A Voting Scheme for BLOB Replication

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirement for the

Degree

Master of Science

By

Preethi Vishwanath

May 2007

© 2007

Preethi Vishwanath

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Chris Pollett

Dr. Robert Chun

Mr. Tuong Truong

Abstract

The introduction of handheld devices has thrown up interesting research issues in the areas of BLOB replication in highly mobile ad-hoc wireless networks. One of such handheld device, Microsoft's Zune MP3 music player not only acts as a warehouse (30G) of audio and video data, but also exploits its in-build Wi-Fi capability to allow for exchange of these streams between any two players in close vicinity to each other. Our work proposes the use of a distributed, topologically aware Byzantine algorithm for creating BLOB replicas in the ad-hoc network. We also provide simulation results of our algorithms under different topologies and replicas created from our test bench. We also compare our work against other [8] [18] algorithms for BLOB replication in ad-hoc wireless networks.

Keywords: Ad-hoc networks, replication, Binary Large Objects, Byzantine

Table of Contents

List of Figures

List of Tables

1 Introduction

The last few years has seen a rapid increase in the usage of light-weight highly mobile intelligent devices (example. Cell phones, laptops, PDAs, MP3 players). These devices, in addition to serving their primary purpose as a compute/communication/entertainment device, have also enabled other previously unforeseen usage models. For example, Microsoft's Zune [1] MP 3 player is being marketed as a device that enables its users to receive songs from other neighboring Zune players (within 30 ft) through its WiFi interface. Users of these devices may export their songs and playlists to other interested Zune users who may play them up to three times or for up to three days. As it stands Zune does not support repeated squirting of songs. If this were allowed and if the ability to forward peer requests were also allowed, then these devices could act as forwarding nodes (routers), either to display playlists/music files from their immediate neighbors or to relay requests and responses for songs of interests between two neighbors. The Zune setup above is thus an example of a MANET.

Such a network, composed entirely of mobile hosts is termed as a mobile ad-hoc network [3] or MANET. A MANET differs from conventional mobile networks in that it is completely constructed out of mobile nodes, is fairly short-lived, and all nodes in the network have forwarding capability that allows nodes not within radio range of each other to communicate through one or more intermediate mobile nodes. In MANETs, since the host nodes move quite frequently, the network topology keeps changing, and can also become completely disjoint. The Zune setup above is thus an example of a MANET.

One of the approaches for users associated with Zune devices is to purchase a copy of the song. However, this approach has monetary expense associated with it. Our attempt is to reduce this cost without causing severe impact to the bandwidth associated with it. In addition given the resource constraints in the MANET, where most nodes are small form factor devices (example. PDA, cell phone, MP3 players) with little memory resources, a *replicate-all* strategy will not be the most optimal. A lot of data items (for example. BLOBS) stored in nodes are not directly exposed to other nodes, but through documents that express some form of correlation [7] between certain data items, for example, play lists in an MP3 player. The application software on these devices might itself impose the limitation that a node should either replicate the entire play list or none at all. This may lead to completely inefficient utilization of underlying memory resources where-in a node may be forced to keep copies of multiple data items that it has no use for simply because it requested a hugely popular music song.

Our proof-of-concept is a Java RMI based framework that enables BLOB replication studies in a MANET like distributed systems environment. We further utilize this framework and propose the use of a distributed Byzantine voting algorithm that allows multiple nodes to agree on a common node for replication for every data item of interest.

We also propose addition of biases to the Byzantine process that biases votes from different nodes based on network conditions in the MANET. For example, during the voting process a node that is part of multiple networks is preferred over a node with a single neighbor for purposes of replication. Similarly, distances, and relative frequencies of access of BLOBS are also considered.

The principal contribution of our work is in establishing a methodology and framework for conducting research on replication strategies for composite data (like BLOBS) in an MANET like distributed systems environment. We have also investigated a few different mechanisms for intelligent replication of BLOB data, by allowing for use of network biases in individual hosts that then collectively drive the decision on replication. Extension of our work to a Tree model, which performs a depth based Byzantine Agreement, has ensured that the replica is generated at a closer distance, even in scenarios where it is difficult to arrive at a consensus on which node to replicate in case of a ring model. We compare our replication algorithm with the Continuous Broadcast (CB) [8], Static Access Frequency (SAF) [18], Dynamic Access Frequency and Neighborhood (DAFN) [18] for bandwidth and cost savings.

This report has been organized into various sections which work their way to explain the concept in more detail. Chapter 2 provides us with background information while Chapter 3 talks about the various replication methods that we intend to compare our replication approach against. Chapters 4 and 5 provide a brief explanation of our design and the experimental setup. Chapter 6 and chapter 7 discuss the results and compares them with other replication models. Chapter 8 concludes our work and provides ideas for possible extensions.

2 Background and Theory

In this chapter, we examine prior work in the area of mobile ad-hoc networks, touch upon the basics of Byzantine algorithms and provide a brief introduction to XML and Java RMI, the basic building blocks in our framework.

2.1 Mobile ad-hoc networks

With the increase in the number of handheld devices with Wireless capabilities, and the fast adoption of usage models like text messaging, music sharing among these devices, there is renewed interest in research in the area of mobile ad-hoc networks. Prior work in the area of Mobile ad-hoc networks has focused on improving network connectivity in order to ensure accessibility to all nodes. Most schemes proposed have been based on some sort of a prediction scheme [6] that relies on estimating when a certain wireless link is *likely* to go down and take corrective action prior to the event.

Unfortunately, very little research has focused on *continued data availability* in the presence of dynamic network changes affecting continued network connectivity, which is the prime focus of our work. One mechanism to ensure continued data availability at all nodes is to *replicate* all the data on all nodes in the MANET. Such a scheme would make the issue of network connectivity rather redundant, since data is broadcast from every node from time to time to ensure that it is always available to all nodes as they enter or leave the network. In addition, any updates on any of the nodes are continually broadcast to ensure *update consistency.* Such an approach however is completely infeasible. There are resource constraints on each node that prevent mindless replication, the bandwidth requirements for such a scheme would stress the overall network bandwidth, and update consistency is harder to maintain when several nodes work on the same data item simultaneously. Initial work on replica creation in mobile systems applied the traditional three-tiered Client Server traditional database replicate methodology wherein one node is classified as the master for any single piece of data and all queries in the absence of replicates are directed to the master. There is also the assumption of a single warehouse that is accessible from all hosts at any given point in time. As more requests are made to the same piece of data, replicas are created based on frequencies of access and available storage. Metadata information about replica accessibility is then passed between different nodes to ensure accessibility to all data from all nodes. This scheme, while effective in restricting resource usage, fails in the basic premise of mobile ad-hoc networks which is that *all nodes* are mobile, including the node hosting the database, hence disconnects between the clients and the main database is possible.

Since our focus is not on data that is *continually* updated, but on read-only data like music, we will not be examining consistency issues among replicas, which is the main focus in other studies [17]. We will be looking at other replication strategies whose principal motive is to ensure accessibility of data to all nodes in the MANET, and have some sort of heuristic regarding the choice of node on which to replicate [7] [8].

We will in the following chapter examine some of the replica allocation methods [8], and propose our framework for mechanisms for replicating BLOBS on different nodes in an MANET.

2.2 XML

XML (Extensible Markup Language) is a W3C initiative that provides a text-based markup 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 in the same document. The structure of an XML document represents a tree form, example a postal address element may have as its sub children the different components that represent it, example the name, street, city, state and zip as elements. This expressiveness of XML lends itself as an effective mechanism for information interchange between different components in a software system.

In most handheld devices, data (example play lists) are described in an XML format. A very simple XML document describing a song play list, for example, would usually consist for each song element in the document, the name of the singer, and a pointer to the local table from where to obtain the song from.

Since XML is the accepted mechanism for information interchange between nodes in the MANET, it is essential that any test bed implementation for MANET also have an XML parsing and processing front end engine.

We will describe our XML front end in later chapters as part of our overall design.

2.3 Byzantine algorithm

A Byzantine agreement is a fairly well researched problem in distributed systems where multiple entities, some of which may fail in arbitrary ways, have to agree on a common “value” for a variable. A Byzantine agreement relies on the set of well behaved components to arrive at a common consensus for the value of any variable and hence keep the overall system reliable.

We believe that Byzantine algorithms are very relevant to the field of MANETs where a node might suddenly stop responding, or may start acting in a very unreliable manner (for example providing different values for the same query to different agents), given the meager amount of compute/memory resources available to it.

We propose use of the Byzantine voting process biased based on certain network topology parameters for the purposes of determining which BLOB to replicate on what node. In future chapters, we provide a more concrete description of the application of the Byzantine algorithm to our design.

2.4 Java RMI

Java Remote Method Invocation [20] infrastructure is often used as the basis to create client-server infrastructures using the Java programming language. RMI allows the programmer to be completely oblivious to the underlying network, and presents exactly the same programming paradigms, in terms of object access and method invocations on these remote objects as local objects. The Java Runtime is responsible for converting object arguments for any of these methods into a byte stream, packaging them across to the remote host providing the service and then reconstructing these objects on the remote host before invoking the stub function that then actually calls on the actual object method passing it the composite reconstructed object. A naming service (rmiregistry) needs to be run on any host exporting objects and methods for remote access, so as to allow other Java runtimes to query and obtain information about these objects.

We have chosen to use the Java RMI framework since it is the most suitable for modeling an MANET like environment. The RMI framework allows us to be completely oblivious to the underlying mechanisms (example sockets) for transport of data, and instead spend most of our effort on algorithm development for blob replication in MANETs.

The discussion of the design and implementation of various classes in our Java framework test bed will be done in the later chapters.

3 Replica Allocation Methods

In this chapter we first propose our algorithm for replica allocation in mobile ad-hoc networks. We then qualitatively compare our algorithm with some of the other [8] [18] algorithms for replica allocation in MANETs.

3.1 Byzantine replication

In our opinion, the fundamental problem in an MANET that does not deal with replicate consistency is the fact that a node cannot rely on the presence of any other node in the MANET. Nodes drop in-and-out of the network frequently, and accessibility to any of the BLOBS can be lost at any time.

The usual [12] Byzantine agreement algorithm is as follows.

```
/* For each BLOB being replicated */

1. Initial vote cast (self machine)

2. Each node

   2.1  If node interested in the BLOB
        a. Tabulate votes
        b. Toss a coin
           If heads
               Change vote for next round if more than
               5/8 of the nodes agree on a common
               machine, else agree on a default node.
           Else
               Change vote for next round if more than
               6/8 of the nodes agree on a common
               machine else agree on a default node.

   2.2 If node not interested in the BLOB
       Convert vote to majority vote

   2.3 If node faulty
           Randomly cast a vote.

3. Consensus = 7/8 of the voters agree on the same machine for replication
```

We propose an algorithm that uses a modified Byzantine voting process among nodes in the MANET for deciding where to place the replica. We extend the regular Byzantine algorithm by adding *biasing* to the voting process where-in the probability for the selection of a node as a replica is directly proportional to its *network hop distance* from the original node. The *network hop distance* is defined based upon the connectivity graph between nodes at any instance of time. If nodes are close to each other, as defined by direct connectivity between each other (i.e. approximately the same radio distance), then they have a network hop distance of zero. We term the collection of all nodes that have a network hop distance of zero from each other as being in a *sub mesh.* The network distance then is the distance between any two sub meshes. Hence, the farther away a requesting node is, the greater is the probability of choosing the sub mesh as the replication node. The key insight in choosing such an approach is the thinking that nodes farther away are more susceptible to be dropping out of the MANET than a node closer to the original node. The *bias* factor is introduced in the voting algorithm based on the product of the hop distance with the relative access frequency for that node for the BLOB data in question.

As observed in the above algorithm, a Byzantine Decision is only reached when either 5/8 or 6/8 voters come on a consensus. Similarly a Byzantine Agreement is only reached when 7/8 voters come to a consensus. Thus, only a BLOB with users having a very high access rate and located at considerable distance would account in replication.

Our approach also restricts the number of replicas of any BLOB to be exactly one. We believe that given the resource constraints in most nodes, this is a good optimization, especially whenever the number of BLOBS of interest approaches the overall capacity of the nodes constituting the MANET.

Finally, our approach places an upper bound on the number of BLOBS to replicate based on the overall access frequencies to the BLOB.

We have designed a test bench that implements our algorithm on a Java RMI based framework. The framework uses various Object oriented design principles and is easily extensible to allow for studies of different MANET network topologies, and other algorithms for BLOB replication.

A high level description of the execution of our testing framework is as follows:

Phase 0:

```
    1. Parse XML Store consisting details of the BLOBs accessed and the machines on
       which they are located.
```

Phase 1:

```
    2. Loop over  num_trials
    3. Randomly pick a voter
            a. Look at the voter's playlist.
            b. For each BLOB in the playlist
                    i. Add to each BLOB in the playlist distance from voter to BLOB.
/* At this point every voter has the information for each BLOB. */
```

Phase 2:

4. Do we replicate?
    a. Check if BLOB has more than α fraction of the total accesses. If so, move to Phase 3, if not move to the next BLOB

Phase 3:

5. Replicate BLOB. Choose machine to replicate onto by performing Byzantine Agreement across a distributed network.

Phase 4:

6. Run simulator on the output generated to calculate and display new bandwidth access by each voter for the particular BLOB.

Bandwidth is calculated as (access frequency) * distance.

The complete software design for our test bed that also implements our algorithm for replication is presented in the next chapter.

We next describe some of the other approaches suggested in literature for BLOB replication in MANETs and compare and contrast our approach against those.

3.2 Constant Broadcast Replication

Constant Broadcast (CB) replication [18] is a strategy that relies on the servers in the mobile network to continuously broadcast their values at every clock tick. Since the delivery mechanism is wireless, a broadcast based strategy is useful since it reduces the overall bandwidth requirements as might be necessitated in unicast communication. The task of choosing the blob to replicate is left to each node as it receives the broadcast value. The node may choose to entirely ignore the broadcast if it has no memory resources, or may choose to replace the least recently BLOB in favor of the new one. The CB replication does not suggest either alternative. It only proposes a mechanism that ensures a high probability of all BLOBS being available even if some of the nodes drop off from the MANET.

The CB replication strategy, however does not apply any *intelligence* to optimizing the overall available space in all nodes on the MANET. Locally, unless nodes keep local history of access to different BLOBS, it may always be the case that at every tick, the least recently used is discarded in favor of the new BLOB that arrives. If the node performs *strided* access to BLOBS, a simple LRU policy with no history of access may always result in the node losing the most popular BLOB all the time.

A high level description of the execution of our testing framework is as follows:

Phase 0:

1. Parse XML Store consisting details of the BLOBs accessed and the machines on which they are located.

Phase 1:

```
2. Loop over  num_trials
3. Randomly pick a voter
        a. Look at the voter's playlist.
        b. For each BLOB in the playlist
                i. If (blob_id not present in blob_list)
                   {/* Add to blob_list */}
               ii. Else continue.
```

Phase 2:

```
4. Replicate to all machines
```

Phase 3:

```
5. Do we accept?
        a. Check whether memory available on machine.
                i. If yes , accept replicated copy
               ii. else reject
   /* However, for our analysis we assume that all the machines have enough memory
   space and hence do accept all the BLOBs replicated */
```

Phase 4:

```
6. Run simulator on the output generated to calculate and display new bandwidth
   access by each voter for the particular BLOB.

   Bandwidth is calculated as (access frequency * distance).
```

3.3 Static Access Frequency Replication

In the Static Access Frequency Replication method [8], every node allocates replicas during a specific period only (relocation period). Replicas are allocated based on the access frequency of that node to the BLOB in question for the immediately previous interval, and the availability of resources.

SAF replication method overcomes the limitations of CB replication by maintaining access frequencies of every node to every BLOB. However, it is a bit too conservative in not exploiting knowledge of the network topology when creating replicas. This solution is not resource optimal for the overall network, and in the worst case scenario, the same BLOB might be replicated at all nodes, if the access to the BLOB is the highest at all nodes. In such a case, the SAF replication method is equivalent to the *replicate-all* strategy, and hence is not a good choice for minimizing resource utilization.

A high level description of the execution of our testing framework is as follows:

Phase 0:

```
1. Parse XML Store consisting details of the BLOBs accessed and the machines on
   which they are located.
```

```
Phase 1:
    2. Loop over  num_trials
    3. Randomly pick a voter
            A   Look at the voter's playlist.
            B   For each BLOB in the playlist
                    i. Increment the counter of the particular BLOB.
/* At this point every voter has the information for each BLOB. */
Phase 2:
    4. Do we replicate?
            A   Check if BLOB has more than $\alpha$ fraction of the total accesses. If so,
            replicate this BLOB and continue; if not go to the next BLOB.
Phase 3:
    5. Replicate the BLOB to the machine which accesses the BLOB most.

Phase 4:
    6. Run simulator on the output generated to calculate and display new bandwidth
       access by each voter for the particular BLOB.
```

Bandwidth is calculated as (access frequency * distance).

3.4 Dynamic Access Frequency Neighborhood Replication

The Dynamic Access Frequency and Neighborhood (DAFN) replication [8] method improves over the SAF replication method by reducing the number of duplicates in the network. In the DAFN method, every node first prepares a list of eligible duplicates to the limit of available resources on that node. Next, it consults its immediate neighbors to check and see if they have chosen a candidate BLOB that is there on its list. If such a situation arises, the node with the higher access frequency replicates the BLOB, and this node picks the next candidate based on its sorted (in descending order) access frequency, and repeats the above process for this BLOB with its immediate neighbors. This process is repeated every replication interval.

The DAFN replication method is certainly a huge improvement over the SAF method, since in addition to reducing the number of duplicate replicas it also increases the probability of continued accessibility of BLOBS since neighboring nodes are less likely to go out of radio range.

The DAFN replication method, however, still does not completely utilize the knowledge of the network topology at each instant to its advantage. It is not *necessary,* for instance to have replicas created at nodes that are more than one hop away. We will show, for an example configuration, how the DAFN method would still create more replicas than were strictly necessary.

Our experiments will compare and contrast our Byzantine based replication scheme with the other access methods described above both for overall resource effectiveness, and bandwidth requirements on the network.

```
Phase 0:
    1. Parse XML Store consisting details of the BLOBs accessed and the machines on
       which they are located.

Phase 1:
    2. Loop over  num_trials
    3. Randomly pick a voter
           A      Look at the voter's playlist.
           B      For each BLOB in the playlist
Increment the counter of the particular BLOB.
/* At this point every voter has the information for each BLOB. */

Phase 2:
    4. Do we replicate?
           A      Check if BLOB that is accessed the most has more than $\alpha$ fraction of
           the total accesses. If so, replicate this BLOB and continue; if not halt.

Phase 3:
    5. Replicate top BLOB. Choose voter to replicate onto.
           A      Is the machine which has been selected adjacent to the original
           location
                 i. If Yes, then look for the next highest machine
                ii. Else, Replicate on that machine.

Phase 4:
    6. Run simulator on the output generated to calculate and display new bandwidth
       access by each voter for the particular BLOB.
```

Bandwidth is calculated as access frequency * distance.

4 Software Design

In this chapter, we examine component by component, the overall software architecture of the test bed that simulates an ad-hoc network, and implements the Byzantine replication algorithm on this test bed.

4.1 Software Architecture

We will first present the overall software architecture of our test bed, and then explain each component in detail. Our test bed is a true distributed system test bed. This means that the user can set up different physical machines that represent sub-meshes in the ad-hoc network. Each sub-mesh can have an arbitrarily large number of nodes. Utility scripts allow for automated setting up the entire test bed. We have developed the framework on Linux, though since our test bed is entirely written in C/Java, we do not anticipate portability to be an issue. We have used Sun's JVM for development since we encountered a major issue using the default RMI package that ships with Linux (gcj). Environment details for our test bench, including version numbers for all components are provided below. Figure 1 illustrates the major components in the software stack.

Component	**Version**
Operating System	Redhat Fedora Core 6
Kernel Version	2.6.18-1.2798.fc6
C-Compiler	gcc 4.1.1
Java Compiler	Eclipse Java Compiler v_677_R32x, 3.2.1 release
Database	PostgreSQL 8.1.4

Table 1: Components Information

The XML document (one per Node Entity) describes the Node Entity. Figure 2 shows an example XML structure for a node entity. As can be seen, the XML document basically configures this node with a unique id of 0, and states that this mobile node is interested in four blobs, with blob ids from 0-3. The locations of each BLOB are randomly assigned by the System Manager in Figure 1. Each of the locations essentially represents a sub mesh within the larger MANET. In our model, each physical host machine really models a mobile sub-network or *sub-mesh*. There may be multiple node entities on a machine/sub-mesh. A Node Entity represents a mobile host node. There is no distinction in interaction between two node entities on the same sub-mesh versus two node entities on different sub-meshes.
The XML parser is the component that parses the XML configuration file and shreds it into the PostgreSQL DB directly.

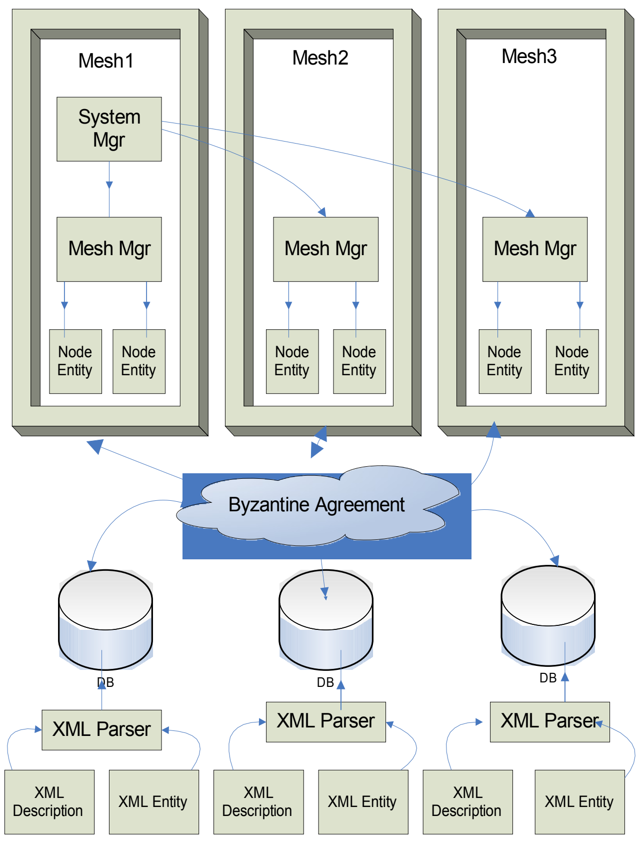


Figure 1: Structural components in the Software Stack

```
<node-details>
        <node-id>0</node-id>
        <blob>0</blob>
        <node-id>0</node-id>
        <blob>1</blob>
        <node-id>0</node-id>
        <blob>2</blob>
        <node-id>0</node-id>
        <blob>3</blob>
</node-details>
```

Figure 2: The XML description of a node Entity

i XML Processing

To allow for direct shredding of an XML document into an open source database (example. 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 XML parser consists of the following three subcomponents

a Front-end parser

The algorithm and implementation for parsing of the XML document has been borrowed from the open source C-XML parser, expat [19]. The parser provides for identification of XML elements and stubs for implementing callbacks when an XML element is encountered. In Figure 3, 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 3: XML Parser Main Loop

b 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 in Figure 4 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 4: An XML Element Representation (in an XML Tree)

In addition, for every leaf node (an address record), a context-sensitive structure of the form in Figure 5 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* node_id;
        char* blob_id;
        struct addr_rec* next;
};
```

Figure 5: Contextual representation of XML Data

The XML parse tree is finally created for in-memory tree based representation of the XML document, in addition to creating a context sensitive linked list (of address records)

c PostgreSQL extensions

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 voterID VARCHAR,
                                  OUT blobID VARCHAR)
RETURNS SETOF record
AS '/home/hlthantr/proj_one/libproj1', 'get_xml_data'
LANGUAGE C IMMUTABLE STRICT;
```

Figure 6: 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)
{
        /* Local variable declarations omitted */
        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) {
                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 7: Implementation of User Defined Function (PostgreSQL C Backend)

ii Network Model

Once the XML document that contains the blobs that this node entity is interested has been shredded into the local database on this machine, and the RMIC compiler has been used to compile all the components that export stub functions that can be invoked from other machines, the System manager component in Figure 1 is started. The System manager configures the initial blob to machine and node Entity to machine mapping. It also configures individual nodes to be normal, failed, or abnormal. Abnormal node entities change their vote in a Byzantine algorithm between rounds. An abnormal Node Entity basically reflects a node in an MANET that keeps crashing, or frequently going out of radio range with other nodes. The System Manager also interacts with the Machine manager that is responsible for the instantiation of all node entities on this physical machine. As mentioned before, a Machine in our model really represents a partial subnet consisting of nodes that are at most one hop away from each other.
The System manager then instructs each node entity to perform random access on all the BLOBS that the entity may be interested in. This information is later used in the Byzantine process to determine the BLOBS to replicate, and the machine to replicate them on. The System manager finally sets up the network model. It configures the hop distances between every node entity to allow for the use of biases in the Byzantine voting process.

Once the System manager has set up the test bed, it starts all the node entities, which basically implement the Java Timer interface, allowing them to wake up from time to time. Every time the node entity wakes up, it checks its current role. The system manager might have instructed the node entity to simply randomly access the BLOBS it is interested in, or its current state might reflect that it is required to cast its vote in a Byzantine algorithm intended to determine which blob to replicate on what machine.

iii Byzantine Replication Algorithm

Once all the nodes have been configured, they are *started* by invoking the javax.swing.Timer.start() method that these nodes extend. This makes all the nodes completely independent of the System Manager. The System Manager main thread puts itself to timed sleep, and checks for the status of the Byzantine voting process by polling each of the node entities from time to time. The nodes wake up when the timer event is fired, cast their vote for the current round, send their vote to all the other nodes, and receive votes from the other nodes. Each node then independently tabulates the votes, and changes its vote for the next round if greater than 5/8 of the nodes agree on a common machine on which to replicate this BLOB. Some of the nodes may not have any interest in this BLOB, in which case they simply convert their vote to the majority vote. If the node was configured as an abnormal node, then it changes its vote every round based on a random number modulo the number of machines in the network. Distributed consensus is achieved when 7/8 of the voters agree on the same machine for replication.
We have added *bias* to the voting process by considering past history of access by every node to the blob it is interested in. A vote by each node is scaled up by the relative access frequency to this blob by this node times the hop distance of this node to the current location of the BLOB. Clearly, accesses by nodes resident on the same machine do not carry any weight (rightly so), since they should not have any say in the replication process.

Our algorithm while very similar to the SAF [8] method, differs in the crucial aspect that in addition to considering the relative frequencies of accesses to BLOBS, we also consider the distance of each node making the access. Our intuition is to place the replica as far away from the original, provided there are enough nodes in the farther end of the network that are interested in the BLOB in question. The concept of *abnormal* nodes allows us to model nodes that frequently move in and out of radio range in the MANET. These nodes should not be very influential in the decision making process. Using the Byzantine algorithm prevents the influence that these nodes might have otherwise had, in a simple majority voting algorithm, for instance. We believe our strategy of evolving distributed consensus through a Byzantine voting process for BLOB replication to be better than the other suggested algorithms since we also take into account dynamic changes in the network (by modeling abnormal voters for instance), and giving *less* priority to such nodes for the purposes of replication.

In the next section, we examine the overall software design of the test bed. The overall software architecture of our test bed follows the best practices in Object Oriented Design.

4.2 Design of our Setup

In this section, we detail the software patterns, different classes used for our Java based framework. We will first detail the different classes used for each software component, and then look at the interrelationships between each of the components. These interrelationships are examined using the conventional UML notation in Figure 8.

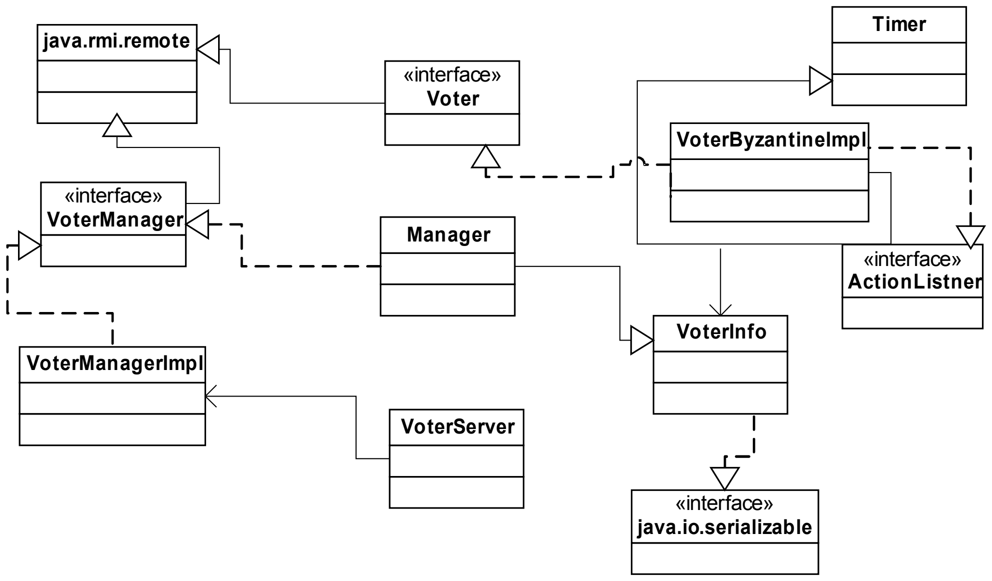


Figure 8: UML representation of the classes in the Java test bench

i System Manager

The System Manager component in our test bed framework is responsible for initial configuration of the test bed, and in moving the test bed through various phases of the network state. The System Manager component moves the testbed through the initial setup phase where Node to Machine mapping, Blob to machine mapping, configuration of individual node entities and discovery of different machines (through RMI registry services) is performed.
System Manager moves all the node entities from the first phase, where each entity is asked to randomly access its BLOB of interest, to the next phase, where the *biased* Byzantine agreement is performed on each BLOB after setting the network parameters and connectivity for the MANET.
The basic functionality for the System Manager is implemented by the Manager Class in Figure 8. It is run only on a single machine and *uses the* RMI registry to locate the per-machine Machine Manager Object. It invokes the node entity creation function on the machine manager to create each node on the machine. The System Manager *contains* the VoterInfo object that is used to communicate configuration information to each of the node entities.

ii Mesh Manager

The Mesh Manager is a daemon (continuously running process) that serves the purpose of instantiating different node entities on the machine on which it resides. It has only a single method, *createVoter* that creates the voting node entity on this host machine, and returns to the caller (the System Manager) a handle to the newly created node entity. Following the basic OO principles of keeping the definition separate from the implementation, the Machine Manager component consists of the VoterManager class (definition) and the VoterManagerImpl class (implementation). Such a design allows for the System Manager that uses the Machine Manager to be completely agnostic about the actual implementation. The *VoterServer* class is used to create a single instance of the VoterManagerImpl daemon on this machine.

iii Node Entity

The Node Entity is the principal component in the test bed framework. It is instantiated by the mesh manager, and its initial state and movement through different broad phases is controlled by the System Manager. In all other aspects, it is completely independent, and interacts with other node entities for performing the Byzantine algorithm for determining the replica location for each BLOB.
Like the Mesh Manager component, the Node Entity is also split between the *definition* and the *implementation*. The definition of the Node Entity class is contained in the interface Voter, and is captured below. Note that since some of the methods of the Node Entity could be invoked by other Node Entities through RMI, the Voter class needs to extend the *java.rmi.Remote* package, and any object invoking its methods needs to *catch* or *throw* the *RemoteException* in its accessor methods. The definition of the interface with brief explanations for each of the public methods is provided in Figure 9 below.

```
interface Voter extends java.rmi.Remote
{

// Sets the current behavior for this voter
public void setupVoter(VoterInfo vtr) throws java.rmi.RemoteException;

// Initializes the Voter for the ethernet type network model
public int initByzantine_eth(int blobID, int pos, String prev_agreement)
throws java.rmi.RemoteException;

// Method to allow other voters to determine the current IP Quad pos
public int  getPos() throws java.rmi.RemoteException;

// These functions below control the firing of the javax.swing.timer
public void start() throws java.rmi.RemoteException;
public void stop() throws java.rmi.RemoteException;

// Method for broadcasting current vote
public void receiveVote(int i, int w, Voter v)
throws java.rmi.RemoteException;

// Helper methods for remote voters to determine status of this voter
public boolean isDone()throws java.rmi.RemoteException;
public boolean isFaulty() throws java.rmi.RemoteException;
public boolean isIdle() throws java.rmi.RemoteException;
public int getRoundNumber() throws java.rmi.RemoteException;
public int  getDecision() throws java.rmi.RemoteException;
public int  getDecisionVotes() throws java.rmi.RemoteException;
// Prints the current frequency of BLOB access for this voter
public void printBlobFreq() throws java.rmi.RemoteException;

}
```

Figure 9: Interface Definition for the Node Entity (Voter class)

5 Experimental Setup

5.1 Introduction

Let us first consider a real life scenario for MANET by explaining both the network and the use-case model. One of the good examples would be that of multiple Zune devices, a Microsoft music player, forming an ad-hoc network. These devices can communicate with each other and exchange music.

There are two alternatives to obtain music:

1. They can purchase music or they could "squirt" via WiFi music from another Zune device in their ad-hoc network. However, there is the limitation when squirting music (data) in that the receiving entity can play it only a limited number of times before they would have to squirt the music again.
2. Some users may even choose to just purchase their favorite music.

The following salient issues stand out in such a use case model.

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 device, there is a monetary cost associated.
3. There may be no device with a permanent copy of the music needed within squirting distance.

The optimization opportunities in such a use case would be to create an ad-hoc network that 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. The intuition here being that the farther away the new device is from the original, the higher is the probability of continued availability of the data within the ad-hoc network.

We will consider the efficiency of our scheme for two different network models:

1. Ring Network
2. Tree Model

5.2 A Ring Topology

A ring topology is shown in Figure 10 below. A ring topology allows every node to be connected to two neighbors and act as a forwarding agent from one to the other. In our test bed, M1-M8 would really represent sub-meshes with multiple nodes that are in radio range of each other.

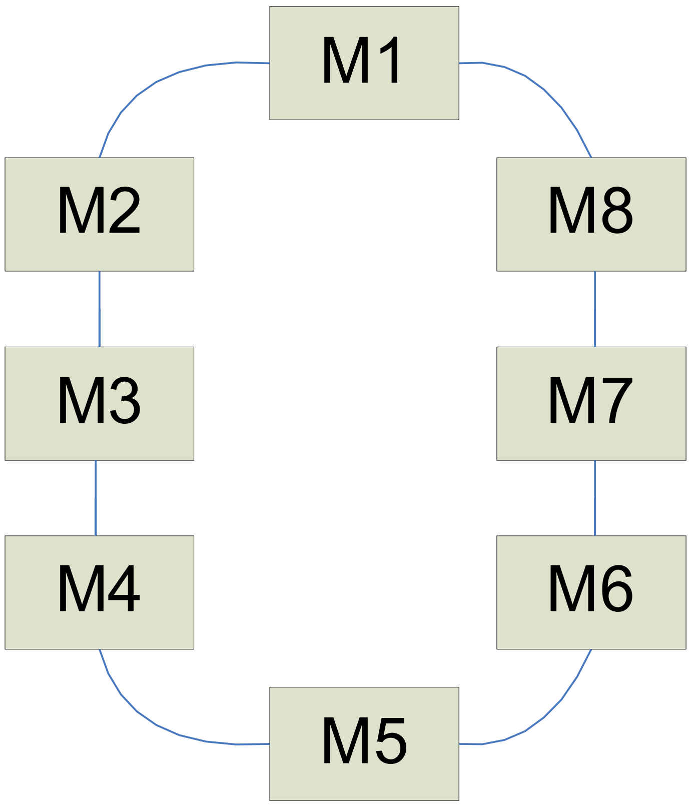


Figure 10: A sample Ring Network

A ring topology allows for requests and data to be forwarded on two independent paths on the network. For example, nodes in M1 can communicate with nodes in M3 either through M2, or through M8-M7-M6-M5-M4. Presence of redundant paths allows for such a network to be completely connected even if one of the nodes becomes disconnected, as may happen in an ad-hoc network.

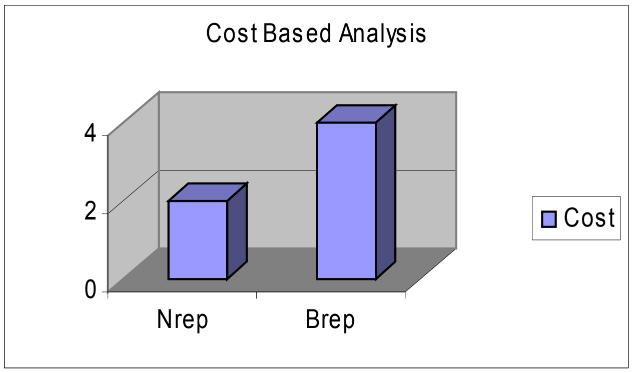
As a first step, we compare our Byzantine replication method with the basic *no-replicate* strategy. To keep the discussion simple, we consider the simple scenario of a single BLOB on the network, and there being three users, namely V2, V6 & V7 interested in accessing this BLOB, on machines M2, M6 & M7.

The Blob is on machine M2, initially.

Assuming all interested users access the blob equally, for this network configuration, the Byzantine algorithm would result in the BLOB being replicated on M6, since it is four hops away from M2, as compared to M7, that is three hops away, and the frequency of access to this BLOB by both V6 and V7 are the same.

The Replication performed results in a copy of the BLOB on machine M6 and leads to a reduction in bandwidth for the overall network from this BLOB traffic. V6 has the BLOB data replicated on its machine and V7 has to travel one hop distance to obtain the BLOB. V2 can continue to access the BLOB on M2 and is not impacted by the replication.

Figure 11 compares the Byzantine replicate strategy with the no replica strategy. For cost based analysis we assume 1 unit cost per BLOB per User, while for Bandwidth we assume 2 units per BLOB per machine as the cost to download the song from the central server. We see a fairly significant cost saving and a reduction in bandwidth requirements based on our replication strategy.



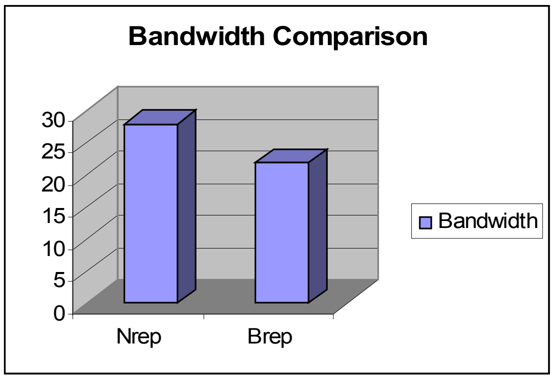


Figure 11: Comparing Byzantine replication with No-replica strategy

6 Results

In this chapter, we present results of running our test.

6.1 XML Parser

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

```
$ psql testdb

testdb=# \i proj6.sql
            DROP FUNCTION
            CREATE FUNCTION
            DROP TABLE
            CREATE TABLE
            INSERT 0 4
            INSERT 0 2
            INSERT 0 4
            INSERT 0 2
            INSERT 0 4
            INSERT 0 3
            INSERT 0 2
            INSERT 0 4
             voter_id | blob_id
            ----------+---------
             0       | 0
             0       | 2
             0       | 1
             0       | 3
             1       | 0
             1       | 2
             2       | 0
             2       | 2
             2       | 1
             2       | 3
             3       | 1
             3       | 2
             4       | 0
             4       | 3
             4       | 2
             4       | 1
             5       | 1
             5       | 3
             5       | 0
             6       | 0
             6       | 3
             7       | 1
             7       | 3
             7       | 2
             7       | 0
            (25 rows)
```

Figure 12: Output from SQL Script (Truncated)

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

6.2 Ring Model

Figure 13 shows the output which was observed when Byzantine Agreement is performed for one of the BLOBs B1. Since this is a ring network, all the users, exchange information of their frequency. Depending on the machine which has maximum frequency for a particular BLOB, agreement would be reached by all the users to create a replica of the BLOB on that machine.

```
VTR[0]:VOTE:[0]:RND[1]
VTR[5]:VOTE:[0]:RND[1]
ROUND: 1 VOTER: 5 DECISION: 0
ROUND: 1 VOTER: 6 DECISION: 0
ROUND: 1 VOTER: 1 DECISION: 0
ROUND: 1 VOTER: 7 DECISION: 0
ROUND: 1 VOTER: 2 DECISION: 0
ROUND: 2 VOTER: 3 DECISION: 0
ROUND: 3 VOTER: 0 DECISION: 0
Voter 5  Agreement Reached!! Agreement is 0
VTR[6]:VOTE:[0]:RND[3]
Voter 6  Agreement Reached!! Agreement is 0
VTR[7]:VOTE:[0]:RND[3]
Voter 7  Agreement Reached!! Agreement is 0
VTR[3]:VOTE:[0]:RND[3]
Voter 3  Agreement Reached!! Agreement is 0
VTR[4]:VOTE:[0]:RND[3]
ROUND: 3 VOTER: 4 DECISION: 0
Voter 4  Agreement Reached!! Agreement is 0
VTR[1]:VOTE:[0]:RND[4]
Voter 1  Agreement Reached!! Agreement is 0
VTR[0]:VOTE:[0]:RND[4]
Voter 0  Agreement Reached!! Agreement is 0
VTR[2]:VOTE:[0]:RND[4]
Voter 2  Agreement Reached!! Agreement is 0
!!!BYZANTINE REACHED FOR BLOB #0!!!
```

Figure 13: Byzantine Agreement on a Ring Network

6.3 Tree Model

Tree Model deals with internet ids as the addresses. An Internet address typically consists of four quads; for example quad1.quad2.quad3.quad4.

Byzantine Agreement is performed on each quad. Agreement reached for a particular quad serves as the basis of selection for the next quad, i.e.; for a quad id to participate in the voting process, its previous quad should have the same id on which consensus was reached by all the users.

This logic can be understood better by referring the output observed in case of Internet Protocol , which is observed in the below Figure 14.

```
STARTING BYZANTINE FOR BLOB #0

VOTER: 4 :BLOB: 0POSITION: 0 :AGREEMENT IS: 192.
VOTER: 7 :BLOB: 0POSITION: 0 :AGREEMENT IS: 192.
VOTER: 2 :BLOB: 0POSITION: 0 :AGREEMENT IS: 192.
VOTER: 3 :BLOB: 0POSITION: 0 :AGREEMENT IS: 192.
VOTER: 0 :BLOB: 0POSITION: 0 :AGREEMENT IS: 192.
VOTER: 6 :BLOB: 0POSITION: 0 :AGREEMENT IS: 192.
VOTER: 1 :BLOB: 0POSITION: 0 :AGREEMENT IS: 192.

VOTER: 4 :BLOB: 0POSITION: 1 :AGREEMENT IS: 192.168.
VOTER: 5 :BLOB: 0POSITION: 1 :AGREEMENT IS: 192.168.
VOTER: 2 :BLOB: 0POSITION: 1 :AGREEMENT IS: 192.168.
VOTER: 7 :BLOB: 0POSITION: 1 :AGREEMENT IS: 192.168.
VOTER: 3 :BLOB: 0POSITION: 1 :AGREEMENT IS: 192.168.
VOTER: 0 :BLOB: 0POSITION: 1 :AGREEMENT IS: 192.168.
VOTER: 6 :BLOB: 0POSITION: 1 :AGREEMENT IS: 192.168.
VOTER: 1 :BLOB: 0POSITION: 1 :AGREEMENT IS: 192.168.

VOTER: 4 :BLOB: 0POSITION: 2 :AGREEMENT IS: 192.168.0.
VOTER: 5 :BLOB: 0POSITION: 2 :AGREEMENT IS: 192.168.0.
VOTER: 7 :BLOB: 0POSITION: 2 :AGREEMENT IS: 192.168.0.
VOTER: 2 :BLOB: 0POSITION: 2 :AGREEMENT IS: 192.168.0.
VOTER: 3 :BLOB: 0POSITION: 2 :AGREEMENT IS: 192.168.0.
VOTER: 0 :BLOB: 0POSITION: 2 :AGREEMENT IS: 192.168.0.
VOTER: 6 :BLOB: 0POSITION: 2 :AGREEMENT IS: 192.168.0.
VOTER: 1 :BLOB: 0POSITION: 2 :AGREEMENT IS: 192.168.0.

VOTER: 4 :BLOB: 0POSITION: 3 :AGREEMENT IS: 192.168.0.0.
VOTER: 5 :BLOB: 0POSITION: 3 :AGREEMENT IS: 192.168.0.0.
VOTER: 7 :BLOB: 0POSITION: 3 :AGREEMENT IS: 192.168.0.0.
VOTER: 2 :BLOB: 0POSITION: 3 :AGREEMENT IS: 192.168.0.0.
VOTER: 3 :BLOB: 0POSITION: 3 :AGREEMENT IS: 192.168.0.0.
VOTER: 0 :BLOB: 0POSITION: 3 :AGREEMENT IS: 192.168.0.0.
VOTER: 6 :BLOB: 0POSITION: 3 :AGREEMENT IS: 192.168.0.0.
VTR[1]:VOTE:[1]:RND[2]Voter 1  Agreement Reached!! Agreement is 0
VOTER: 1 :BLOB: 0POSITION: 3 :AGREEMENT IS: 192.168.0.0.
STOPPING VOTER #1
*****BYZANTINE AGREEMENT REACHED FOR BLOB*****0VOTER ID 0AGREEMENT = 192.168.0.0.
```

Figure 14: Byzantine Agreement for Internet Protocol Model

One of the limitations of a ring network is that there is a probability of arriving at a conclusion that no replication is required, even when there is large number of accesses from a distance.

Let us consider a small example to explain our concept further

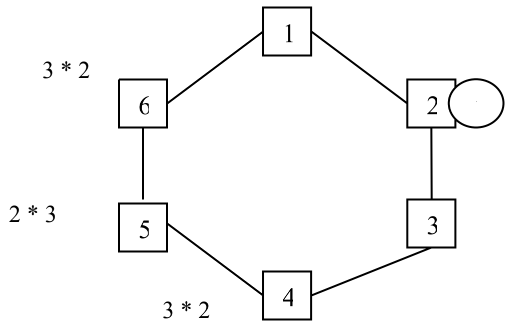


Figure 15: Limitation of Ring Network

As observed in the above figure there are 3 users with the same weight 6 units. Thus this might end up in no replication taking place.
In order to overcome this problem, we introduce our idea to a Tree Model Structure. In case of a tree model structure a Byzantine Agreement is performed at each depth i.e. if the IP address is of the form quad1.quad2.quad3.quad4, then Byzantine Agreement is performed for each quad. From the above figure it is apparent that the three nodes are located very close to each other; i.e. along the same depth.

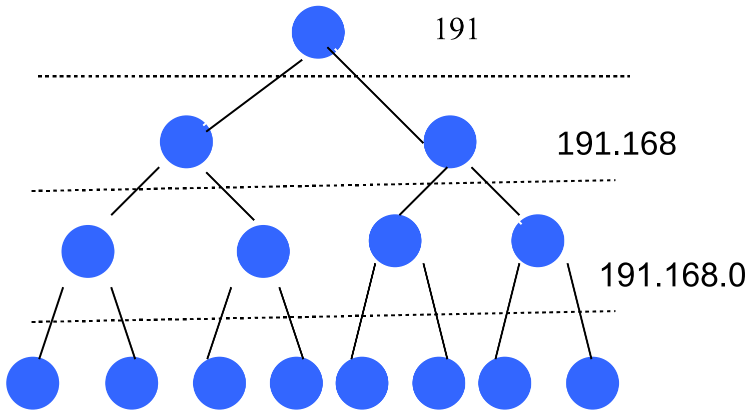


Figure 16: Depth Analysis – Tree Model.

Let us assume that the master copy of the BLOB is present at 190.168.0.000.
Users 6,5 and 4 are located at a distance and on machines

191.168.0.100,
191.168.0.101,
191.168.0.102

By performing Byzantine Agreement on each quad we actually manage to move our copy closer to the desired location even if not on the exact machine. i.e.; we manage to move it from 190.168.0.000 to 191.168.0 which would be much closer than the original location.

6.4 Analysis - Ring Model

Our simulation for the ring model consists of eleven users distributed across eight machines. Since Song 1 located on machine 2, with a frequency of fifty one is the song with highly frequency, we use song 1 for our future analysis.

Song 1 is located on machine 2 and is being accessed most frequently by user 7 who is located on machine 4, thus having to travel a distance of four for each access, resulting in a frequency of 32 for the 8 times he tried to squirt the music. The same behavior is also observed for user 11 located on machine 3 and user 3 located on machine5.

Byzantine Agreement is performed and all the users come on a consensus to purchase one more copy of song 0 and place it on machine 4.

Table 2 and Figure 1 provides us a brief statistics on the improvements observed from our approach in comparison to a Non Replication based approach.

Users	Location	# of Accesses	Bandwidth Non - Replicated Scheme	Bandwidth Replicated Scheme
1	M1	2	2	2
2	M3	2	2	2
3	M5	4	12	4
4	M3	2	2	2
5	M1	2	2	2
6	M5	5	15	5
7	M4	8	32	0
8	M2	0	0	0
9	M6	3	12	6
10	M8	3	6	6
11	M6	4	8	4

Table 2: Frequency analysis for Ring Network

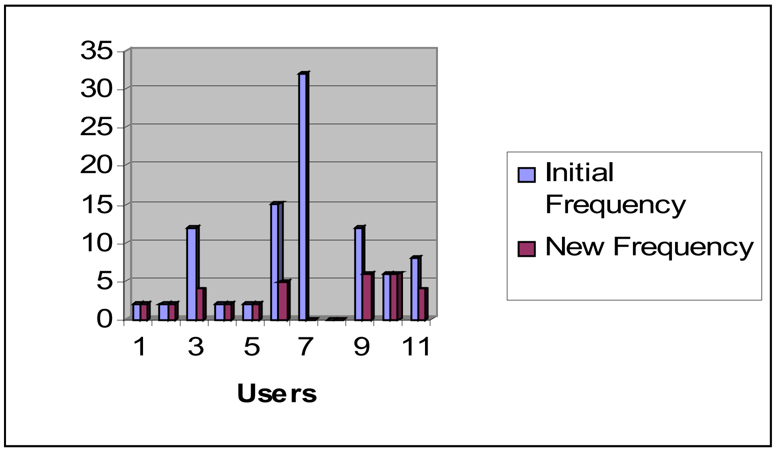


Figure 17: Frequency ratio

As observed in the above graph, there has been a tremendous improvement in frequency ratio with respect to Users 7, 6 and 3.

6.5 Analysis - Tree Model

Let us consider an Internal Protocol Model which consists of machines distributed across a network.

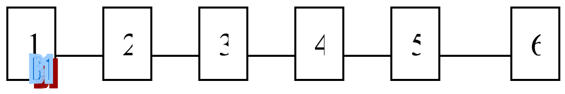


Figure 18: Machines across a Tree Model

As in the ring model, our simulation here consists of 4 songs, 11 users distributed across an Ethernet Model consisting of 8 machines. Song 1 located on machine 1 is the most popular song and would be our point of interest through out this section.
Users 2 and 7 with their frequency (distance * number of times) of 36 and 28 and located on machines 7 and 8 are users with highest frequency squirting song1. On running a simulator for this logic on our implementation, it was observed that all the users arrived at a conclusion that it was advisable for them to purchase song1 and place it on machine 7.

Table 3 below points out the difference between the initial logic and the advantage obtained with our implementation, was observed for Music 0.

Users	Location	# of Accesses	Bandwidth Non - Replicated Scheme	Bandwidth Replicated Scheme
1	M3	1	2	2
2	M7	6	36	0
3	M2	1	1	1
4	M4	0	NA	NA
5	M6	5	10	2
6	M1	3	0	0
7	M8	4	28	4
8	M1	4	0	0
9	M3	1	2	2
10	M4	2	6	6
11	M6	2	10	2

Table 3: Frequency analysis for Tree Model.

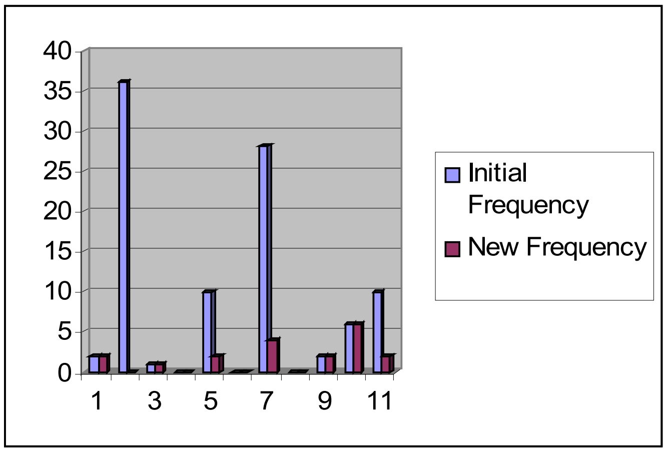


Figure 19: Frequency ratio for Existing system versus changed system.

As observed in the above chart, the bandwidth covered by user2 has reduced from 37 to 0. Such type of improvements can also be observed for users 4 and users 6. Most importantly we observe that all of the users either show an improvement in their bandwidth or have the same bandwidth as before. None of the users need to cover more bandwidth than before.

7 Comparison

In this chapter, we compare our Byzantine based replication strategy with three other replication strategies described previously, the Continuous Broadcast, Static Access Frequency and the Dynamic Access Frequency Neighborhood for cost and bandwidth savings.

In all these, we have used real access frequency numbers derived from our model, and backfilled those into the other models.

7.1 Continuous Broadcast (CB) Replication

The Continuous Broadcast Replication strategy has been previously described in the Chapter 3. Here we look at how the CB replication strategy is applied to a ring configuration.

i Setup

Our Setup for comparison of the CB Model with our approach consists of eight machines distributed across a ring network. Blob B1 (located on machine M2) and accessed by Users V2 V3 V6 and V7 is the most commonly accessed BLOBs and hence serves as the point for comparison for our analysis.

ii Analysis – Results

In case of CB Replication, after a fixed duration copies of BLOB commonly accessed are provided to all users, irrespective of whether they access the same. The choice of replication is left to the user.

Table 4 provides us with differentiation between the Continuous Broadcast approach, our approach and the traditional approach (i.e.; an approach wherein no replication is performed). The term NA (Not Applicable) has been used for users who do not access BLOB B1.

User	Mesh Id	Continuous Broadcast	Traditional Approach (no replication)	New Approach (replication performed)
1	M1	1	NA	NA
2	M2	0	0	0
3	M3	1	1	1
4	M4	2	NA	NA

5	M5	3	NA	NA
6	M6	4	4	1
7	M7	3	5	0
8	M8	2	NA	NA

Table 4: Bandwidth Comparison between models

below, provides a summation of the bandwidths for each approach and attempts to compare them.

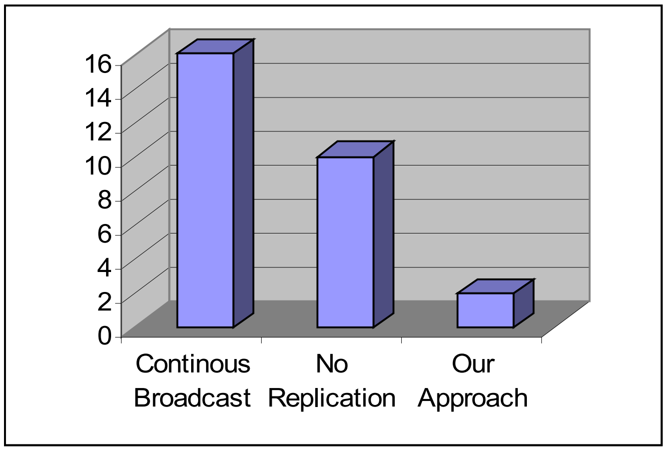


Figure 20: Bandwidth comparison

Cost-based comparison for all the 3 models i.e.; Continuous Broadcast, No Replication, Our Approach has been provided in Figure 19 below. This analysis has been performed for only one BLOB B1. In case of Continuous Broadcast a copy of the BLOB B1 would be broadcasted to all the machines. Thus assuming a cost of 1 unit/copy, we observe that the cost associated with Continuous Broadcast is around 8. Since only copy of the BLOB is present in case of Traditional Approach (No replication), the cost is around 1 unit.

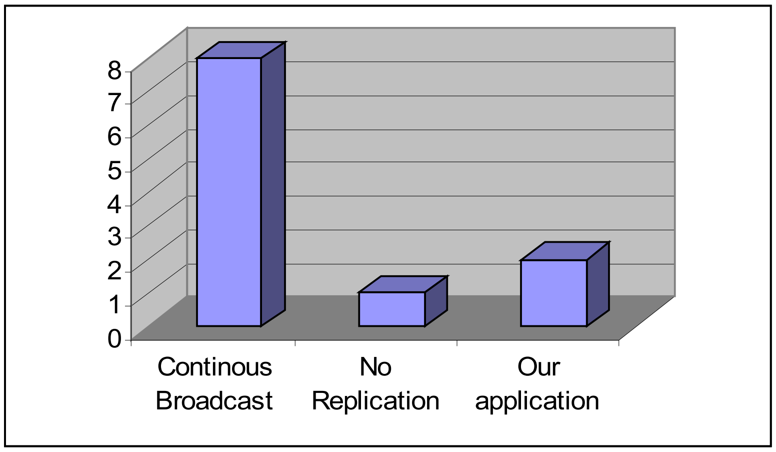


Figure 21: Cost-based comparison

7.2 Static Access Frequency (SAF) Model

We next compare our model against the SAF model described in Static Access Frequency in Chapter 3.

i Setup

For the purpose of comparison, we will consider a single BLOB (let's say B1). B1 is located on machine M1 and is accessed by approximately 6 users distributed across a ring network.

Lets say the frequency of access for BLOB BLOB1 for the 6 users is as specified in Table 5 below.

User	Mesh	Access
U1	M1	10
U2	M2	25
U3	M3	18
U4	M4	15
U5	M5	18
U6	M6	20
U7	M7	22
U8	M8	20
U9	M9	18
U10	M10	15

Table 5: Frequency of access for BLOB B1.

ii Analysis on the SAF Model

The logic behind the SAF model can be summarized as, "a copy of the BLOB should be replicated to the machine from which access is made maximum number of times". Following this logic on our above setup, we observe that BLOB B1 would be replicated to machine M2, since user U2, who has the highest access, is located on the same.

iii Analysis of our Voting Algorithm

The main difference between our approach and SAF model lies in the fact that we take the distance factor into account as well. In short, our approach states that an efficient replication would be achieved if it is possible to provide a replica on an object which not only accesses the BLOB frequently but is also located at a distance from where the BLOB originally resides.

As mentioned in our setup, we assume that the six machines are located at a distance of 1 unit from its adjacent machine. Table 6 below is derived using these priniciples.

User	(Access * distance)	New Frequency
U1	10 * 0	0
U2	25 * 1	25
U3	18 * 2	36
U4	15 * 3	45
U5	18 * 4	72
U6	20 * 5	100
U7	22 * 6	132
U8	20 * 5	100
U9	18 * 4	72
U10	15 * 3	45

Table 6: Frequency access /User for our implementation

From Table 6 we observe that User U7 located on machine M7 has very high accesses frequency and is also located at a higher distance (six) from M1; the original location of BLOB B1.

iv Comparison Results

USER	***Frequenc y SAF***	***Frequency Our approach***
U1	0	0
U2	0	25
U3	18	36
U4	30	45
U5	54	36
U6	80	20
U7	110	0
U8	100	20
U9	72	36
U10	45	45

Table 7: Bandwidth based Comparison

On the basis of the the data observed in the above table, we mparison of the bandwidth that nee methods; SAF and Our Algorithm.
Bandwidth is calcu

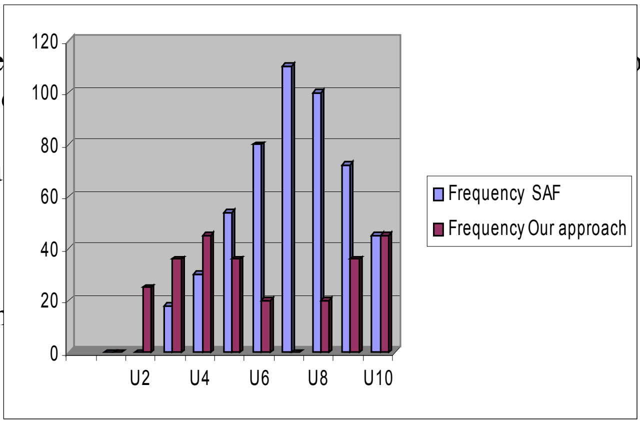


Figure 22: bandwidth comparison/user

The above graph shows that there are certain regions where considerable amount of benefit has been obtained by using our approach but there are some other users for whom using the SAF method appears to be more economical.

In order to get a better understanding of the numbers observed in case of the two methods, we perform an analysis of the total bandwidth that is required in both the approaches.

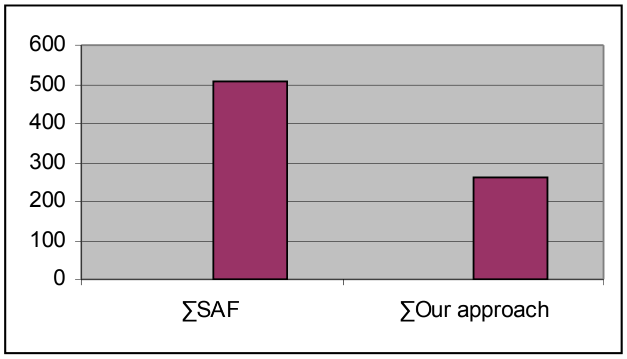


Figure 23: Comparison of Total Bandwidth

As observed in the above graph, for our setup, we observe improvement over SAF Model.

7.3 Dynamic Access Frequency and Neighborhood (DAFN) Replication

i Setup

For ease in understanding let us consider a single BLOB (let's say B1). B1 is located on machine M1 and is accessed by approximately 6 users distributed across a ring network.

Lets say the frequency of access for BLOB BLOB1 for the 6 users is as specified in Table 8 below.

User	Mesh	Access
U1	M1	10
U2	M2	30
U3	M3	28
U4	M4	15
U5	M5	18
U6	M6	20
U7	M7	25
U8	M8	20
U9	M9	18
U10	M10	15

Table 8: BLOB Frequency / User for DAFN Model

The highest frequency of access is at Machine M2. However, this machine M2 is adjacent to machine M1 which already consists of a copy of this replication. DAFN model states that if two adjacent machines consist of copies then the replica should not be created on the adjacent.
Thus, we check the machine which has next highest access. In our case, a copy of replica is created by the DAFN model on machine M3, the next highest machine which does not have a copy of B1 on its adjacent nodes (machines M2 and M4).

Our approach states that an efficient replication would be achieved if it is possible to provide a replica on an object which not only accesses the BLOB frequently but is also located at a distance from where the BLOB originally resides.

As mentioned in our setup, we assume that the six machines are located at a distance of 1 unit from its adjacent machine. Table 9 below is constructed using these principles.

User	(Access * distance)	New Frequency
U1	10 * 0	0
U2	30 * 1	30
U3	28 * 2	56
U4	15 * 3	45
U5	18 * 4	72
U6	20 * 5	100
U7	25 * 6	150

U8	20 * 5	100
U9	18 * 4	72
U10	15 * 3	45

Table 9: Frequency access /User for our implementation

On the basis of the above table, we observe that User U7 located on machine M7 has very high accesses frequency and is also located at a distance from M1 ; the original location of BLOB B1.

ii Comparison Results

USER	DAFN	Our approach
U1	0	0
U2	30	30
U3	0	56
U4	15	45
U5	36	36
U6	60	20
U7	100	0
U8	100	20
U9	72	36
U10	45	45

Table 10: Comparison between DAFN and our approach

On the basis of the analysis performed in the above two sections and the data in Table 10, we plot the below graph which provides us with a comparison of the bandwidth that needs to be accessed / user for each of the above two methods: SAF and Our Algorithm. Bandwidth is calculated

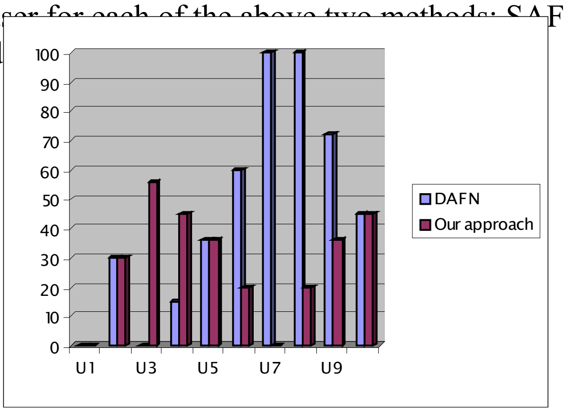


Figure 24: Bandwidth Comparison

In order to get a better understanding of the numbers observed in case of the two methods, we perform an analysis of the total bandwidth that is required in both the approaches.

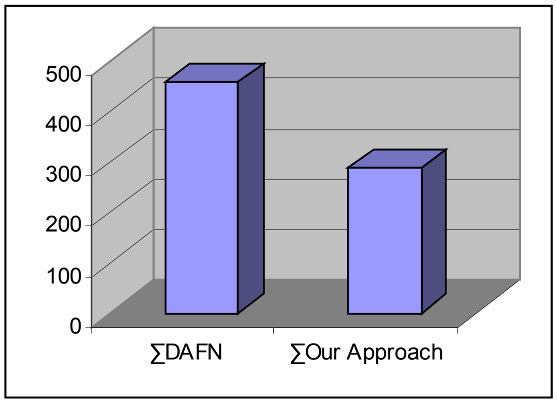


Figure 25: Total Bandwidth Comparison

The above graph shows that there are certain regions where considerable amount of benefit has been obtained by using our approach but there are some other users for whom using the SAF method appears to be more economical.

As observed in the above graph, for our setup, we observe improvement over DAFN Model.

8 Conclusion

We have built an easily extensible; Java based distributed systems test bed for doing studies in BLOB replication. We have used our framework to propose a new distributed consensus algorithm for BLOB replication in a mobile ad-hoc network. We have compared and contrasted our approach with current research in this area and have shown our approach to perform better for certain commonly occurring topologies in both bandwidth and cost saving compared to other mechanisms.

However, we have not considered *replica update* issues in our current framework. We believe that in an MANET environment targeted towards music & video sharing, updates are highly infrequent and rare.

We believe our work can be extended to include other network topologies besides the ring and the Tree Model.

References

[1] Zune, from Wikipedia the free encyclopedia. http://en.wikipedia.org/wiki/Zune

[2] Karumanchi, G., Muralidharan, S., Prakash R., "Information Dissemination in Partitionable Mobile Ad Hoc Networks", Proceedings of 18th IEEE Symposium on Reliable Distributed Systems, Lausanne, Switzerland, 1999

[3] D.J. Baker, J. Wieselthier and A. Ephremides, "A distributed algorithm for scheduling the activation of links in a self-organizing, mobile, radio network", Proceedings of IEEE ICC'82, 1982.

[4] A. McDonald and T.Znati, "A Mobility based Framework for Adaptive Clustering in Wireless Ad Hoc Networks", IEEE Journal on Selected Areas in Communications, vol. 17, no. 8, pp. 1466-1486, August 1999

[5] S. Jiang, D. He, and J. Rao, "A Prediction-based Link Availability estimation for Mobile Ad Hoc Networks," in Proceedings of IEEE Infocom, Anchorage, Alaska, April 2001

[6] W. Su, S. J. Lee, and M. Gerla, "Mobility prediction and Routing in Ad Hoc Wireless Networks", International Journal of Network Management, 2000

[7] T Hara, N Murakami, S. Nishio, "Replica Allocation for Correlated Data Items in Ad Hoc Sensor Networks", SIGMOD Record, Vol 33, No. 1, March 2004

[8] T Hara, "Effective Replica Allocation in Ad Hoc Networks for improving Data Accessibility", Proceedings of IEEE Infocom 2001, pp 1568-1576.

[9] Introduction to Algorithms, 2nd edition. Cormen, Leiserson, Rivest, and Stein. McGraw Hill. 2001

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

[11] 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

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

[13] 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

[14] 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.

[15] 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.

[16] 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.

[17] K. Rothermel, C. Becker, J. Hahner, "Consistent Update Diffusion in Mobile Ad Hoc Networks", TR-2002-04, University of Stuttgart, IPVS, Germany.

[18] Fundamentals of Database Systems, Fourth Edition, R. Elmasri, S. Navathe, 2003

[19] "Using Expat", http://www.xml.com/pub/a/1999/09/expat/index.html

[20] Java RMI White paper
http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp