Efficient Replication of XML Documents with BLOB data

CS297 Report

by

Preethi Vishwanath

Advisor: Dr. Chris Pollett

Department of Computer Science

San Jose State University

December 2006

List of Figures and Tables

1. Introduction

With the advent of XML support in many databases that could be run in distributed mode, new issues in replication of XML data arise. For instance, if an XML document is used to manage several BLOBs, such as might occur in multi media applications, then in a distributed setting both the XML document and the BLOBs they contain might be replicated independently of each other. The goal of our project is to develop a replication aware algorithm for this kind of XML data and to provide an example implementation on an open source database such as Postgres.

XML (Extensible Markup Language) is a W3C initiative that provides a text-based mark-up language as a means to describe data. With XML, one can specify both the grammar of the language, in terms of new element definitions, and its use, usually in the form of tree-structured records of the new elements in the same document.

A distributed database is a database that presents a single logical view to its users, but one in which the underlying data is stored across multiple nodes that are connected to each other via some form of communication network.

Replication is a technique which is used in distributed databases to avoid data loss in case of a catastrophe by implementing fail-safe redundancy for records by maintaining an exact replica of part or entire database over multiple nodes. Replication also helps to reduce access times by distributing queries across these nodes that exploits certain database properties such as relative distance of the data node from the query client, current load on the database and the like.

Binary Large Object Binaries (BLOBS) are first class objects in database systems, much like strings (varchar) and integers (int) and is unstructured binary data stored as a single entity in a database management system [3]. This format is typically used to store streaming data like audios, videos in a database. With the advent of online video and audio websites (like Google Video, YouTube, Napster etc), the usage of BLOBS in databases is on the rise. To ensure high availability, these BLOBS are usually replicated across multiple nodes in a distributed fashion. Typically, some websites also offer playlists that are usually composed of multiple BLOBs stored in an XML document, which may also then be replicated. Clearly, there exists a need for a framework where it might be possible to merge these independent requirements into a more efficient collective framework.

During the course of the semester, I had the opportunity to investigate multiple of these issues, including examining current research, summarizing the same and formulating problem statements that will be eventually used for furthering research in development of such a framework. The rest of the document summarizes the problem statements we worked on during the course of the semester, along with our solutions to the same, and how they help in our thesis work. We finally conclude with a detailed proposal that weplan to work on in the next semester.

2. XML parsing in PostgreSQL

2.1 Goal

To allow for intelligent replication of BLOB data contained in an XML document in an open source database (e.g. Postgres), it is essential to add XML parsing functionality to allow for discovery of BLOB metadata in an XML document. Our first deliverable was to add XML parsing support to Postgres by implementing a User Defined Function (UDF). User Defined Functions are functions which permit users to add additional functionality in the form of extensions that can be called from regular SQL procedures to the database. These functions can be written in high level programming languages like C, C++, Perl, Java etc. These are compiled into a shared library (or jar files) and are loaded by the server upon first use.

The ability to parse an XML document and shred it into the database is necessary for us since most real world usages of BLOBs usually embed these within an XML document, which may contain, in addition to details about the BLOB (e.g. record name, size), the details of the replicas of the BLOB as well.

For this deliverable, our example XML document consists of multiple records of postal addresses, and our extension function shreds these records into individual elements of the postal address and inserts them into the database.

2.2 Design and Implementation

This deliverable [18] consists of the following parts

(a) parsing of the XML document,

(b) creation of the intermediate data representation and

(c) Shredding of the data into the database

2.2.1 XML Parser

The algorithm and implementation for parsing of the XML document has been borrowed from the open source C-XML parser, expat [2]. The parser provides for identification of XML elements and stubs for implementing callbacks when an XML element is encountered. In the code snippet below, the user implemented function callbacks *startElement* and *endElement* are invoked for every TAG (new element), while the function *charHandle* is invoked for the enclosed data

```
void create_xml_tree(char* fname, int size)
{
       char buf[BUFSIZ];
       int done;
       XML_Parser parser = XML_ParserCreate(NULL);
       XML_SetElementHandler(parser, startElement, endElement);
       XML_SetCharacterDataHandler(parser, charHandle);
       fname[size] = '\0';
       FILE *fp = fopen(fname, "r");
       do {
       size_t len = fread(buf, 1, sizeof(buf), fp);
       done = len < sizeof(buf);
       if (!XML_Parse(parser, buf, len, done))
              return;
       } while (!done);
       fclose(fp);
       XML_ParserFree(parser);
       return;
}
```

Figure 1: XML Parser Main Loop

2.2.2 Intermediate Representation

We have added stub implementations for *startElement, endElement and charHandle* that create an in-memory tree based representation of the XML document. For every XML element enclosed by tags, a structure of the form below is instantiated and added to the tree. At each nesting level in the document, the tree depth is increased, and the new element is added as a child to the current node being processed.

```
struct xml_node
{
        char* name;
        char* value;
        int depth;
        int nchilds;
        struct xml_node* prev;
        struct xml_node* child[MAX_CHILD];
};
```

Figure 2: An XML Element Representation (in an XML Tree)

In addition, for every leaf node (an address record), a context-sensitive structure of the form below is instantiated and added to a list of address records. This allows for easy traversal for a database-style record traversal. Below is the exact match for a database-style record traversal.

```
struct addr_rec
{
        char* fname;
        char* lname;
        char* street;
        char* state;
        char* country;
        struct addr_rec* next;
};
```

Figure 3: Contextual representation of XML Data

The code snippet below with the associated global variables create the in-memory tree based representation of the XML document, in addition to creating a context sensitive linked list (of address records)

```
struct xml_node* curr_node = NULL;
struct xml_node* head_node = NULL;
struct addr_rec* rec_list = NULL;
struct addr_rec* head_rec_list = NULL;

void startElement(void *userData, const char *name, const char **atts)
{
  struct xml_node* tmp_node = NULL;
  if (curr_node == NULL) {
       curr_node = create_node(name, 0);
       head_node = curr_node;
  } else {
       tmp_node = create_node(name, (curr_node->depth + 1));
       curr_node->child[curr_node->nchilds] = tmp_node;
       curr_node->nchilds ++;
       tmp_node->prev = curr_node;
       curr_node = tmp_node;
   }
}
void endElement(void *userData, const char *name)
{
       if (rec_list == NULL) {
              rec_list = create_addr_rec();
              head_rec_list = rec_list;
       }
       if (curr_node->nchilds == 0) {
              char* val = curr_node->value;
              init_rec(rec_list, val);
              if (rec_list->next == NULL) {
                     rec_list->next = create_addr_rec();
                     rec_list = rec_list->next;
                     rec_list->fname = val;
              }
       }
       curr_node = curr_node->prev;
}
void charHandle(void *userData,
              const XML_Char *s,
              int len)
{
       char* my_val;
       if (is_empty((const char *)s,len)) return;
       my_val = (char *)malloc((len+1)*sizeof(char));
       memcpy(my_val, s, len);
       my_val[len] = '\0';
       curr_node->value = my_val;
}
```

Figure 4: XML Parse Tree & Linked List (Intermediate Representation)

2.2.3 PostgreSQL extension

Finally, we have to invoke the XML parser functionality from within SQL statements executed on the command line. This consists of implementing an SQL based front end User Defined Function, and a backend responsible for interfacing with the XML parser

and returning the XML records. The backend is implemented as a Set Returning Function (SRF) that returns XML address records one record at a time.

The SQL front-end UDF is defined as below. Notice that the implementation function is included in the shared object libproj1.so and is implemented as get_xml_data

```
CREATE OR REPLACE FUNCTION get_xml_data(IN integer,
                                        IN text,
                                        OUT f6 VARCHAR,
                                        OUT f7 VARCHAR,
                                        OUT f8 VARCHAR,
                                        OUT f9 VARCHAR,
                                        OUT f10 VARCHAR) RETURNS SETOF record
AS '/home/hlthantr/proj_one/libproj1', 'get_xml_data'
LANGUAGE C IMMUTABLE STRICT;
```

Figure 5: SQL Invokable User Defined Function (PostgreSQL FrontEnd)

The function get_xml_data instantiates the XML parser and creates the intermediate representation after obtaining the name of the XML file as input. It returns one XML address record for every call made to it. This is achieved by traversing the address record linked list one XML address record at a time. Note the use of first call, and iterative manner of obtaining record like data from PostgreSQL.

```
Datum get_xml_data(PG_FUNCTION_ARGS)
{
       FuncCallContext *f_ctx;
       MemoryContext oldC;
       TupleDesc tup_desc;
       AttInMetadata *attinmeta;
       int call_cntr;
       int max_calls;
       struct addr_rec* xml_addr = NULL;
       text* t;
       int32 fsize;

       if (SRF_IS_FIRSTCALL()) {
              f_ctx = SRF_FIRSTCALL_INIT();
              oldC = MemoryContextSwitchTo(f_ctx->multi_call_memory_ctx);
              f_ctx->max_calls = PG_GETARG_UINT32(0);
              t = PG_GETARG_TEXT_P(1);
              fsize = VARSIZE(t) - VARHDRSZ;
              create_xml_tree((char *)VARDATA(t), fsize);
              attinmeta = TupleDescGetAttInMetadata(tup_desc);
              f_ctx->attinmeta = attinmeta;
              f_ctx->user_fctx = (void *)head_rec_list;

              MemoryContextSwitchTo(oldC);
```

```
        }
        f_ctx = SRF_PERCALL_SETUP();
        call_cntr = f_ctx->call_cntr;
        max_calls = f_ctx->max_calls;
        attinmeta = f_ctx->attinmeta;
        xml_addr = (struct addr_rec *)f_ctx->user_fctx;
        if (call_cntr < max_calls) {
               char **values;
               HeapTuple tuple;
               Datum result;
               values = (char **)palloc(5* sizeof(char *));
               update_values(xml_addr,values);
               tuple = BuildTupleFromCStrings(attinmeta, values);
               result = HeapTupleGetDatum(tuple);
               f_ctx->user_fctx = (void *)(xml_addr->next);
               SRF_RETURN_NEXT(f_ctx, result);

        } else {

               SRF_RETURN_DONE(f_ctx);
        }
}
```

Figure 6: Implementation of User Defined Function (PostgreSQL C Backend)

2.2.4 Test Files

The following XML document was used for testing the newly implemented XML parser that was installed as a shared object at /home/prithari/proj_one/libproj1

```
<address-book>
        <entry>
               <person>
                      <first>Basil</first>
                      <last>Elton</last>
               </person>
               <street>North Point Lighthouse</street>
               <state>CA</state>
               <country>USA</country>
        </entry>
        <entry>
               <person>
                      <first>Elton</first>
                      <last>John</last>
               </person>
               <street>Monroe Street</street>
               <state>NY</state>
               <country>USA</country>
        </entry>
</address-book>
```

Figure 7: An example test XML file

The following composite SQL script was used to test the XML parser functionality

```
DROP FUNCTION get_xml_data(integer, text);
CREATE OR REPLACE FUNCTION get_xml_data(IN integer,
                                IN text,
                                OUT f6 VARCHAR,
                                OUT f7 VARCHAR,
                                OUT f8 VARCHAR,
                                OUT f9 VARCHAR,
                                OUT f10 VARCHAR) RETURNS SETOF record
            AS '/home/prithari/proj_one/libproj1', 'get_xml_data'
            LANGUAGE C IMMUTABLE STRICT;
DROP TABLE tblP;
CREATE TABLE tblP(fname VARCHAR, lname VARCHAR, saddr VARCHAR, state VARCHAR,
cntry VARCHAR);
INSERT INTO tblP VALUES
                    ((get_xml_data(4,'/home/prithari/proj_one/test.xml')).f6,
                     (get_xml_data(4,'/home/prithari/proj_one/test.xml')).f7,
                     (get_xml_data(4,'/home/prithari/proj_one/test.xml')).f8,
                     (get_xml_data(4,'/home/prithari/proj_one/test.xml')).f9,
                     (get_xml_data(4,'/home/prithari/proj_one/test.xml')).f10
                     );
SELECT * FROM tblP;
```

Figure 8: SQL Script for testing

2.2.5 Output

The log below represents the output (truncated) log from running proj1.sql

```
$ psql testdb

testdb=# \i proj1.sql
       DROP FUNCTION
       CREATE FUNCTION
       DROP TABLE
       CREATE TABLE
       INSERT 0 4

         fname  |  lname   |         saddr          |  state  | cntry
        --------+----------+------------------------+---------+-------
         Basil  | Elton    | North Point Lighthouse | CA      | USA
         Elton  | John     | Monroe Street          | NY      | USA
         John   | Grisham  | bailey Street          | England | UK
         John R. | Legrasse | 121 Bienville St.      | LA      | USA
        (4 rows)
```

Figure 9: Output from SQL Script(Truncated)

As observed in the above example, the data (addresses) from the XML file have been shredded and inserted to the table

3. A Distributed Byzantine Algorithm

3.1 Goal

One of our goals is to determine an efficient mechanism to access replicated data that exists across different machines. One of the common problems is to decide which of the replicated copies to access. This can be achieved by evolving a distributed consensus, where the node in question participates in a "voting" process with several other nodes, some of which might be faulty, and all agree on a solution if a certain number of voters (7/8 in our case) agree.

3.2 Design and Implementation

To implement this algorithm, we have adapted the solution for Byzantine agreement on a single node and extended it for multiple machines using Java RMI. The basic version relies on each "voter" being a subclass of the javax.swing.Timer that is instantiated with a configurable delay at the end of which each voter casts his vote for this round, and polls votes received from all other voters in the previous round to see if 7/8 voters agree. To allow for the program to converge, a biasing element is introduced that biases this voter in favor of the majority (5/8) for each successive round. Two of the voters are configured as faulty, allowing them to change their vote completely randomly, even within the same round.

To extend this implementation for our purpose, we have used Java RMI methods to create voters, and for voters to poll each other. The main Manager object is responsible for creating voters, and for polling each voter to check if they have individually arrived at an agreement. The Manager exits the program when all voters have reached a Byzantine agreement

The Manager obtains a reference to a VoterManager which defines a manager interface that allows for instantiation of voters. Java RMI allows for the calling program to be completely unaware of whether the instantiation occurs locally or on a remote node. Note that the interface extends java.rmi.Remote, which is necessary for object references to be serializable, and hence be marshalled.

```
interface VoterManager extends java.rmi.Remote
{
        public Voter createVoter(int delay, boolean flty, String db,
                                 String user, String passwd, String tblName)
        throws java.rmi.RemoteException, ClassNotFoundException,
        java.sql.SQLException;
}
```

Figure 10: Interface definition for a VoterManager that creates voters on a host

The VoterManager supplies each voter it creates with information regarding delay, the nature (faulty/non-faulty), and database related information. The VoterManagerImpl provides an implementation of the above interface. Each voter, in turn, has a well defined interface that is used by other voters to gather votes. In addition, each voter also provides methods for the Manager to initialize, start, and check for Byzantine completion. The interface definition below summarizes the methods a Byzantine voter must implement

```
import java.util.*;
interface Voter extends java.rmi.Remote
{
        public void initialize(ArrayList list)
                throws java.rmi.RemoteException,
                        java.sql.SQLException,
                        ClassNotFoundException;
        public void start() throws java.rmi.RemoteException;
        public boolean isDone() throws java.rmi.RemoteException;
        public void receiveVote(int i, Voter v)
                throws java.rmi.RemoteException,
                ClassNotFoundException,
                java.sql.SQLException;
        public int getRoundNumber() throws java.rmi.RemoteException;
        public int  getDecision() throws java.rmi.RemoteException;
}
```

Figure 11: Interface definition for each Voter.

The initialize(), start(), isDone() methods are used by the Manager to initialize voter objects that have been created, to start the Byzantine algorithm on each voter and to check for local Byzantine completion on each node respectively.

The voters also vote and poll each other to check on agreement for a 0/1 value for a single variable in a Byzantine fashion. The public methods getRoundNumber, receiveVote and getDecision are meant for communication among voters to exchange individual status. We use JDBC methods to access the PostgreSQL DB that acts as the backing store to keep every vote received in every round. Every tuple in our relation is of the form <voterID, roundNumber, numVotes, headCnt, tailCnt>. Byzantine Agreement is observed on each node independently, when ALL votes in a round have been received AND 7/8 voters agree on the value of the vote.

The program [19] can be configured to run on multiple nodes, with multiple voters/node, and some configurable number of faulty nodes.

3.3 Output

In each round, every voter casts a vote. If 7/8 of his votes agree, he reaches a decision., i.e. never changes his vote for future rounds. Once all the voters reach a decision the Byzantine Agreement is reached. Truncated output for the final round is shown below.

```
Voter 0  :       Round : 2       Decision : 0
Voter 1  :       Round : 2       Decision : 0
Voter 2  :       Round : 3       Decision : 0
Voter 3  :       Round : 2       Decision : 0
Voter 4  :       Round : 2       Decision : 0
Voter 5  :       Round : 2       Decision : 0
Voter 6  :       Round : 2       Decision : 0
Voter 7  :       Round : 2       Decision : 0
```

Figure 12: Output (Truncated) from Byzantine agreement on a boolean variable

In round 4 Byzantine Agreement is reached, since all voters have a decision by round 3.

4. Extending the Byzantine Algorithm for multiple BLOBS

4.1 Goal

Previously, we implemented a Distributed Byzantine implementation where all the nodes in question arrived at a decision for a Boolean valued data. In a real world scenario, such an assumption is too simplistic. In this project, we solve the real-world problem of determining the most suitable machine in which to replicate a BLOB, where the answer is arrived at by distributed consensus between multiple voters. Consider that there are a set of BLOBS BSET = {B1, B2, B3, B4, B5, B6} and a set of machines MSET = {M1, M2, M3}, and a set of Voters VSET = {V1, V2, V3, V4 } each with partial interest in some of the BLOBs in B, and an opinion on where the BLOB should be replicated. Let's further assume that each of the voters in VSET represent this opinion in the form of an XML

document, then the composite problem statement would be to uniquely determine, through consensus, a machine M that belongs to MSET for every blob B that belongs to BSET. We further introduce the notion of an "idle" voter, who votes randomly for any BLOB that he has no knowledge about (i.e. not present in his XML document). This voter, however, adopts the majority opinion for any BLOB as soon as 3/4 voters agree.

The Byzantine problem statement is formulated as one of arriving at a consensus among a set of voters on one of several machines as the choice for replicating each BLOB in the set.

4.2 Design and Implementation

For the implementation of this algorithm, we extensively used our previous deliverables, the first deliverable record structures were changed for the XML representation of voter opinion. An example voter opinion in XML form is below

```
<blob-details>
        <machine_code>0</machine_code>
        <blob>0</blob>
         <machine_code>7</machine_code>
        <blob>2</blob>
        <machine_code>7</machine_code>
        <blob>3</blob>
</blob-details>
```

Figure 13: BLOB insight available to a particular voter (in an XML form)

For this voter, the following table summarizes his opinion about different blobs.

BLOBID	Machine
0	0
1	X
2	7
3	7

Table 1: BLOB to Machine Mapping for an individual voter

For BLOBID 1, this voter does not have a vote and will behave like an IDLE voter. Most of the basic algorithms did not change in this deliverable [20], except for additional logic that got added to the top level Manager, first for accessing the BLOB->machine mapping, and then to obtain distributed consensus between all voters one BLOB at a time. Some snippets of the code below illustrate these changes.

```
sqlText = "SELECT machine_code, blob_id FROM " + tblConfig
+ " WHERE voter_id = " + idx;
ResultSet rs= sql.executeQuery(sqlText);
while (rs.next())
{
       String mName = rs.getString(1);
       String mBlob = rs.getString(2);
       Integer m_Name = new Integer(Integer.parseInt(mName));
       Integer m_Blob = new Integer(Integer.parseInt(mBlob));
       ht_blobMachine.put(m_Blob, m_Name);
if(!nMachines.contains(m_Name))
{
          nMachines.add(m_Name);
       }
       if (!nBlobs.contains(m_Blob))
{
          nBlobs.add(m_Blob);
       }
 }
 rs.close();
```

Figure 14: Populating the Blob->Machine ID for every voter

```
for (int i = 0; i < nBlobs.size(); i ++)
{
      // Initialize Voters to start for Blob #i
      initializeVoters(vList, nMachines.size(),i);
      startVoters(vList);
      while (true)
      {
      if (voterDone(vList))
      {
           System.out.println("!!!BYZANTINE REACHED FOR BLOB #" + i + "!!!");
           break;
       }
 }
}
```

Figure 15: Manager Iterating over all BLOBS for distributed consensus

There were a few more changes made to the Byzantine algorithm, mostly to deal with keeping counts for each machine and introducing the idea of an IDLE voter, but overall, the structure or basic consensus algorithm did not change much.

4.3 Output

Byzantine algorithm was run to find an agreement among every voter for every BLOB on the machine to go to for a particular BLOB. As is evident, this algorithm takes much longer to converge than a simple agreement on a Boolean variable like the one before. There are 8 set of possible values [for 8 machines], there is a notion of idle voters that act as random voters for BLOBS that they do not have a mapping for initially, and there are voters who are faulty. All these leads to very large convergence times as can be seen in the truncated output below

```
VTR[4]:VOTE:[0]:RND[61]
Voter 4  Agreement Reached!! Agreement is 0
VTR[2]:VOTE:[1]:RND[62]
Voter 2  Agreement Reached!! Agreement is 0
VTR[0]:VOTE:[0]:RND[62]
Voter 0  Agreement Reached!! Agreement is 0
VTR[1]:VOTE:[0]:RND[62]
Voter 1  Agreement Reached!! Agreement is 0
VTR[6]:VOTE:[0]:RND[62]
Voter 6  Agreement Reached!! Agreement is 0
VTR[5]:VOTE:[0]:RND[62]
Voter 5  Agreement Reached!! Agreement is 0
VTR[3]:VOTE:[0]:RND[62]
Voter 3  Agreement Reached!! Agreement is 0
VTR[7]:VOTE:[5]:RND[63]
Voter 7  Agreement Reached!! Agreement is 0
!!!BYZANTINE REACHED FOR BLOB #0!!!
VTR[1]:VOTE:[6]:RND[14]
Voter 1  Agreement Reached!! Agreement is 6
VTR[6]:VOTE:[6]:RND[14]
Voter 6  Agreement Reached!! Agreement is 6
VTR[5]:VOTE:[6]:RND[14]
Voter 5  Agreement Reached!! Agreement is 6
VTR[3]:VOTE:[6]:RND[14]
Voter 3  Agreement Reached!! Agreement is 6
VTR[7]:VOTE:[2]:RND[15]
Voter 7  Agreement Reached!! Agreement is 6
VTR[2]:VOTE:[6]:RND[15]
Voter 2  Agreement Reached!! Agreement is 6
VTR[4]:VOTE:[6]:RND[14]
Voter 4  Agreement Reached!! Agreement is 6
VTR[0]:VOTE:[6]:RND[15]
Voter 0  Agreement Reached!! Agreement is 6
!!!BYZANTINE REACHED FOR BLOB #1!!!
VTR[3]:VOTE:[3]:RND[55]
Voter 3  Agreement Reached!! Agreement is 3
VTR[1]:VOTE:[3]:RND[55]
Voter 1  Agreement Reached!! Agreement is 3
VTR[6]:VOTE:[3]:RND[55]
Voter 6  Agreement Reached!! Agreement is 3
VTR[7]:VOTE:[3]:RND[56]
Voter 7  Agreement Reached!! Agreement is 3
VTR[2]:VOTE:[1]:RND[56]
Voter 2  Agreement Reached!! Agreement is 3
```

```
VTR[4]:VOTE:[3]:RND[55]
Voter 4  Agreement Reached!! Agreement is 3
VTR[5]:VOTE:[3]:RND[56]
Voter 5  Agreement Reached!! Agreement is 3
VTR[0]:VOTE:[3]:RND[56]
Voter 0  Agreement Reached!! Agreement is 3
!!!BYZANTINE REACHED FOR BLOB #2!!!
VTR[1]:VOTE:[7]:RND[15]
Voter 1  Agreement Reached!! Agreement is 7
VTR[6]:VOTE:[7]:RND[15]
Voter 6  Agreement Reached!! Agreement is 7
VTR[5]:VOTE:[7]:RND[15]
Voter 5  Agreement Reached!! Agreement is 7
VTR[0]:VOTE:[7]:RND[15]
Voter 0  Agreement Reached!! Agreement is 7
VTR[2]:VOTE:[4]:RND[16]
Voter 2  Agreement Reached!! Agreement is 7
VTR[7]:VOTE:[4]:RND[16]
Voter 7  Agreement Reached!! Agreement is 7
VTR[4]:VOTE:[7]:RND[15]
Voter 4  Agreement Reached!! Agreement is 7
VTR[3]:VOTE:[7]:RND[16]
Voter 3  Agreement Reached!! Agreement is 7
!!!BYZANTINE REACHED FOR BLOB #3!!!
....Done
```

Figure 16: Output (Truncated) of the Byzantine algorithm for BLOB mapping

5. Future Work

My research will focus on algorithms to improve the replication designs of Binary Large Objects that are identified by an XML document

With the explosion of applications which make use of streaming media content to be delivered directly into intelligent devices (e.g. Cell phones, laptops, PCs, handhelds), content that is usually composed in the form of playlists (XML documents), there is renewed interest in delivery mechanisms for this content.

To simplify the understanding of our setup let us take a scenario of multiple Zune devices, a Microsoft music player, within a network. These devices can communicate with each other and exchange music. There are two alternatives to obtain music: They can purchase music or they could "squirt" via bluetooth music from another Zune device in their network. However, there is one limitation when you squirt music (data); namely, users can play it only a limited number of times and after that they need to squirt the music again. Alternatively some users may even choose to just purchase their favorite music.

This results in three major issues coming forth:

1. There is a considerable amount of bandwidth consumed each time data is squirted
2. Each time a permanent copy of music is created on the new machine, there is a monetary cost from the download service.
3. There may be no device with a permanent copy of the music needed within squirting distance.

The goal of our CS298 is to be able to create an ad-hoc network which minimizes the bandwidth and monetary cost and maximizes the availability of the desired data. This might be achieved by judiciously replicating a copy of this data on some other Zune device which not only is also looking for the same data but also is located at a distance farther away from the original device.

We will attempt to implement our scheme for two difference network models:

1. Token Ring Network
2. Internet Protocol (IP) Model

Work performed this semester has enabled us to create software infrastructure that will help us in our thesis work, e.g. in parsing XML documents in the database backend, a consensual algorithm among multiple entities etc.

Our model consists of a bunch of Voters and a bunch of BLOBs, located on various machine. Each voter votes for the BLOB that he requires, count for each BLOB is determined by the number of times a BLOB is “voted for” by a voter * the Distance at which the BLOB is . Once all the voters complete their votes on the BLOB that they require a determination is made of the most commonly used BLOBs which are not easily accessible.

Byzantine agreement is performed among the various machines to determine which, machine the BLOB should be moved to, so that all the machines which are not located closer to the original data can easily access the same.

6. References

[1] Principles of Distributed Database Systems (2nd edition). M. Tamer Ozsu and Patrick Valduriez. Prentice Hall. 1999

[2] “Using Expat”, http://www.xml.com/pub/a/1999/09/expat/index.html

[3] Distributed Algorithms (The Morgan Kaufmann Series in Data Management Systems). Nancy A. Lynch. Morgan Kaufmann Publishers. 1996

[4] Dynamic XML Documents with distribution and replication. Serge Abiteboul, Angela Bonifati, Gregory Cobena, Ioana Manolescu, Tova Milo. Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM Press. 2003. Pages 527-538

[5] Database replication techniques: a three parameter classification. M Wiesmann, F Pedone, A Schiper, B Kemme, G Alonso. Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems. IEEE Computer Society. 2000. Page 206

[6] T. Hara, ``Effective replica allocation in ad hoc networks for improving data accessibility," Proc. IEEE Infocom 2001, pp.1568-1576, 2001.

[7] T. Hara, ``Replica allocation methods in ad hoc networks with data update," ACM-Kluwer Journal on Mobile Networks and Applications, Vol.8, No.4, pp.343-354, 2003.

[8] T. Hara and S.K. Madria, ``Dynamic data replication schemes for mobile ad-hoc network based on aperiodic updates," Proc. Int'l Conf. on Database Systems for Advanced Applications (DASFAA 2004), pp.869-881, 2004.

[9] T. Hara, N. Murakami, and S. Nishio: ``Replica Allocation for Correlated Data Items in Ad-Hoc Sensor Networks," ACM SIGMOD Record, Vol.33, No.1, pp.38-43, 2004.

[10] H. Hayashi, T. Hara, and S. Nishio, ``Cache Invalidation for Updated Data in Ad Hoc Net-works," Proc. Int'l Conf. on Cooperative Information Systems (CoopIS'03), pp.516-535, 2003.

[11] G. Cao, L. Yin, C.R. Das, ``Cooperative Cache-Based Data Access in Ad Hoc Networks," IEEE Computer Magagine, Vol.37, No.2, pp. 32-39, 2004.

[12] L.D. Fife and L. Gruenwald, ``Research issues for data communication in mobile ad-hoc network database systems," ACM SIGMOD Record, Vol.32, No.2, pp.42-47, 2003.

[13] G. Karumanchi, S. Muralidharan, and R. Prakash, ``Information dissemination in partitionable mobile ad hoc networks," Proc. Symposium on Reliable Distributed Systems (SRDS'99), pp.4-13, 1999.

[14] J. Luo, J.P. Hubaux, and P. Eugster, ``PAN: Providing reliable storage in mobile ad hoc networks with probabilistic quorum systems," Proc. ACM MobiHoc 2003, pp.1-12, 2003.

[15] K. Rothermel, C. Becker, and J. Hahner, ``Consistent update diffusion in mobile ad hoc networks," Technical Report 2002/04, Computer Science Department, University of Stuttgart, 2002.

[16] F. Sailhan and V. Issarny, ``Cooperative caching in ad hoc networks," Proc. Int'l Conf. on Mobile Data Management (MDM'03), pp.13-28, 2003.

[17] K. Wang and B. Li, ``Efficient and guaranteed service coverage in partitionable mobile ad-hoc networks," Proc. IEEE Infocom'02, Vol.2, pp.1089-1098, 2002.

[18] , Preethi Vishwanath, “Extending PostgreSQL for XML Processing”, http://www.cs.sjsu.edu/faculty/pollett/masters/Semesters/Fall06/Preethi/index.shtml?Del1.html, 2006.

[19] Preethi Vishwanath, “Implementation of Byzantine Algorithm on Distributed Databases”,http://www.cs.sjsu.edu/faculty/pollett/masters/Semesters/Fall06/Preethi/index.shtml?Del2.html, 2006.

[20] Preethi Vishwanath, “Extending Byzantine Agreement for Multiple BLOBs”, http://www.cs.sjsu.edu/faculty/pollett/masters/Semesters/Fall06/Preethi/index.shtml?Del3.html, 2006