A Robust Storage System Architecture

R. C. Good
rcgood@ccnga.uwaterloo.ca

G. V. Cormack
gvcormack@plg.uwaterloo.ca

C. L. A. Clarke
claclarke@plg.uwaterloo.ca

D. Taylor
dtaylor@ccnga.uwaterloo.ca

Technical Report: CS-95-10
Dept. of Computer Science
University of Waterloo, Waterloo, Canada

March 9, 1995

Abstract

Error-correcting codes allow either incorrect data to be corrected or missing data to be rebuilt. They are frequently used with communications channels to recover data lost through line noise and thus provide a 'noise free' bit pipe. Data can also be lost through hardware failure; for instance a disk crash. In case of hardware failure, we want a storage system that has the robustness and tunability of error-correcting codes in order to provide recovery of the lost data. This is especially so when dealing with systems involving a large number of disks as, when taken as a group, they are more error prone than single disks but are a vary practical way of building large data stores.

At present, the most common way to provide data recovery is straight duplication (mirroring) or a code able to detect single failures within a tightly coupled array of disks. A new prototype system has been designed and implemented which uses linear error-correcting codes to provide data storage over a loosely coupled distributed storage system. The fault-tolerance of this system can be varied by choosing the amount of redundant information stored.

Introduction

This paper discusses the design and implementation of a Robust Storage System (RSS) and is an overview of the author's Master's thesis work. The motivation for this project came from its potential integration with an ongoing multi-platform text-retrieval-database project (The MultiText Project [8]). One the goals of this project is to provide a very large text database capable of storing hundreds of gigabytes to terabytes or more of text and completing complex queries in seconds. This database is to be implemented on a small local-area network of workstation/server-class machines and have continuous availability.

Clearly, a robust storage system for both the database's index files and its raw text is needed. Given the intended environment, a storage system, implemented over a network of workstation/server-class machines, each with its own local disk storage, and taking into account the propensity of reads over writes seems a promising design direction. With the number of machines envisioned and the desire for continuous operation even with several machines unavailable, existing systems (e.g., TickerTAIP [5] or Swift/Raid [2]) appeared unsuitable.

Thus, a storage system using linear block error-correcting codes as its data-recovery mechanism was designed and implemented. This system provides a dynamically adjustable trade-off between data capacity and redundancy for a given size of data store. The fundamental difference between this system and a system such as standard RAID [4] is the loosely-coupled nature of the architecture and the core algorithm used to provide data recovery. In standard RAID *eXclusive OR* is used along with the self-identifying nature of disk failures to recompute data lost through such failures by computing the *xor* over all the remaining disks. In RSS, linear error-correcting codes and Rabin's Information Dispersal Algorithm [1] (IDA) are used as the basis for data recovery. The read/write performance is similar to distributed RAID systems such as Swift/RAID.

Motivation

The use of error-correcting codes (ECC) [6] as a data recovery mechanism is motivated by the observations that

- ECC's not only can correct misinterpreted bits but can also 'fill in' bits whose value is unknown (erasures),
- ECC's can be computed from parameters specifying the amount of redundancy required.
- ECC's compute data to be used in recovery and append it to the original data. This speeds decoding in the common case of no errors,

The Information Dispersal Algorithm due to M. O. Rabin uses matrices over finite fields to disperse arbitrary data over several machines so that an intruder needs to obtain a minimum number of these pieces to have access to the original data. These two algorithms are synthesised into a data-recovery algorithm which makes use of matrix algebra over finite fields.

This work presents two separate implementations of this data recovery algorithm which provide different capabilities. The first is a set of programs that use the algorithm to break apart data into a specified number of pieces and recombine a subset of those pieces to form the original data. The second is a prototype client/server system in which one client communicates with multiple servers. The servers compute and store pieces of the client data similar to the files described above. The client broadcasts the data to the servers on *writes* and requests and recombines the pieces to extract the original data on *reads*. Both of these systems are implemented in a way which simplifies debugging and development and allows for (at least a) factor of two reduction in the number of operations performed, compared to thier original implementations. Thus the potential throughput of these systems can be easily increased beyond the performance currently achieved.

Musing

One can view a memory bus, SCSI, Ethernet, a telephone connection and a taut piece of string all as functionally equivalent communications media (busses or networks), albeit with different operating parameters (bandwidth, latency). With this in mind, one could view multiprocessors with different degrees of coupling as groups of processors different 'distances' apart. (One could claim that the Internet is the largest single heterogeneous multiprocessor computer in existence.) This prototype client/server system can be said to run on a homogeneous distributed multiprocessor. Having the model of a distributed multiprocessor in mind influenced the design of this system.

Preliminaries

This section reviews finite fields and matrix algebra over finite fields in order to provide the groundwork which we use to discuss error-correcting codes and Rabin's Information Dispersal Algorithm. The choice of finite field is first described, followed by a description of matrix multiplication and inversion over finite fields. A simple error-correcting code is described as is IDA.

Algebra, finite fields

Recall that a field is a set of elements with the operations addition (+) and multiplication ($\cdot$), additive and multiplicative identities and inverses. Finite fields are fields with a finite set of elements. For the RSS system, the set of integers modulo p, $\mathbb{Z}_p$, where $p \in \{2^8+1, 2^{16}+1, 2^{32}+15\}$ can be used (these three numbers are prime.) For the purpose of dispersal and reconstruction, the data are interpreted as elements of the chosen finite field and are 1, 2 or 4 bytes in size as p is 2^8+1, $2^{16}+1$ or $2^{32}+15$ respectively. The present implementation uses $\mathbb{Z}_{257}$ but as mentioned above, other moduli can be used. Note that the sizes of these fields are larger than the number of values representable with the respective data units. This requires special treatment when storing the computed data units back to disk. This treatment is described in a later section.

Recall the definition of matrix multiplication: If $A = [a_{ij}]$, $B = [b_{ij}]$, $C = [c_{ij}]$ are $n \times l$, $l \times m$, $n \times m$ matrices with a_{ij}, b_{ij}, c_{ij} the $(i,j)^{th}$ component respectively, the product $C = A \cdot B$ is an $n \times m$ matrix with $c_{ij} = \sum_{k=1}^{l} a_{ik} \cdot b_{kj}$. This definition relies on no more than addition and multiplication. Thus, the a_{ij}, etc., can be elements of a finite field. Choosing them as such allows the use of the additive and multiplicative inverses when performing Gauss-Jordan elimination to calculate the inverse of a matrix. Matrix inversion is performed as part of the decoding stage of both IDA and RSS. Of course, the element 0 has no multiplicative inverse and so singular matrices can still arise.

Error-Correcting Codes

Traditionally, error-correcting codes have been applied to data to ensure its integrity when transmitted over a communications channel or stored on some media. Examples are error-correcting modems which can deliver uncorrupted data over a noisy telephone connection, and CD-ROMs which can be successfully read even if their surface is slightly scratched.

The specific type of error-correcting code being discussed in this section is the linear block code[3]. These are codes which apply matrix algebra over the finite field $\mathbb{Z}_2 = \{0, 1\}$. Typically these codes would be used in communication channels to encode bit streams for transmission. In this case the bit stream is interpreted as vectors of elements in $\mathbb{Z}_2$.

These codes can correct bit flips and erasures. Bit flips are errors caused when a particular bit changes state and is incorrectly read at the receiver. Erasures are errors where the value of a particular bit is unknown at the receiver.

A basic explanation of this method follows. A matrix of the form $D = [I_m C]$ is constructed where I_m is an $m \times m$ identity matrix and C is an $m \times k$ matrix. An important property of D is that any m columns are linearly independent.

If $V = (v_1 \ldots v_m)^T$ is an m-element data vector which is to be sent to the receiver, the vector to be transmitted is found by computing the product $M = D \cdot V = (v_1 \ldots v_m c_1 \ldots c_k)^T$. The $c_1, \ldots, c_k$ are the *checksum bits* and provide the redundancy needed to perform error correcting.

These codes are discussed more fully in [6, 3].

Rabin's IDA

Rabin's Information Dispersal Algorithm [1] is used to break apart data so that the pieces can be distributed to multiple sites (providing fault-tolerance) without risking the integrity of the data (providing security.) With the way the data is broken up, only a subset of the original pieces are required to reassemble the original data. The minimum number of pieces required to reassemble the data can be chosen and with fewer than the minimum number of pieces, the data remains unavailable. This algorithm is also based in finite fields, in this case $\mathbb{Z}_p$ where p is a prime number.

In this algorithm, the data to be manipulated are viewed as a sequence of data units of a particular number of bits (b). The field $\mathbb{Z}_p$ is chosen so that $p > 2^b$. This choice allows each data unit is viewed as an element of the field.

The goal is to disperse data to n sites with the property that any m of them can be used to reconstruct the data. To start, we construct a *dispersal matrix*, D. This matrix is formed by collecting a set of vectors such that any subset of size m is linearly independent. The method

$$D \cdot \mathcal{F} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \\ 1 & 5 & 25 & 125 & 111 \\ 1 & 6 & 36 & 216 & 11 \\ 1 & 7 & 49 & 86 & 88 \\ 1 & 8 & 64 & 255 & 241 \end{bmatrix} \cdot \begin{bmatrix} 100 \\ 97 \\ 116 \\ 117 \\ 109 \end{bmatrix} = \begin{bmatrix} 25 \\ 97 \\ 59 \\ 214 \\ 140 \\ 232 \\ 160 \\ 154 \end{bmatrix} = \mathcal{R}$$

Figure 1: A STANDARD 8×5 IDA DISPERSAL MATRIX AND CALCULATION.

$$\begin{aligned} D'^{-1} \cdot R' &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 5 & 25 & 125 & 111 \\ 1 & 6 & 36 & 216 & 11 \\ 1 & 7 & 49 & 86 & 88 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 25 \\ 97 \\ 140 \\ 232 \\ 160 \end{bmatrix} \\ &= \begin{bmatrix} 132 & 125 & 132 & 125 & 1 \\ 65 & 1 & 165 & 45 & 238 \\ 89 & 105 & 57 & 112 & 151 \\ 214 & 56 & 85 & 65 & 94 \\ 15 & 227 & 75 & 167 & 30 \end{bmatrix} \cdot \begin{bmatrix} 25 \\ 97 \\ 140 \\ 232 \\ 160 \end{bmatrix} = \begin{bmatrix} 100 \\ 97 \\ 116 \\ 117 \\ 109 \end{bmatrix} = \mathcal{F} \end{aligned}$$

Figure 2: IDA RECONSTRUCTION WITH ROWS 3, 4 AND 8 MISSING.

used in [1] is to choose n rows of the form: $(1 \alpha_i \alpha_i^2 \cdots \alpha_i^{m-1})$ where each of the α_i's is a non-zero distinct element of $\mathbb{Z}_p$.

The data to be dispersed is formed into an $m \times L$ matrix, $\mathcal{F}$, where L depends on the number of data units in the original data. The matrix product $D \times \mathcal{F} = \mathcal{R}$ is computed and each row of the resulting $n \times L$ matrix is sent to one of the n sites. Figure 1 illustrates a specific example of this calculation. For clarity, only the computation for one column of the data matrix and product matrix is shown in the examples. The remaining columns are computed in the same fashion.

We observe that each site contains a specific row from the resulting product matrix R. The i^{th} site holds the i^{th} row of the product matrix. This row can also be obtained by multiplying the i^{th} row of D by $\mathcal{F}$. Thus the i^{th} site can be associated with the i^{th} row of D.

The original data is recovered by obtaining the data from m sites and forming it into an $m \times L$ reconstruction matrix R'. A matrix D' is formed from the rows of D associated with each of the responding sites. (If each site stores its associated row in D, this step can be easily done.) The example in Figure 2 has rows 3 and 4 from R unavailable, thus using two of the three checksum rows. Observe that $R' = D' \cdot \mathcal{F}$. The original data is then recovered by computing $D'^{-1} \cdot R' = \mathcal{F}$.

As mentioned in [1], the dispersal and reconstruction steps reduce to computing inner products. For sufficient data, where $m \ll L$, the cost of computing D'^{-1} is non-critical.

RSS algorithm

This section describes the algorithm used in the programs and prototype system described below. The algorithm is a synthesis of the underlying ideas from error-correcting codes and IDA described above. The parameters that will be uused in discussing the RSS algorithm are introduced. The methods used to encode, store and decode data are explained. Finally, the technique used to reconstruct arbitrary missing pieces of encoded data is given.

The basic idea of this algorithm is to treat each of the columns in the product matrix in the IDA as a 'bit' vector in an error-correcting code; the main difference being that the ECCs are over $\mathbb{Z}_2$ and the IDA is over $\mathbb{Z}_{257}$.

Parameters

Some of the key parameters used in discussing this algorithm are as follows:

- n – the number of pieces of data produced by the encoding step. This corresponds to the number of rows in the dispersal matrix D and the resulting matrix R.
- m – the minimum number of pieces needed to reconstruct the original data. This corresponds to the number of columns in D and the number of rows in $\mathcal{F}$.
- k – the number of redundant pieces of data. Up to this many pieces of data can be lost without preventing full data recovery.

These three parameters are related by $n = m + k$.

By arranging that the parameters used to encode the data are stored with the resulting pieces of data at the various locations, we ensure that content of the pieces obtained from these locations is sufficient to reproduce the original data. No other parameters need be supplied to the decoding step.

Encoding

Like IDA, the RSS algorithm is based on finite-field matrix operations. In this case, the finite field is $\mathbb{Z}_p$ where $p \in \{2^8+1, 2^{16}+1, 2^{32}+15\}$, which are prime numbers. The choice of which finite field to use is driven by the size of the data unit used to group the data being encoded. The data units can be 1, 2 or 4 bytes in size and the prime number used should be just greater than the largest value representable in a data unit.

As in IDA, a dispersal matrix D_1 is formed by choosing rows of the form $(1\alpha\alpha^2 \cdots \alpha^{m-1})$ where each of the α's are distinct elements of the finite field being used. Notice that the choice of finite field limits the size of the largest matrix that can be generated and in particular the maximum number of encoded pieces of data.

In IDA the objective is to ensure the integrity of the dispersed data by making each fragment of the data unable to give up any of the original data. In the RSS scheme we would like

$$D \cdot \mathcal{F} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 252 & 10 & 247 & 5 \\ 5 & 233 & 45 & 217 & 15 \\ 15 & 187 & 126 & 152 & 35 \end{bmatrix} \cdot \begin{bmatrix} 116 \\ 97 \\ 99 \\ 107 \\ 121 \end{bmatrix} = \begin{bmatrix} 116 \\ 97 \\ 99 \\ 107 \\ 121 \\ 156 \\ 242 \\ 167 \end{bmatrix} = \mathcal{R}$$

Figure 3: A COLUMN REDUCED 8×5 DISPERSAL MATRIX AND CALCULATION.

$$C \cdot \mathcal{F} = \begin{bmatrix} 1 & 252 & 10 & 247 & 5 \\ 5 & 233 & 45 & 217 & 15 \\ 15 & 187 & 126 & 152 & 35 \end{bmatrix} \cdot \begin{bmatrix} 116 \\ 97 \\ 99 \\ 107 \\ 121 \end{bmatrix} = \begin{bmatrix} 156 \\ 242 \\ 167 \end{bmatrix} = \mathcal{C}$$

Figure 4: THE AMOUNT OF COMPUTATION CAN BE REDUCED BY COLUMN REDUCTION.

to have as much of the data as possible freely available and only use the 'checksum' pieces to reconstruct any of the original data which may be unavailable. Our synthesis is motivated by the observation that ECC's provide the desired 'checksum' property and the desired dispersal property is provided by IDA.

With this motivation, we can 'column reduce' D_1 to a matrix D with the form $\begin{bmatrix} I_m \\ C \end{bmatrix}$ where I_m is an $m \times m$ identity matrix and C is a $k \times m$ matrix formed from what is 'left over' after column reduction. Column reduction is performed by doing primitive column operations: i) exchange two columns, ii) multiply a column by a non-zero constant, and iii) add one column to a multiple of another column, replacing the first column. The goal of these column operations is to form the upper identity matrix.

With the matrix D in this form, we can perform the remainder of the IDA. This is shown in Figure 3.

We form the original data into a $m \times L$ matrix $\mathcal{F}$ and compute $D \cdot \mathcal{F} = R$. Notice that R is of the form $\begin{bmatrix} \mathcal{F} \\ \mathcal{C} \end{bmatrix}$ where $\mathcal{C}$ is the matrix containing the effect of the checksum computation. With the column reduction, we reduce the computation required to that of computing $C \times \mathcal{F} = \mathcal{C}$ where C and $\mathcal{C}$ have k rows, as shown in Figure 4.

The rows of R are stored either in separate files (in the case of the set of programs implemented) or on separate disks (in the case of the prototype client/server storage system implemented). The first m rows of R will sometimes be refered to as the 'clear data' and the last k rows as the 'checksum data'.

Masking

At this point we must deal with the problem of translating the rows of R into a sequence of bytes that can be written to a file or disk. In this discussion we assume the use of one-byte data units and operate in the finite field $\mathbb{Z}_{257}$. The extension of this technique to other finite fields is straightforward.

We define the set $\mathbf{2^8} = \{0, \ldots, 255\}$ i.e., the set of non-negative integers less than 2^8. Eight-bit bytes can be thought of as elements of $\mathbf{2^8}$ by the obvious translation. Since $\mathbf{2^8}$ is a subset of $\mathbb{Z}_{257}$ elements in $\mathbf{2^8}$ are also in $\mathbb{Z}_{257}$ and no translation is necessary. This was implicitly used in the initial steps of encoding. Going from $\mathbb{Z}_{257}$ to $\mathbf{2^8}$ presents the problem of what to do about the element 256 which is in $\mathbb{Z}_{257}$ but not in $\mathbf{2^8}$.

A simple 'escape' mechanism could be used. This could cause q elements of $\mathbb{Z}_{257}$ to require $2q$ bytes to represent. As well, the location of data (within the dispersed data) needed to reconstruct an arbitrary piece of original data could only be found by reading from the beginning of all the pieces of dispersed data. A solution with a smaller bound on the 'expansion' factor and allowing efficient 'random access' is clearly desirable.

The technique used is as follows. Let δ be the difference in sizes of the sets $\mathbb{Z}_{257}$ and $\mathbf{2^8}$ ($\delta = |\mathbb{Z}_{257}| - |\mathbf{2^8}|$). Let $N = |\mathbf{2^8}| - \delta$. For this discusion, $\delta = 1$ and $N = 255$. We take a block of at most N elements from a row of R. We scan this block[1] to choose one of the δ elements of $\mathbf{2^8}$ *not* appearing in the block (regarding $\mathbf{2^8}$ as a subset of $\mathbb{Z}_{257}$). Such an element exists because of the choice of N. This element we call the *mask*.

The elements in the block are interpreted as bytes except for the element 256 which is translated into the mask. The byte representing the mask is written out followed by the N bytes corresponding the the data in the block. To decode, we read in the byte representing the mask, then read in the N bytes representing the block; translating one-to-one except when the mask value is seen. The mask is translated into 256. By the method used to choose the mask, we know that any mask seen as data must represent the elements of $\mathbb{Z}_{257}$ that don't 'fit' into a byte. For example, the sequence $(100, 256, 0, 232, 256, 99)$ would be written out as the sequence of bytes $(1, 100, 1, 0, 232, 1, 99)$. The mask in this example is 1, chosen since it is the first unused byte value in the original sequence.

This method results in 255 elements being written in 256 bytes for an 'expansion' factor of less than 0.4%. By using larger finite fields, the expansion factor becomes even less. In addition, the location of bytes representing arbitrary elements in a row can be found with a straightforward calculation.

It should be noted that this masking need only be applied to the bottom k rows of R. The top m rows of R contain the original data and therefore will not contain the element 256.

Decoding

The original data has been encoded into separate pieces of data which are then stored in several different locations; either in files or on disks. Decoding a minimum-size subset of these pieces of data will reproduce the original data.

[1] If using two or four byte data units, some sort of hashing would be used to determine this element.

$$
\begin{aligned}
D_1^{-1} \cdot R_1 &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \mathit{15} & \mathit{187} & \mathit{126} & \mathit{152} & \mathit{35} \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ \mathit{5} & \mathit{233} & \mathit{45} & \mathit{217} & \mathit{15} \end{bmatrix}^{-1} \cdot \begin{bmatrix} 116 \\ \mathit{167} \\ 99 \\ 107 \\ \mathit{242} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \mathit{245} & \mathit{55} & \mathit{130} & \mathit{42} & \mathit{43} \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ \mathit{49} & \mathit{88} & \mathit{205} & \mathit{87} & \mathit{86} \end{bmatrix} \cdot \begin{bmatrix} 116 \\ \mathit{167} \\ 99 \\ 107 \\ \mathit{242} \end{bmatrix} = \begin{bmatrix} 116 \\ \mathit{97} \\ 99 \\ 107 \\ \mathit{121} \end{bmatrix} = \mathcal{F}
\end{aligned}
$$

Figure 5: RSS RECONSTRUCTION WITH ROWS 2 AND 5 MISSING.

First, observe that the first m pieces of data (i.e., the data representing the first m rows of the matrix R) are exactly the original data we want. Thus an attempt is made to obtain the first m pieces of data from their respective locations. If this is successful, the pieces of data are assembled together and the original data is obtained at minimal computational cost.

We now consider the situation in which one to k of the desired pieces are unavailable. An attempt is made to obtain the required number of pieces of data (of the remaining k checksum pieces potentially available) in order to have m pieces. Each piece of data obtained will contain a row from R as well as the associated row from D. The 'associated' rows are formed into an $m \times m$ matrix, D_1, and the subset of the rows of R is formed into an $m \times L$ matrix R_1. The matrix D_1 should be in the form of an identity matrix overlaid by the 'associated' rows and the i^{th} row of R_1 should be associated with the i^{th} row of D_1. This can be achieved with suitable row exchanges or by obtaining the data in a suitable order.

Observe that the matrix R_1 is also the product $D_1 \times \mathcal{F}$. Thus, computing $D_1^{-1} \cdot R_1 = \mathcal{F}$ yields the original data. Figure 5 gives an example of this calculation with rows 2 and 5 missing.

Notice that we only need to compute two rows of $\mathcal{F}$. In general, because of the form of D_1^{-1}, only as many rows as are missing need to be computed. Thus, the cost of recovering the data is proportional to the number of missing pieces of data.

If fewer than m pieces of data can be obtained, then complete reconstruction of the original data isn't guaranteed. However, unobtainable pieces of data can be 'filled in' with zeros and associated with an unused row of the original dispersal matrix D. The result will be partial reconstruction of the original data with one or more rows of zeros in place of the missing data. If there is sufficient redundancy inherent in the original data, some other method may be useful in reproducing the missing data.

Direct Reconstruction of missing pieces

It is possible to recompute up to k missing pieces of data directly, without needing to produce the original data as an intermediate step. The method is efficient and can be used to add more redundancy to existing data by adding another data ‘container’ (disk or a file) and reconstructing the ‘missing’ piece.

We assume that the missing piece of data is associated with some known row of the dispersal matrix. We observe that the missing row is the vector - matrix product formed by multiplying that row of the dispersal matrix by the matrix representing the original data. Of course we don’t have the original data – it is dispersed across the collection of data containers.

However, we know that the original data can be computed by collecting m pieces of dispersed data (from the m to n pieces available) and their associated rows. The associated rows are formed into an $m \times m$ matrix and inverted. The data is formed into an m row matrix and the product of the first and second matrices is computed. This product is the original data.

By substituting this matrix computation for the original data and doing some algebra we can find a matrix that when multiplied by the acquired data produces exactly the missing pieces of data.

If D is the dispersal matrix and $\mathcal{F}$ is the original data, then the n rows of the product $D \cdot \mathcal{F} = R$ are the data to be dispersed. If R' is an m row matrix formed from a subset of the rows of R and the m rows of D' are a subset of the rows of D (such that the i^{th} row of D' has the same row index in D as the i^{th} row of R' has in R), then the original data can be found by computing $D'^{-1} \cdot R' = \mathcal{F}$. If $\hat{D}$ contains the rows of D corresponding to the rows of R that are missing, the the missing rows can be found by computing $\hat{D} \cdot \mathcal{F} = \hat{R}$.

$$
\begin{aligned}
\hat{R} &= \hat{D} \cdot \mathcal{F} \\
&= \hat{D} \cdot D'^{-1} \cdot R' \\
&= \mathcal{D} \cdot R'
\end{aligned}
$$

where $\mathcal{D} = \hat{D} \cdot D'^{-1}$.

Thus, computing $\mathcal{D} \cdot R' = \hat{R}$ performs the desired reconstruction.

In practice, if reconstructing a piece of the clear data, one of the redundancy/checksum pieces would be ‘sacrificed’ and its storage space used to hold the reconstructed clear data piece. When the failed hardware is replaced, its space is used to hold the reconstructed checksum pieces. Over time this has the effect of migrating the clear data to the ‘oldest’ disks and the checksum data to the ‘newest’ disks. Clear data pieces are written to newer disks when their current disk fails. Thus there is an orderly progression of data through the generations of hardware. The goal would be to keep as much of the frequently used data completely in clear pieces to avoid the cost of decoding the checksum pieces. In addition, the cost of decoding one of the checksum pieces would be paid for only once; the first time data is requested which must be decoded. That cost can be amortised over the future accesses.

Programs & Prototype

This section describes the implementation of two separate systems which employ this encoding/decoding algorithm. The first of these systems is a set of programs which operate on files; encoding a specified file into a number of related files or decoding a subset of the related files into the original file. Another program will recompute any missing related files without first reproducing the original file. In this case the original file corresponds to the matrix $\mathcal{F}$ and the related files to the rows of R.

The second of these systems is a client/server system which implements the RSS encoding/decoding algorithm by storing a row of R on each server and having the servers perform the encoding in parallel. The client handles decoding the data obtained from the servers and reproducing the original data.

The performance of these systems suggests potential encoding/decoding speeds of at least 1MB/s.

Programs

There are three programs in this system: *disperse*, *recombine* and *reconstruct*. The encoding is performed by *disperse*, writing the encoded data in multiple files. The available files are read by *recombine* and a file containing the original data is written if possible. Any missing files can be rewritten with *reconstruct* by computing them directly from the available files.

When started, *disperse* is given the parameters n and m as well as a file name. A dispersal matrix is formed as described above and the bottom k rows are formed into the matrix D'. The matrix $\mathcal{F}$ is defined which has m rows and the smallest number of columns which give at least 1024 entries. Bytes are read from the original file (`foo`) and loaded into $\mathcal{F}$ in row-major order, ie. successive rows are filled in. The m rows of $\mathcal{F}$ are written out to the first m files `foo.000`, $\ldots$, `foo.`$m-1$. The remaining k files `foo.`m, $\ldots$, `foo.`$n-1$ are written from the rows of $D' \cdot \mathcal{F} = R'$. In addition to the rows of D' and R', the parameters n and m are recorded. This matrix loading and multiplication is performed iteratively on whatever data remains.

By recording these parameters with the output files, *reconstruct* needn't be given the parameters it will use; they can be inferred from the available files. Consistency checks are performed to ensure that all the data used was produced with the same values of parameters.

Prototype

The prototype client/server system is designed to be readily used in conjunction with an emerging multi-platform text-retrieval database system. This is a prototype of a system to provide robust storage for the raw text and index files of the text-retrieval database. The design of the storage system takes advantage of the existence of multiple server-class machines connected by a local-area broadcast network.

This system is accessible through an application library which provides an *open*, *close*, *read* and *write* interface. The current system provides only a flat name space for the data objects managed through this interface. Files can be *open*ed by name for reading or writing. A token is returned which is used in subsequent interactions. Data is transferred between application buffers and the server through *read* and *write* operations. The *close* operation invalidates the token and indicates that no further data is needed or to be provided.

The RSS client is implemented in an application library, allowing it to be ‘plugged-in’ to existing software such as *ftp* and *http* servers.

The client is responsible for both sending data to the servers (during *write*s) and requesting and decoding data received from the servers (during *read*s). The RSS client exists within an application running on one of the machines in this system. In addition, there can be multiple clients operating within the system.

Each machine in this system runs an RSS server which is uniquely identified by an integer id, starting at 1 and increasing sequentially. The server is responsible for computing and storing a row of the matrix R for some file. Which row the server maintains is determined by the client, as described below. The server is also responsible for replying to a client's request for a particular row associated with some file.

The clients and servers communicate by a message queue accessed with *enQueue()* and *deQueue()* routines. When a client enqueues a message, the message is broadcast to servers; all the servers can dequeue that message. Messages enqueued by servers are sent to the client addressed in the message and only dequeued by that client. Thus the clients communicate one-to-many with the servers and the servers communicate one-to-one back to the initiating client. The servers can be thought of as a single 'virtual' server running on a 'distantly coupled' multi-processor. In this view, there is the standard communication model at work between the client and 'virtual' server.

The queues themselves are implemented in memory shared between a communication process (called *comm*) and either the server or client. This process simply takes a message from its queue (having been placed there by *enQueue()*) and sends it on the network; it also receives a message from the network and places the message in its queue (to be removed by *deQueue()*). Shared memory is used to allow messages to be passed between *comm* and its partner without the cost of copying. If *comm*'s partner is a client, it sends a message via UDP broadcast. Otherwise, *comm*'s partner is a server and messages are sent via UDP to the host/port addressed in the message. Thus each server sees the same message from a client and each message received by a client can come from any server.

In addition, the utility programs *driver* and *R_cat* were written. The queue routines are used by *driver* to send control messages to the servers for debugging purposes and to collect statistics while the system is running. *R_cat* makes use of the application library to provide a function similar to the Unix utility *cat*. By default, data is read from *stdin* and written to *stdout* unless otherwise redirected with the -i and/or -o switches. The -i switch specifies an RSS input name; the -o switch specifies an output name. With the -o switch, *R_cat* can be placed at the end of a command pipeline; with -i, it can be placed at the beginning of a pipeline. The two switches can be used together to 'copy' a file from one name to another, within the system. This program was used to obtain performance measures.

In practice, there is to be an even 'striping' of the data and redundancy over all the disks in the system to avoid creating 'hot-spots'. For a write transaction ($open \cdot write* \cdot close$), a random integer in $[0, n-1]$ is chosen by the client and sent to the servers. The server adds this number to its unique id (also in $[0, n-1]$). The result (mod n) is the row of the result matrix the server is responsible for computing and storing.

Performance

In these tests, the elapsed time is measured with *csh*'s internal *time* command for *disperse*, *recombine* and *reconstruct* For the client/server system, the system routine getclock() was used from within *R_cat* to obtain elapsed times. The performance of the three programs are measured under various 'failure' circumstances.

Environment

The performance testing environment is as follows:

- 4 DEC αXP running OSF/1 1.3B,
- 150MHz, 64-bit,
- 64MB mem, 2GB disk,
- external SCSI

In addition, there is a 10Mb/s dedicated Ethernet connecting the four with one being a gateway to the campus network.

Surface plot description

To determine the programs execution speed over a range of parameters, tests were run which exercises the programs *disperse*, *recombine* and *reconstruct* in various failure circumstances. The values of the parameters n and k are varied in two grades: from 1 to 10 and from 10 to 60 in steps of 5 with the condition that $k < n$. The plots shown have the parameters n and k range from 10 to 60. This gives an idea of how the algorithm performs for larger parameter values. The *size* parameter is varied from one to five to 10 MB.

For each of these parameters, one of the programs is executed with those parameters and a certain environment. The environment contains a file of size *size*. The times for this program to execute a series of tests are measured the internal *time* command of *csh*. The *real* and *system* times are added and average over the set of tests with these parameters. The results are plotted as surfaces with the height above the n, k plane representing the execution time. Three surfaces are shown; corresponding to the one, five and 10MB file.

Results

The tests performed for a set of parameters are as follows:

- *disperse* a file of *size* MB into n pieces with k of them redundant (simulates a *write*, Figure 6),
- *recombine* the pieces to form the original file (simulates a normal *read*; Figure 7),
- *recombine* the pieces when one of the pieces holding original data is removed (simulates a small failure; Figure 8),
- *reconstruct* the missing pieces from what remains (simulates recovery; Figure 9),

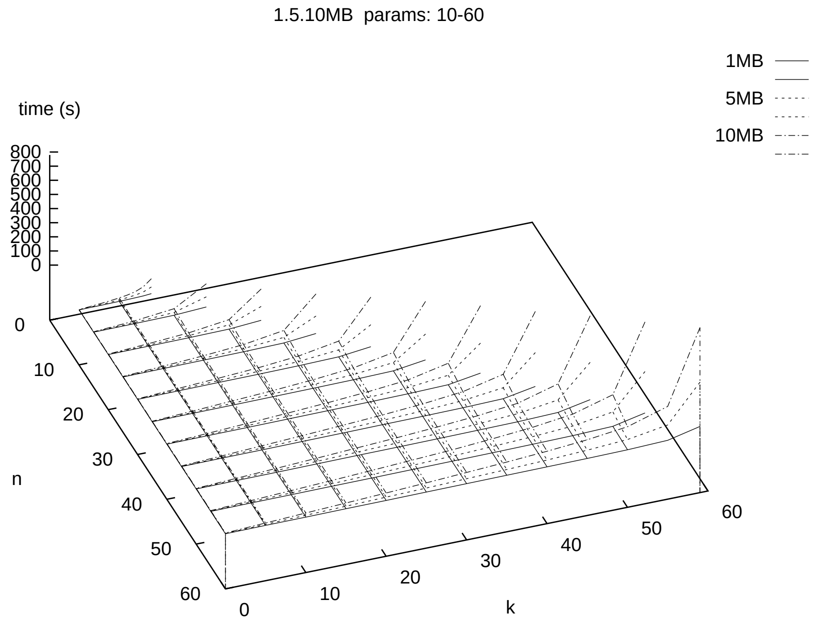

Figure 6: *disperse* 1, 5, 10MB.

- *recombine* the pieces when a maximum number (k) of the original data is removed (simulates a large failure; Figure 10),
- *reconstruct* the missing pieces from what remains (simulates recovery; Figure 11).

There were approximately 8600 samples collected.

The following six plots show the results of the above tests performed on a one, five and 10 MB file. The plots show three surfaces; the lowest corresponds the the 1MB file, the highest to the 10MBfile. The surfaces are plotted as a function of the parameters n and k. Attention is drawn to the second of the plots. This plot represents the common case of read with no failures[2]. The flatness of these surfaces indicate the throughput in the common case of reads with no failures is independent from the values of the parameters n and k used.

[2]failures are simulated by removing some of the pieces of data written by *disperse*.

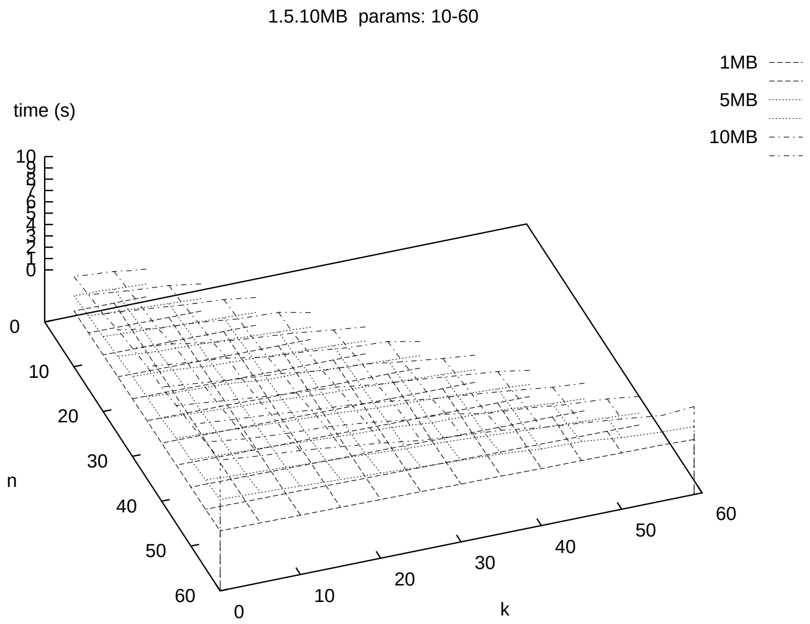

Figure 7: *recombine* NO FAILURES; 1, 5, 10MB.

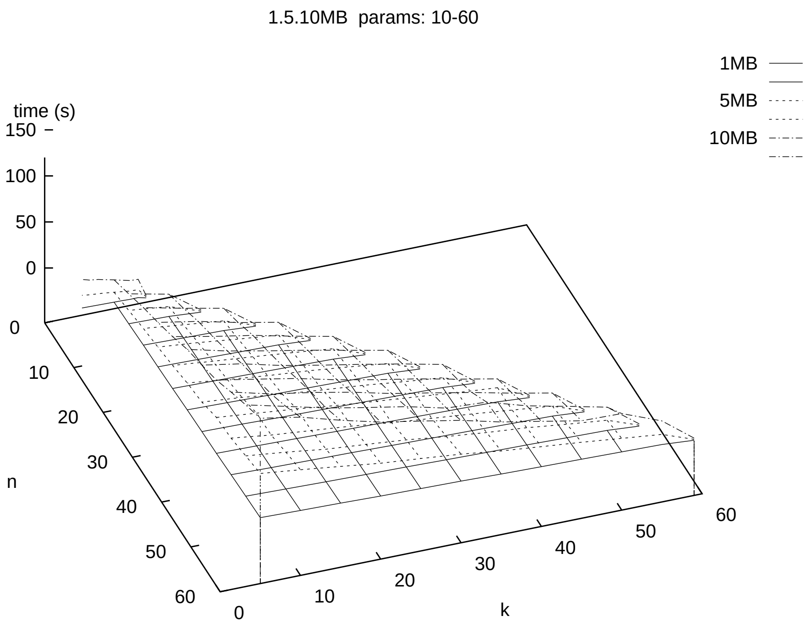

Figure 8: *recombine* SINGLE FAILURE; 1, 5, 10MB.

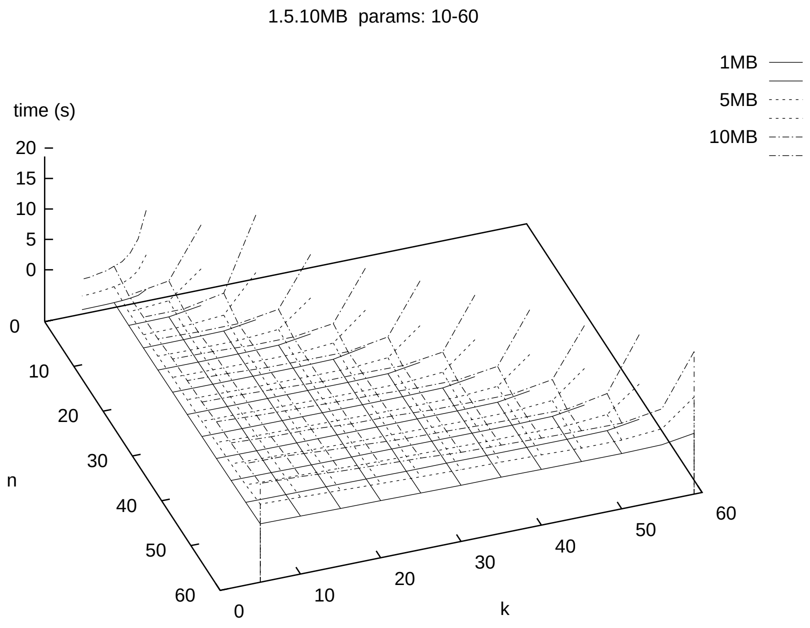

Figure 9: *reconstruct* SINGLE FAILURE; 1, 5, 10MB.

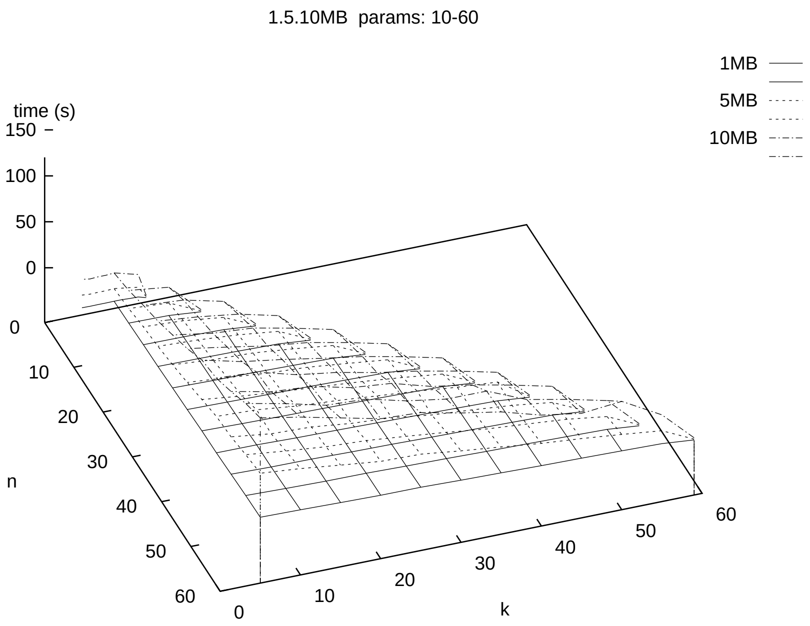

Figure 10: *recombine* MAXIMUM FAILURES; 1, 5, 10MB.

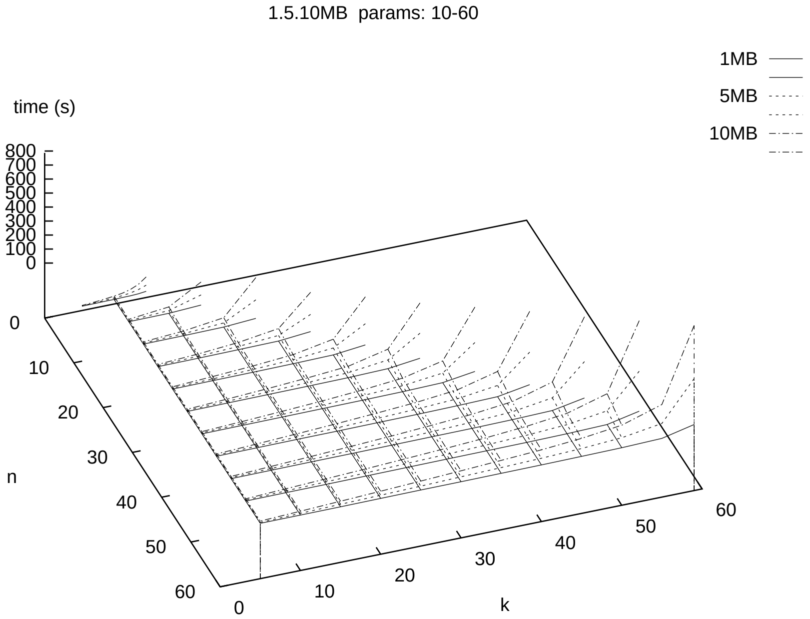

Figure 11: *reconstruct* MAXIMUM FAILURES; 1, 5, 10MB.

The client-server system

The RSS client program *R_cat* was used measure the performance of the storage system. This program was used to read from and write to files of various sizes using the data store provided by the servers. There was one server running on each machine and one of the servers computed the checksum data; three servers stored the 'clear' data. This corresponds to *disperse* writing out four files with one of the files being redundant. Two throughput test runs were performed. One test was with no 'failed' servers. With this test, the first three servers are able to supply the requested data without the client having to do any computation. The other was with a test where one of the first three servers was unavailable, forcing the client to use the checksum data from the fourth server to perform data reconstruction.

Figure 12 shows the throughput (in KB/s) to write files of various sizes with and without failed servers. Figure 13 shows the throughput when reading files under these conditions.

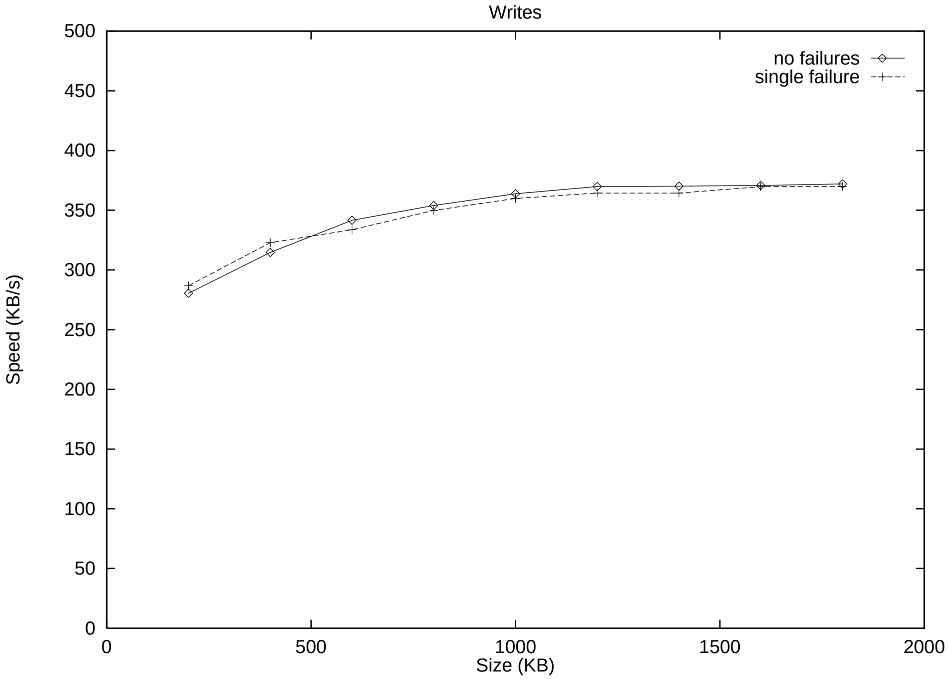

Figure 12: *R_cat* WRITE THROUGHPUT; WITH AND WITHOUT FAILURES.

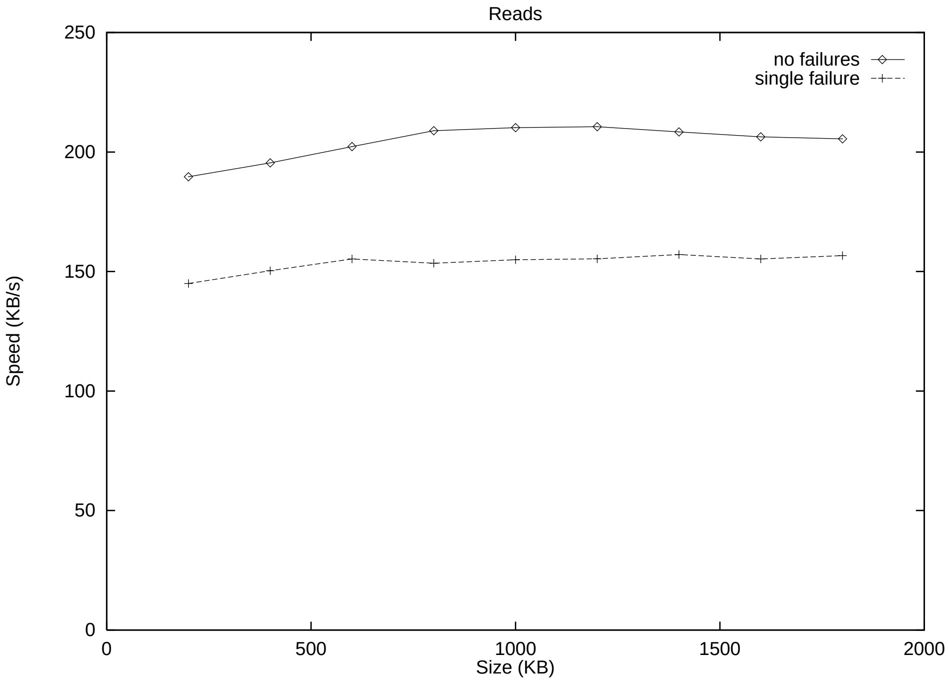

Figure 13: *R_cat* READ THROUGHPUT; WITH AND WITHOUT FAILURES.

Interpretation

The flatness of the surfaces in Figure 7 suggests that the performance of this data recovery algorithm does not depend on the values of the parameters n and k in the case of the first m data pieces being available. This is to be expected as this case involves no computation. Comparison of Figure 8 and Figure 10 show that the time to recover a file is largely independent of the number of missing pieces of the file.

Figure 13 shows the *read* throughput of the client/server system to be stable with respect to file size. The performance in the case of failure is lower, as expected. Together, these results suggest that a larger scale client/server system would be feasible.

Other Applications

In this section we discuss some potential applications for this data recovery algorithm. These are speculations on possible uses of this data recovery algorthim. These are not currently projects for future consideration.

Embedded application

This application involves using RSS between a disk controller and a number of disk drives. The intention is to provide a higher level protocol between the disk controller and disk drive; one which can preform data recovery when needed.

The RSS server is implemented in firmware on the disk drive unit; the RSS client is implemented in firmware on the disk controller. The client/server communications takes place over a SCSI bus. The result would be a SCSI adapter/RSS client that supports up to seven disks/RSS servers. The disk controller presents an interface providing one virtual disk, 20-30 GB in size. If the disks are 'hot swappable', data can continue to be accessible during maintenance to replace failures.

RAID-like archive server

The traditional methods of 'backing up' a system are becoming less feasible as the size of the typical data store increases at a faster rate than the transfer rate of tapes is increasing. At 400KB/s, data on a 5GB disk drive would take about $3\frac{1}{2}$ hours to be written to tape. If there was a 10:1 ratio of disks drives to tape drives (it could easily be higher), a large disk array could take a day and a half to fully back-up to tape. Maintaining continuous availability while properly handling updates to the data stored further complicates matters. Perhaps the whole concept of 'back-ups' needs to be re-thought.

There are two separate concerns. First, protecting against data loss through hardware failure. Second, protecting against data loss through software or user error. The latter can be best dealt with through enhancements to existing file system models, logging or immutable file systems for instance. The former concern can be addressed by building such enhanced file systems on top of fault-tolerant hardware using the RSS model to provide data recovery.

A cluster of processors can be built which provides a level in the data storage hierarchy which falls between secondary storage (disks) and off-line storage (tapes). Sufficient capacity and throughput (due to the parallel nature of RSS writes) could be obtained using optical media that off-line tape storage needn't be necessary. Hence, on-line archival storage.

Geographically distributed data repositories

There is no inherant reason for the hosts taking part in the storage server to be in close proximity. The principal reason they are is to take advantage of a local-area broadcast network for performance reasons.

By emulating a broadcast network on existing network topologies and placing the RSS servers at geographically distributed sites, a wide-area data repository which is fault-tolerant in the face of complete site failures can be constructed. In essence the design would be logically similar to the above archive server but because of the differences in interconnect bandwidth, care needs to be taken in the selection of the target environment.

Parallel tape storage

By combining the function of programs like *disperse* and friends with the operation of a robotic tape drive (actually, any type of 'juke-box'), a 'virtual tape drive' with 100GB capacity can be built[3]. When a given tape fails because of age or defect, a blank tape can replace it and the equivalent of *reconstruct* performed. Such a system could provide high capacity on-line storage for cents per megabyte.

Hierarchical Organisation

The RSS client presents an interface with *open/close*, *read/write* functionality. The RSS server expects to interface with a data-storage unit providing the same interface. Thus a straightforward hierarchical organisation of RSS clients and servers can provide an arbitrarily large data storage unit. On-line storage of multi-terabyte quantities of data would be feasible. Such a system could be assembled in 1995 at a cost of one to two million dollars, giving storage costs on the order of dollars per megabyte.

Directions

There are a number of direction for future development. They include

- integration with the text-retrieval database system making each text-retrieval engine a client of the storage system;
- extension of the client/server message set to implement a complete file system for general purpose applications;

[3] Rumour has it [7] that a 200GB tape drive that uses VHS video tape will soon be available. Add a VHS tape robot and increase the 100GB number by a factor of 40 to 4TB.

- increasing the data unit used from one byte to two or four. This will halve or quarter the number of multiplications and additions required to encode/decode[4] the data, increasing the performance accordingly;
- improving the UDP transport mechanism;
- finite-field computations could be either done in or assisted by hardware, improving performance.

Summary

The prototype presented implements a fault-tolerant, distributed-storage system. The prototype system is implemented on a small local-area network containing a group of server-class machines. Data recovery is made possible by computing a set number of 'checksum' blocks and storing them along with the original data. The number of checksum blocks computed determines the number of lost blocks that can be tolerated. The data-recovery algorithm is based on linear algebra over finite fields, error-correcting codes and Rabin's Information Dispersal Algorithm.

The parameters controlling the number of pieces produced by dispersal can be varied over a wide range. The system presented can easily adapt to different numbers of nodes automatically.

The recovery algorithm is optimised for the common case of read accesses with no failures by performing 'column reduction' on the dispersal matrix. Performance can also be enhanced by using a 'larger' underlying finite field.

There are two separate implementations of this algorithm. The first is a set of programs which operate on files and produces a (specified) number of file fragments; a subset of which can reproduce the original file's data.

The second is a client/server system implemented on a small local-area network of server-class machines. This system stores the data fragments on different servers. When a server is unavailable, any data it stored can be obtained from the remaining servers. This client/server system could be hierarchically arranged to provide an on-line archival storage service of terabyte capacity and greater. An application library provides the programming interface to the client/server system. This allows existing server such as *ftpd*, *nntpd*, *httpd*, etc. to be easily modified to use this storage system.

[4]if the checksum data needs to be used to recover the original data.

References

[1] Michael O. Rabin, *Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance*, Journal of the ACM, **36**:2 pp.335-348, April 1989.

[2] Darrel D. E. Long and Bruce R. Montague, *Swift/RAID: A Distributed RAID system*, USENIX Association Computing Systems, **7**:3, Summer 1994.

[3] Shu Lin, *An introduction to error-correcting codes*, Prentice-Hall, ch.3, 1970.

[4] Randy H. Katz, Garth A. Gibson, and David A. Patterson, *Disk System Architectures for High Performance Computing*, Technical Report UCB/CSD 89/497, UC Berkeley, CA, 1989.

[5] Pei Cao, Swee Boon Lim, Shivakuar Venkataraman and John Wilkes, *The TickerTAIP Parallel RAID Architecture*, ACM TOCS **12**:3, August 1994.

[6] Scott A. Vanstone and Paul C. van Oorschot, *An Introduction to Error Correcting Codes with Applications*, Kluwer Academic Publishers, 1989.

[7] Geoffrey Rowan (*growan@globeandmail.ca*), *The Globe and Mail*, Thompson Newspapers Co. Ltd., p. B5, March 4, 1995.

[8] G. V. Cormack F. J. Burkowski, *The MultiText Project*, University of Waterloo, 1994, 1995.
URL: *ftp://plg.uwaterloo.ca/pub/mt/*