Proceedings of the IASTEDInternational Conference
Parallel and Distributed Computing Systems
November 3-6, 1999,MIT, Boston,USA

An Automatically Reconfigurable Distributed Data Storage System for High Data Availability

Gwang S. Jung

Department of Mathemathics and Computer Science

Lehman College, The City University of New York
Bronx, NY 10468 gjung@alpha.lehman.cuny.edu

Qutaibah M. Malluhi and **Farida Chowdary**

Department of Computer Science

Jackson State University, Jackson, MS 39217
qmalluhi@homs.jsums.edu fachowdh@ccaix.jsums.edu

Abstract

In a largely distributed data storage system, high data availability becomes a major issue of concern. Storage server failure increases with the number of servers, or data objects get inadvertently deleted or corrupted. Maintaining highly available data storage service in a distributed environment is an acute problem. We propose a novel scheme by which highly available and dependable data service is achieved. The proposed scheme stripes a file into data blocks and distributes them over distributed storage servers. To retrieve the original file, the data blocks are downloaded from the storage servers in parallel and then merged. The scheme employs a reconfigurable architecture consisting of a cluster of distributed data servers. The reconfiguration of the storage is performed in order to recover missing data blocks by a coding method and to relocate these onto operational backup servers for continuous data availability.

keywords: Distributed Systems, Availability, Coding, Reliability, Storage Systems.

1. Introduction

The number of computer systems and users connected to the Internet has been growing rapidly. In a very large distributed computing environment such as the Internet, the likelihood of server failure increases with the number of servers. Failure may occur due to software and hardware malfunctioning, excessive server load, network congestion, and/or natural disasters. Such failures may lead to data unavailability and therefore less dependable service to the user.

In this paper, we develop a scheme for enabling high data availability and dependable service in a distributed environment. The proposed scheme slices a file into data blocks and distributes them over storage servers. To retrieve the original file, the data blocks are downloaded from the storage servers in parallel and then merged. Data blocks may however become unavailable due to the server being down or the blocks being deleted or corrupted in some way. In the event of a problematic file, the reconfiguration scheme rebuilds the file by recovering the missing or corrupt blocks and redistributes them over the operational servers. A problematic file is defined here as the one where some blocks are not available for correct download. The reconfigurability ensures the availability of the data blocks before a request is made to download them. The user can rely on dependable service as he has access to the data in a timely fashion. The restoration of the missing data blocks is based on a coding scheme also presented in this paper.

Current distributed file storage systems, such as the Coda, the Andrew File System, and the Echo File system, store data objects across multiple storage servers. The Coda file system, which inherits largely from the Andrew File System, was developed to focus on the availability issue. It does not reconfigure the system to provide the data availability. Instead it keeps read-only replicas of files at remote sites in case of a server failure and disconnected operation. The Echo file system has various ways of detecting faults, such as server failure, automatically and can report these by a daemon process that sends messages to people responsible for dealing with faults. However, reconfiguration is done manually. In general, availability of data is provided by keeping the file at a primary site for download, and its replicas at other sites in case of primary site failure. The Echo system relies heavily on redundant copies of everything in case of failure, including servers and entire data objects [1][3]. The secondary site monitors the primary site availability and vice versa. None of these systems uses an autonomous tool that monitors overall data availability and reconfigures the storage system automatically.

Our proposed scheme recovers only the part of a data object that is problematic, and redistributes it onto an operational server. Redundant parity blocks are added to the original data blocks. The parity blocks are used for recovering original data blocks if a portion of the data blocks is unavailable. The data recovery is automatically performed by a decoding procedure based on the redundant parity blocks. Our reconfiguration scheme is done on a periodic and automatic basis without user knowledge or manual intervention.

In this paper, we describe a prototype, named ***RDSS*** (***R***econfigurable ***D***istributed Data ***S***torage ***S***ystem), developed based on the proposed scheme. Files in the RDSS are striped into data blocks and distributed onto a chosen set of functional data servers for storage. RDSS automatically checks the server status and reconfigures the data servers if necessary.

In section 2, we describe the architecture of the RDSS. The coding technique, which underlies the reconfiguration scheme for rebuilding the missing data blocks, will be discussed in section 3. In section 4, the reconfiguration scheme of the RDSS is explained in some detail. Section 5 will present the conclusion.

2. The Architecture of the Reconfigurable Distributed Data Storage System

The Reconfigurable Distributed Data Storage System (RDSS) provides an efficient, reliable, and highly available data storage and retrieval environment. In the RDSS environment, files are striped into data blocks and then distributed over data servers for storage. A user retrieves a file by using multithreaded communication links to the data servers. The servers then transmit the data blocks in parallel to the user. The user collects and merges these blocks into the original file. If necessary, parity blocks are retrieved and decoded to recover original unavailable data blocks. High data rates are achieved through utilizing the cumulative bandwidth of multiple network paths. The bandwidth of the user's client can therefore be fully utilized. The reconfigurability ensures the availability of the data blocks before a request is made to download them. Scalability can be achieved by simply adding more servers to increase the level of parallelism in data delivery and the degree of data availability and reliability.

A schematic view of the RDSS is shown in Figure 1. The four major components of the RDSS are: data/ parity block distributor, data/parity block server, autofixer, and the user client.

(i) Distributor: The main functions of this component are to slice a file object into a set of equal-sized data blocks and to upload them to distributed data/parity block servers. The *married* block concept is used to group *m* data blocks, out of *p striped* data blocks of the original file, together with *k* parity blocks for fault recovery, where $k \le m$ (refer to the Figure 3). An encoding algorithm is used to encode *m* data blocks into *k* parity blocks such that *k* out of *m* data blocks missing can be tolerated [6]. These parity blocks are uploaded to the parity servers. After the blocks (data and parity blocks) have been successfully uploaded, the information about distributed data blocks, such as logical block identification and its physical address (e.g. the URL of a block if the data servers are Web servers), is created by the Distributor. The information is kept in a *metafile* and uploaded to a server called *master* server (one of the data block servers). The metafile can be replicated over several servers for higher availability.

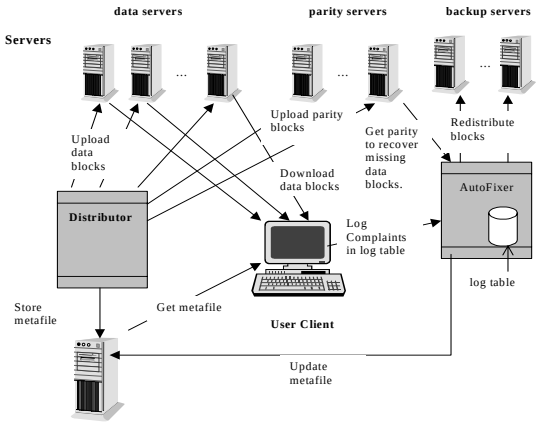

Figure 1 The Architecture of the RDSS

(ii) Server: There are three types of servers: data servers, parity servers, and backup servers. A data server stores data blocks in its file system. After a data server stores a data block, it sends an acknowledgment back to the Distributor containing the physical address of the data block. The parity server stores the parity blocks to tolerate missing data blocks. Backup servers are used to store newly recovered data blocks, initially stored in problematic data servers.

(iii) AutoFixer: The AutoFixer is used for maintaining high data availability of the system through reconfiguration. It monitors the availability of files and the status of the data servers periodically. It is responsible for making sure that data blocks are

retrievable from a data server. The AutoFixer is able to automatically rebuild missing or corrupt data blocks by a decoding algorithm based on the corresponding parity blocks, and redistributes them to ensure the data availability. The Figure 2. shows the process of the storage reconfiguration performed by the AutoFixer. In Section 3 the reconfiguration scheme of the AutoFixer will be described in detail.

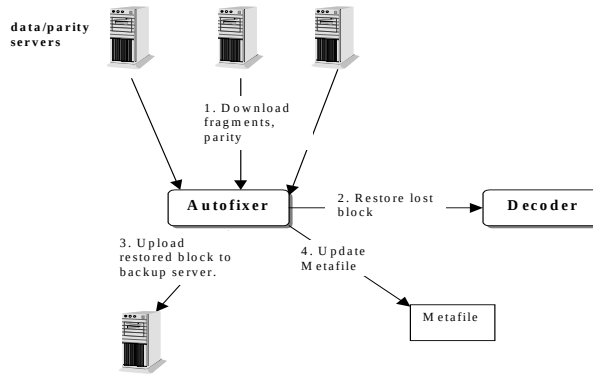

Figure 2 The Process of Storage Reconfiguration by the AutoFixer

(iv) User Client: The user client first downloads the metafile from a master server and creates multithreaded links to data block servers. Data blocks are then downloaded from the data servers to the client from the data block servers in parallel. If any data blocks are found to be missing, the client downloads the associated parity blocks, and then automatically rebuilds the lost data blocks using the decoding algorithm. The data blocks are finally merged into the original file. The client also has an active part to play in the reconfiguration process by communicating pertinent information to the AutoFixer.

3. Coding for Recovering Missing Blocks

In this section, the coding algorithm is described in some detail. This algorithm is the basis for the recovery of lost blocks in our reconfiguration scheme. In this method, we encode m symbols of the original data object into $n = m + k$ symbols. The m original symbols are called an *information word* and the coded n symbols are called a *code word*. If the information word appears in the code word, the code is called to be *systematic*. In the absence of faults, the user client needs to be able to get the information word stored on the various data servers with no decoding. Therefore, we need to use systematic codes for our environment.

All of the arithmetic operations used in this section are operations in a *Galois field*. Throughout this section, we use capital letters to denote vectors or matrices. A superscript is used to indicate the matrix dimensions. For example, $P^{m\times k}$ indicates that the matrix P has m rows and k columns.

3.1. Encoding Process

Let $U = (u_0, u_1, \ldots, u_{m-1})$ be an information word. A code word $V = (v_0, v_1, \ldots, v_{n-1})$ will be produced by multiplying U by an $m \times n$ matrix G. The rows of G will act as a basis for a vector space containing all the code words. Therefore, $V = U \times G$. We require that the rows of G be linearly independent. Otherwise, there will be more than two information words that will be mapped onto the same code word. Moreover, we want the code to be systematic. Therefore, G should be of the form,

$$G = \left[I^{m\times m} \middle| P^{m\times k} \right] = \begin{bmatrix} 1\,0\,0\,\ldots\,0 & g_{0m} & \cdots & g_{0,n-1} \\ 0\,1\,0\,\ldots\,0 & g_{1m} & \cdots & g_{1,n-1} \\ 0\,0\,1\,\ldots\,0 & g_{2m} & \cdots & g_{2,n-1} \\ \vdots\,\vdots\,\vdots\,\vdots\,\vdots & \vdots & \vdots & \vdots \\ 0\,0\,0\,\ldots\,1 & g_{m-1,m} & \cdots & g_{m-1,n-1} \end{bmatrix}$$

Therefore,

$$V = U \times G = U \times \left[I^{m\times m} \middle| P^{m\times k} \right]$$

$$= [U \times I | U \times P] = (u_0, u_1, \ldots, u_{m-1}, v_m, v_{m+1}, \ldots, v_{n-1})$$

Let's assume that $k = n - m$ servers are faulty. This means that k elements of the V are missing. Our goal is to find the way for the AutoFixer to reconstruct the original data (i.e., the vector U). Let V' be the set of m married blocks which the client could download, i.e., V' is V without the k missing elements. Let G' be the $m \times m$ matrix generated by ignoring the k columns of G corresponding to the missing blocks. Clearly, $V' = UG'$. Suppose that G' is invertible, we can obtain U by the equation $V'G'^{-1} = UG'G'^{-1} = U$. Since U can be reconstructed only if G' is invertible, obtaining invertible G' matrix is the key to the decoding process.

3.2. Generating the G matrix

Therefore, the matrix G must satisfy the following three conditions:

(1) $G = [I|P]$.

(2) The rows of the G matrix must be linearly independent.

(3) Every m columns of G have to be linearly independent.

Condition (1) ensures that G produces a systematic code, condition (2) ensures that no two information words are mapped into the same code word, and condition (3) ensures that G', which is a column subset of G, is always invertible. The following two steps describe an algorithm for generating the desired G matrix. This algorithm assumes that we are using $GF(q)$ to develop our code and that $n \le q-2$.

step1: Let $y_1, y_2, y_3, \ldots, y_n$ be any n distinct nonzero elements in $GF(q)$. Construct a matrix of the following form.

$$G_1 = \begin{bmatrix} y_1 & y_2 & y_3 & \cdots & y_n \\ y_1^2 & y_2^2 & y_3^2 & \cdots & y_n^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ y_1^m & y_2^m & y_3^m & \cdots & y_n^m \end{bmatrix}$$

where, $y_i \neq y_j$ and $y_i \neq 0$ for $1 \le i < j \le m$.

step2: Transform G_1 into a systematic matrix G using elementary row operations. An elementary row operation on a matrix is one of the following two operations: (1) multiply one of the rows by a nonzero scalar or (2) add a scalar multiple of one row to another.

It can be easily shown [4] that the matrix G generated by this algorithm is invertible.

3.3. Decoding Process

The decoding process may seem very time consuming. It requires the inversion of a matrix followed by a matrix-vector multiplication. However, notice that the inversion is done only once during the initialization process or when the system configuration changes. Notice that G' has the following form:

$$G' = \left[\text{column subset of } I^{m \times m} \middle| \text{ column subset of } P^{m \times k} \right]$$

.The columns of G' are partitioned into two parts. The first part is a column subset of $I^{m \times m}$. The number of columns in this part is equal to the number of operational data servers. The second part is a column subset of $P^{m \times k}$. This part contains at most same number of columns as the number of faulty servers. Since the number of faulty servers is much less than the number of operational servers, G' and its inverse G'^{-1} are sparse matrices. This sparsity can be exploited to obtain more efficient algorithms.

We can show [4] that the vector of faulty (missing) data blocks U_f can be computed by

$U_f = (V_s - \tilde{U_f} \times \tilde{Q_f}) \times Q_f^{-1}$ where,

V_s = vector of received parity blocks

$\tilde{U_f}$ = vector of received data blocks

$\tilde{Q_f}$ = rows of Q corresponding to non-faulty data servers

Q_f = rows of Q corresponding to faulty data servers

$Q = l$ check columns of G corresponding to parity servers

Notice that Q_f^{-1} needs to be computed only once for a certain system configuration. Because $Q_f^{l \times l}$ is much smaller than $G'^{m \times m}$.

4. Storage Reconfiguration Scheme of the RDSS

In this section, we describe the automatic mechanism of the RDSS for reconfiguring data storage.

4.1. The AutoFixer

The AutoFixer, as described briefly in the previous section, handles the recovering and redistributing of unavailable data blocks and is therefore the key component in the reconfiguration scheme. The user client also has an important role to play in the process of the reconfiguration. If the user client has trouble downloading data blocks from a server it lodges a *complaint* to the AutoFixer about the file consisting of the problematic data blocks. The AutoFixer logs the complaint in the log table.

A single report for a problematic file download does not necessarily mean that the corresponding data block servers were problematic or completely unavailable. It could be due to temporary network congestion or the data server being overloaded. Thus, the AutoFixer needs to decide whether to restore missing blocks or not. If a number of complaints against a file is greater

than a threshold value, the Autofixer initiates recovering the problematic file and redistribute the corresponding data blocks.

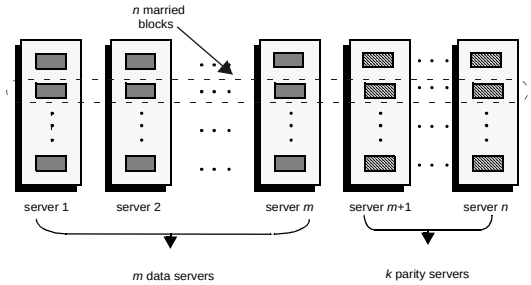

Figure 3 Married Block Concept

The recovery of a problematic file is achieved by a decoding algorithm. Redundant parity blocks are created and associated with the data blocks by an encoding algorithm when the file is striped and distributed over servers. The parity blocks are stored in the parity servers.

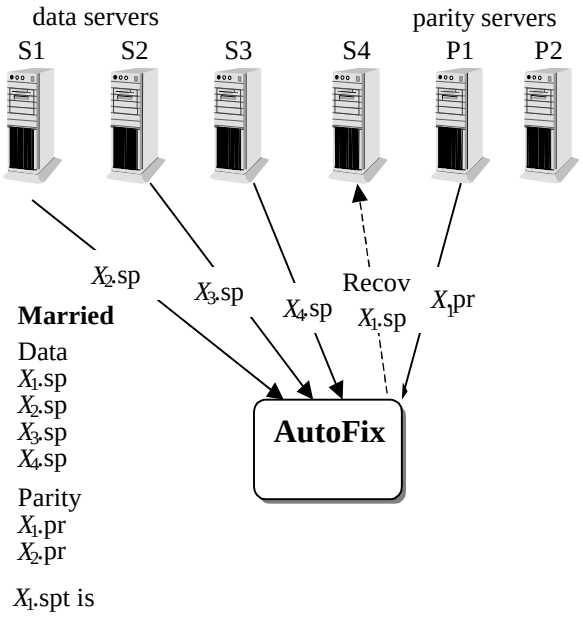

Figure 4 (a) Autofixing a lost block (Case 1)

Let n be the total number of servers and m be the number of data servers, $m<n$. The remaining $k=n-m$ servers are used to store parity blocks for data recovery (Refer to the Figure 3). The choice of m and k is a function of the desired storage overhead and desired degree of fault tolerance. A file object is first striped into p data blocks. For simplicity we assume that m divides p. Otherwise, dummy blocks can be added to make p a multiple of m. These p blocks are divided into groups of m blocks each. Each group is encoded to produce k redundant parity blocks. The m data blocks and their corresponding k redundant blocks are said to be married. Encoding is done in such a way that if any m out of n married blocks are available, the original m blocks can be reconstructed. In Section 4, we discuss the coding scheme central to the storage reconfiguration in more detail.

Once the missing blocks are recovered, the Autofixer distributes these to backup servers. The metafile is updated according to the new data distribution. The data storage of the RDSS is automatically reconfigured by the AutoFixer and therefore the whole reconfiguration process is transparent to the user.

4.2. Reconfiguration Protocol of the AutoFixer

The AutoFixer first receives complaints about a problematic file from the user client. It saves the complaints in the log table in (*file name, identification of the problematic blocks, metafile*) format. It initiates *log_table_analysis* thread on a periodic basis. The *log_table_analysis* thread determines which files should be fixed based on the frequency of the complaints. All the entries corresponding to the files to be fixed are extracted into *to_be_fixed* table. It then initiates the *file_fixing* thread. Based on the entries stored in the *to_be_fixed* table, the fixing thread analyzes the storage configuration of the data and parity block servers of each problematic file, and checks whether it can recover the file. Since a file consists of married group, each married group of the file needs to be examined whether it has any missing blocks and recoverable. The missing blocks are recoverable if the number of missing blocks in a married group is less than or equal to the number of parity blocks in the married group. The *file_fixing* thread then checks the status of the data servers (To ensure the server status, each server is examined repeatedly in a time interval). Based on the server status, the *file_fixing* thread handles the reconfiguration as follows.

(case 1) All servers are operational but some data blocks are deleted.

For each married group that has missing blocks, it determines the number of parity blocks to be downloaded. It then downloads all the necessary data and parity blocks to decode the missing blocks back. After the blocks are recovered, they are stored back onto the original data servers. Therefore the metafile is

kept intact. For example, as shown in Figure 4, X_1.spt, X_2.spt, X_3.spt, and X_4.spt are the data blocks of a married group of file X. Blocks X_1.prt and X_2.prt are the parity blocks of the married group. Assume file X consists of only one married group and X_1.spt file is missing. To recover X_1.spt, either X_1.prt or X_2.prt needs to be downloaded. After X_1.spt is recovered, it is stored back to the original server, S4.

(case 2) One or more data servers are down.

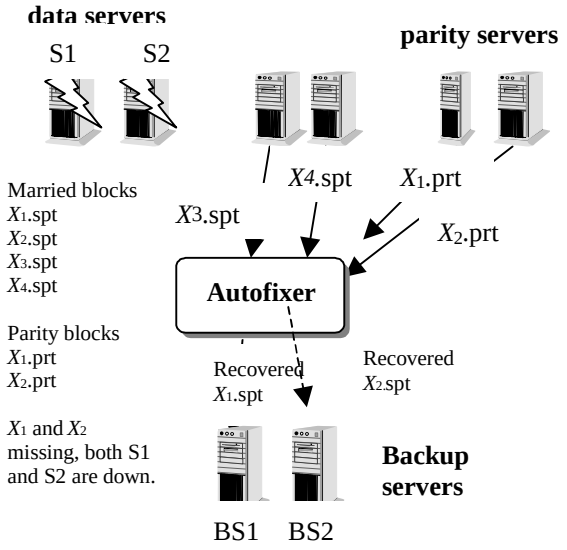

Figure 5 (b) Autofixing a lost block (Case 2)

The recovery of the missing data blocks is done by the same way outlined for the case 1. After data blocks are recovered, they are stored in backup servers. The metafile is then updated and restored in the master server. Figure 5 shows an example.

The protocol of the AutoFixer is depicted in the Figure 6.

5. Conclusions

In this paper, we have described a novel scheme and its prototype RDSS (Reconfigurable Distributed Data Storage System) for enabling highly available and dependable data storage and delivery service in a distributed computing and communication environment.

The RDSS employs multiple distributed data servers, as parallel data block servers, in order to better utilize the available computing, storage, and bandwidth resources. A file object is divided into blocks that are stored over distributed data servers. Efficient data transfer can then be achieved by using multiple links, established between the client and data servers, to transfer data blocks in parallel.

```
StaStart receive_complaints thread;
2. Start log_table_analysis thread;

Thread receive_complaints
 Open socket connection to listen from users' clients
 while (true) {
    Receive complaints from users' clients in
     (file name, metafile, list of problematic blocks);
    Save complaints in the log file;
    Update the frequency of the complaints,
       numofcomplaints, against same file.
 }
end // of receive_complaints thread

Thread log_table_analysis {
 for(i = 1 to number of entries in log table) {
   with the ith entry,
   if (numofcomplaints > f ) // predetermined frequency f
   then { add the ith entry to to_be_fixed table;
          delete entry from log_table;
   }
 }// end of for
 if to_be_fixed_table has been updated then
  start file_fixing thread;
end // of log_table_analysis thread

Thread file_fixing {
 repeat {
  for each entry in the to_be_fixed table do {
   get the configuration of the data and parity block servers
      from the metafile; // from the table entry
   get the married group of data and parity blocks of the file
      from the metafile; // from the table entry
   if ( the number of missing blocks of each married group
      is less than or equal to the number of parity blocks)
   then do  { // begin fixing
      check the status of the data servers
        in a predetermined time period;
      if all the servers are operational then { //case (1)
        for each married group that has missing block(s) do {
          determine the No.of parity blocks to be downloaded;
          download married data and parity blocks of the
            missing block(s);
          recover the missing data block(s) by decoding;
          restore the recovered block(s) into the
            corresponding data server;
        } // end of for each ...
      } // of if then.. case(1)..
      else if only some of the servers are operational then {
        // case (2)
        for each married group that has missing block(s) do {
          determine the No. of parity blocks to be downloaded;
          download married data and parity blocks of
            the missing block;
          recover the missing data blocks by decoding;
          restore the recovered block into backup data server;
          update the metafile;
        } // end of for each ...
        restore the updated metafile into the master server;
      } // of if then.. case(2)..
      delete the entry from to_be_fixed table;
   } // of if then .. begin fixing
   else leave the entry in the table
    //cannot be fixed at this time it will be reexamined later
 until (all entries in the to_be_fixed table are examined);
end // of file_fixing thread
```

Figure 6 The Protocol of the AutoFixer

The salient feature of the RDSS is the storage reconfigurability in the case of faulty data blocks. The reconfiguration of the storage is performed in order to recover missing data blocks by a coding method and to

restore them onto operational data or backup servers for continuous data availability. The reconfiguration process is performed on a periodic and automatic basis without user knowledge or manual intervention.

The RDSS has several advantages over traditional data storage systems. The advantages of our approach include: high data rates, scalability, availability, reliability, and seamless system controlled load balancing.

Acknowledgments: The work described in this paper has been in part supported by the DOD (DAHC94-96-C-0005/0010) and DOE (DE-FG02-97ER25339). The authors particularly would like to acknowledge Mr. Hemant Mahidhara and Mr. Satish Reddy for their great help in the implementation of the RDSS prototype.

6. References

[1] A.D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart, "The Echo Distributed File System", *SRC Research Report 111*, Systems Research Center, Digital Co., Palo Alto, CA, 1993.

[2] G.C. Clark, Jr., and J.B. Cain, *Error-Correction Coding for Digital Communications*, Plenum Press, New York, 1981.

[3] C. Jerian, G. Swart, A.D. Birrell, A. Hisgen, and T. Mann, "Availability in the Echo File System", *SRC Research Report 112*, Systems Research Center, Digital Co., Palo Alto, CA, 1993.

[4] Q.M. Malluhi and W.E. Johnston, "Coding for High Availability of a Distributed-Parallel Storage System", IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 12, pp. 1237-1252, December 1998.

[5] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S. Rosenthal, and F.D. Smith, "Andrew: A Distributed Personal Computing Environment", *Communication of ACM*, vol. 29, no. 3, March 1986.

[6] G.S. Jung, Q.M. Malluhi and W.G. Brown, "A Scheme for High Performance Data Delivery in the Web Environment", in the *Proceedings of International Conference on Parallel and Distributed Systems*, pp. 210 - 217, 1998.

[7] W.W. Peterson and E.J. Weldon, *Error-Correcting Codes*, 2nd ed., Cambridge MIT Press, 1972.

[8] T.R.N. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems*, Prentice-Hall, Inc., New Jersey, 1989.

[9] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, "CODA: A Highly Available File System for Distributed Workstation Environment," *IEEE Transactions on Computers*, vol. 39, no. 4, Apr. 1990.