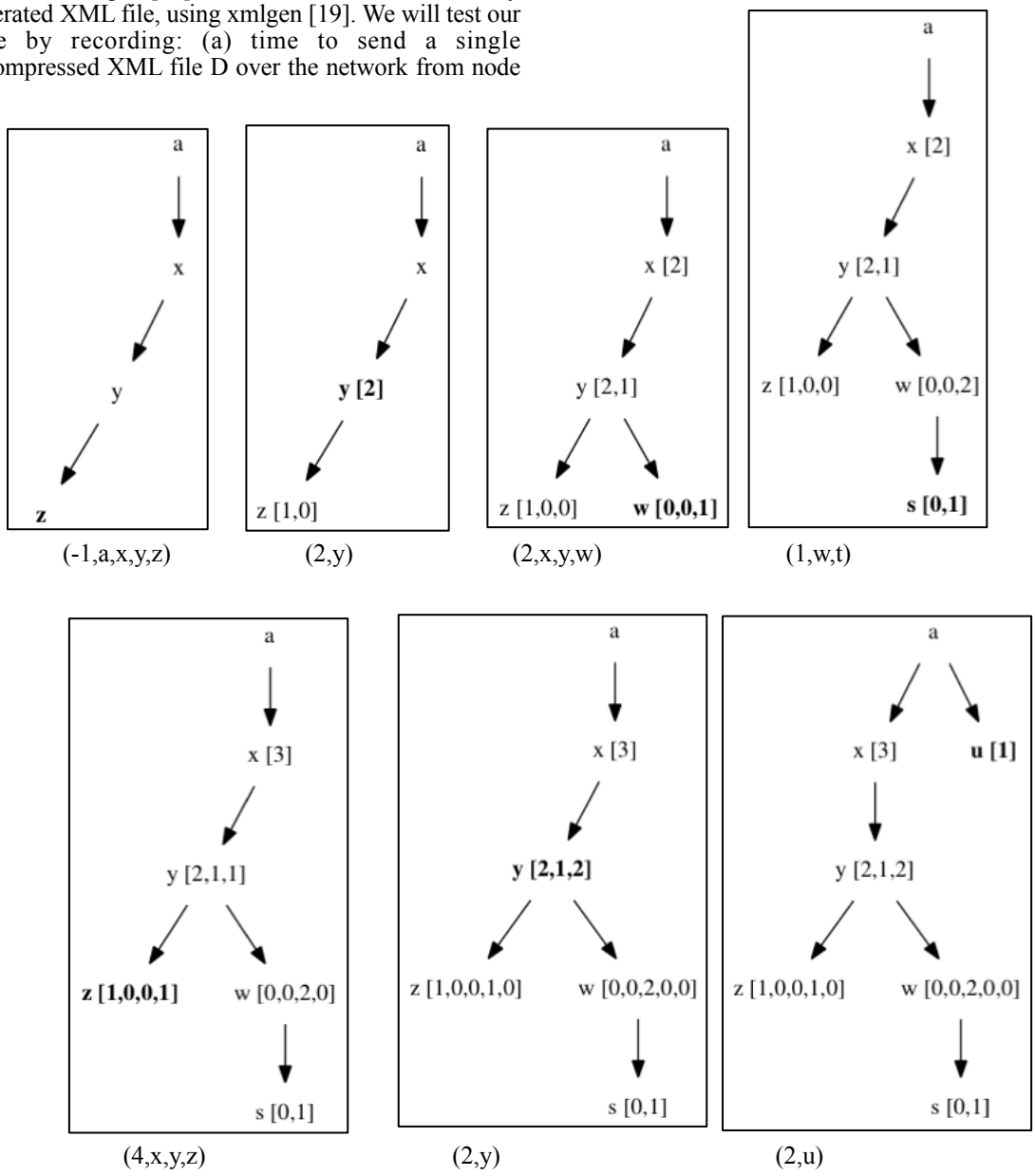

Fig.3 The state of the annotated tree after sending each packet

 ISBN: 978-1-61208-200-4

REFERENCES

[1] W3C, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. http://www.w3.org/TR/REC-xml/, 2011. Retrieved on January 20, 2012.

[2] H. Liefke and D. Suciu, "XMill: an efficient compressor for XML data," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000, pp. 153-164.

[3] P. Tolani and J. Haritsa, "XGRIND: a query-friendly XML compressor," in *Proceedings of the 2002 International Conference. on Database Engineering*, 2002, pp. 225-34.

[4] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese, "XQueC: pushing queries to compressed XML data," in *Proceedings of the 29th international conference on Very large data bases, 2010.*
Volume 29, Berlin, Germany, 2003, pp. 1065–1068.

[5] P. Skibiński and J. Swacha, "Combining efficient XML compression with query processing," in *Advances in Databases and Information Systems*, 2007, pp. 330–342.

[6] Y. Lin, Y. Zhang, Q. Li, and J. Yang, "Supporting Efficient Query Processing on Compressed XML Files," 2005.

[7] T. Müldner, C. Fry, J. K. Miziołek, and T. Corbin, "Updates of Compressed Dynamic XML Documents," in *Eight International Network Conference*, 2010, pp. 315–324.

[8] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld, "Updating XML," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, Santa Barbara, California, United States, 2001, pp. 413–424.

[9] S. Sakr, "An Experimental Investigation of XML Compression Tools," *CoRR*, vol. abs/0806.0075, 2008.

[10] Müldner, T., Leighton, G., and Diamond, J., "Using XML Compression for WWW Communication," presented at the IADIS International Conference WWW/Internet, 2005, pp. 459-466.

[11] T. Müldner, C. Fry, J. K. Miziołek, and S. Durno, "XSAQCT: XML Queryable Compressor," Montréal, Canada, 2009.

[12] G. Leighton, T. Müldner, and J. Diamond, "TREECHOP: A Tree-based Query-able Compressor for XML," *The Ninth Canadian Workshop on Information Theory*, Jun. 2005.

[13] T. Müldner, C. Fry, T. Corbin, and J. K. Miziolek, "Parallelization of an XML Data Compressor on Multi-cores," presented at the PPAM, Torun, Poland, 2011.

[14] T. Müldner, J. K. Miziolek, and C. Fry, "Updateable Educational Applications based on Compressed XML Documents," in *CSEDU (1)*, 2011, pp. 369-371.

[15] W3C, *Canonical XML*. http://www.w3.org/TR/xml-c14n, 2001. Retrieved on January 20, 2012.

[16] "Xerces," *http://xerces.apache.org/xerces-j/*. [Online]. Available: http://xerces.apache.org/xerces-j/. Retrieved on January 20, 2012.

[17] *The gzip home page*. http://www.gzip.org/. Retrieved on January 20, 2012.

[18] *Wratislavia XML Corpus*. http://www.ii.uni.wroc.pl/\textasciitildeinikep/research/Wratislavia/. Retrieved on January 20, 2012.

[19] *xmlgen - The Benchmark Data Generator*. http://www.xml-benchmark.org/generator.html. Retrieved on January 20, 2012.

[20] T. Müldner, G. Leighton, and J. Diamond, "Using xml compression for www communication," in *Proceedings of the International Association for Developement of the Information Society (IADIS) International Conference WWW/Internet 2005 (ICWI 2005)*, 2005, pp. 459–466.

 ISBN: 978-1-61208-200-4