Error Detection, Correction and Erasure Codes for Implementation in a Cluster File-system

Steve Baker

December 6, 2011

Abstract. The evaluation of various error detection and correction algorithms and erasure codes, particularly the Reed Solomon algorithm for suitability in a cluster file-system.

1 Introduction

In a cluster file-system, several layers of error detection and correction already exist at the hardware (disk/network) and OS/software (TCP/IP protocol) layers and it is not necessary to be overly concerned with data corruption of received and stored data, however, failures in those systems can and do happen as well as failures of devices to respond entirely, which is essentially the same as a transmission error. The latter eventuality is likely and must be dealt with at the file-system layer. In this paper we will be examining various algorithms typically used for error detection and correction with respect to erasure recovery. These algorithms will be evaluated for fitness in implementation in a cluster file-system based on the criteria of 1) speed of encode/decode, 2) space usage and 3) effectiveness.

2 A Short Introduction to Coding Theory

We first start with a simple example that illustrates the limits of error detection and correction, using one of the simplest codes – the binary repetition code of length 3 (simply repeat the same bit 3 times), we start with 2 codes representing 0 and 1:

\[ C = \begin{cases} 000 & = 0 \\ 111 & = 1 \end{cases} \]

We can clearly see that for any single error or missing bit, it is possible to determine which code was intended as we can determine the intended bit by majority vote. For two errors, it becomes impossible to know which code was intended, but it is still known that an error has occurred. For three errors the original codeword has been completely transformed from one to the other and it becomes impossible to even know that an error has occurred. In the case
of erasures, where the locations of the errors are known, then we can still recover the original message, even with two erasures.

The (Hamming) distance between two codes is the number of places that are different between the two codewords [H96]. In the above example we have \(d(000, 111) = 3\). The minimum distance of a code is the smallest distance between any two codewords in the code. Given the minimum distance we have the following theorem:

**Theorem 2.1.** [H96] Given a code \(C\) with a minimum distance \(d\):

i) Up to \(t\) errors can be detected in any codeword with distance \(d \geq t + 1\).

ii) Up to \(t\) errors can be corrected in any codeword with distance \(d \geq 2t + 1\).

A few more examples to support the above theorem:

\[
\begin{align*}
C_2 &= \begin{cases} 
00 = 0 \\
11 = 1
\end{cases} & C_3 &= \begin{cases} 
001 = 0 \\
110 = 1
\end{cases} & C_4 &= \begin{cases} 
0000 = 0 \\
1111 = 1
\end{cases} & C_5 &= \begin{cases} 
00000 = 0 \\
11111 = 1
\end{cases}
\end{align*}
\]

First, looking at \(C_2\), we can easily see that if a single bit is changed, we can tell that an error has occurred, but lack sufficient information to know which code was intended, so cannot correct it. If we know the error location however, we can correct it. With careful inspection of \(C_3\), it is possible to see that is is no different than the first repetition code example, and can detect two errors and correct for one, just the same. With the \(C_4\) example we can see that we can detect up to 3 errors, however we can still only correct for one error. If two errors were to occur, it would still be impossible to know which code was intended. To correct for two errors, we require a code with a minimum distance of \(2 \cdot 2 + 1 = 5\), such as \(C_5\) which we can see that even with 2 errors, it is possible to know which code was intended as there will still be a majority of correct bits by which to pick the correct code.

### 2.1 More Complex Codes

If we want to encode more information in the bits available we will need more redundancy in our codewords. In general for every bit or symbol of added redundancy in a code, we can divine an additional error. For error correction codes for each two additional symbols it is possible to recover from an additional error, as we will show below, for each error to be recovered, we must discover both the error magnitude and its location – two pieces of information. For an erasure code, we know the locations of the errors, thus the problem is halved and is thus possible to recover from as many erasures as we have additional redundancy.

A simple linear \([10,8]\)-code in \(GF(11)\) or \(Z_{11}\) [H96] will be used for illustration. The "\([10,8]\)-code" denotes that there are 10 symbols in total in the code of which 8 carry information. The other two symbols are the check-symbols used to detect errors and correct for one. Because the order of the field is a prime, all arithmetic is carried out modulus 11. The code utilizes a parity check matrix \(H\) of the form:
We can generate a generator matrix by transforming the parity check matrix into a standard form by using the following theorem: [H96]

**Theorem 2.2.** If \( G = [I_k | A] \) is the standard form generator matrix of an \([n - k]\)-code \( C \), then a parity-check matrix for \( C \) is \( H = [-A^T | I_{n-k}] \). Thus:

\[
G = \begin{bmatrix}
1 & 0 & a_{1,1} & \cdots & a_{1,n-k} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 1 & a_{k,1} & \cdots & a_{k,n-k}
\end{bmatrix}
\]

\[
H = \begin{bmatrix}
-a_{1,1} & \cdots & -a_{k,1} & 1 & 0 \\
\vdots & \ddots & \vdots & \vdots & \vdots \\
-a_{1,n-k} & \cdots & -a_{k,n-k} & 0 & 1
\end{bmatrix}
\]

And so by transforming \( H \) using elementary row operations, we yield \( G \), like so:

\[
H = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{bmatrix}
\]

\[
(r_1 \rightarrow r_1 + r_2) \rightarrow \begin{bmatrix}
2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{bmatrix}
\]

\[
(r_1 - (1) r_1) \rightarrow \begin{bmatrix}
9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1
\end{bmatrix}
\]

\[
(r_2 \rightarrow r_2 - 2r_1) \rightarrow \begin{bmatrix}
9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 0 & 1
\end{bmatrix}
\]

So if \( D = \{d_1, d_2, \ldots, d_8\} \) are our data words, then \( D \cdot G = C \) where \( C = \{d_1, d_2, \ldots, d_8, (2d_1 + 3d_2 + \cdots + 9d_8), (8d_1 + 7d_2 + \cdots + d_8)\} \). If \( x = \{x_1, x_2, \ldots, x_{10}\} \) is the original codeword and \( y = \{y_1, y_2, \ldots, y_{10}\} \) is the received code, then the **syndrome** is calculated as:

\[
(A, B) = y \cdot H^T = \left( \sum_{i=1}^{10} y_i, \sum_{i=1}^{10} i \cdot y_i \right) \pmod{11}
\]

If no errors are encountered, then \((A, B) = (0, 0)\), this can be understood by examining how \( A \) and \( B \) are computed. For \( d_1 \) for instance we add together \((d_1 + 8d_1 + 2d_1) = 11d_1\), for \( d_2 \) its \((d_2 + 7d_2 + 3d_2 = 11d_2) \) and so on. Each data word should add to 11\(d_i\). In \( \mathbb{Z}_{11} \), 11 times anything will always equal 0. So the two check symbols should cause the syndrome to compute to zero when the received vector is multiplied through the check matrix, anything else implies an error.

Supposing a single error has occurred, then we have for some non-zero \( j \) and \( a \) the following:

\[
(y_1, y_2, \ldots, y_{10}) = (x_1, \ldots, x_{j-1}, x_j + a, x_{j+1}, \ldots, x_{10})
\]
Then:

\[ A = \sum_{i=1}^{10} y_i = \left( \sum_{i=1}^{10} x_i \right) + a \equiv a \pmod{11}, \]
\[ B = \sum_{i=1}^{10} iy_i = \left( \sum_{i=1}^{10} ix_i \right) + ja \equiv ja \pmod{11}. \]

The error magnitude is given by \( A \), and the location of the error is found by dividing \( B \) by \( A \) (or \( B \cdot A^{-1} \)). This simple example shows the essence of syndrome decoding. If we were to have two errors we would need to find the error magnitudes \( a + b \) and the error locator’s \( ja + kb \). Two equations is not enough to resolve 4 variables, however if the error locations \( j \) and \( k \) were known, then it becomes possible to solve the system of linear equations to find \( a \) and \( b \) with just the two equations. A solution should obtainable and be unique, so long as the number of variables exactly matches the number of linearly independent equations. Thus erasure decoding, where the error locations are know, lets us recover more errors than error correction alone can, because we have more information in the form of the error locations.

The distance of the above code can be derived directly from its parity-check matrix \( H \), which in this case is 3. For the distance of a code from its parity-check matrix we have the following theorem: [H96]

**Theorem 2.3.** Suppose \( C \) is a linear \([n, k]\)-code over \( GF(q) \) with parity-check matrix \( H \). Then the minimum distance of \( C \) is \( d \) if and only if any \( d - 1 \) columns of \( H \) are linearly independent but some \( d \) columns are linearly dependent.

We can see from the code above that any two columns are linearly independent, but some 3 columns are linearly dependent. For example \((-1) \cdot [1] + 2 \cdot [2] = [3] \)

### 3 Some Simple Codes

#### 3.1 Repetition Codes

The simplest system is simply repeating the data, preferably 2 or more additional times. In a file-system context this means copying the same data to multiple locations, which can then be retrieved and compared. If one copy cannot be retrieved, one can use the other copies, so as an erasure code, it works well and there is no computational overhead. This is in essence RAID level 1 and the mechanism used in Google’s cluster file-system. While very simple to implement, it is highly inefficient in space. *Triple Modular Redundancy* is a form of this utilized in some memories, where the data used is decided upon by majority vote.

With only two devices (or one singular datum with additional redundancy), all schemes reduce to a repetition code.
3.2 Parity

Often a single bit that is added to a stream of n bits to indicate that the number of set bits is either even or odd. This can detect an odd number of errors (but not an even number) and can be used to replace a single missing bit provided the others are known. Parity has been widely used in RAID systems (level 3-5) as an efficient erasure code to reconstruct missing data.

Given $n$ data disks, the parity $P$ is generated by xor'ing the values of the data disks together, giving us:

$$P = D_1 \oplus D_2 \oplus D_3 \oplus \cdots \oplus D_n$$

Restoring data for a lost data drive $i$ involves computing the parity of the remaining drives, which we’ll call $P_x$ and xor’ing that with the original parity. The value of our lost data is then:

$$D_i = P \oplus P_x = P \oplus D_1 \oplus \cdots \oplus D_{i-1} \oplus D_{i+1} \oplus \cdots \oplus D_n$$

To correct for more than one error using parity one can attempt to employ other parity-based schemes which use more parity bits, such as arranging data into groups, each with their own parity, or arranging the data into a two dimensional array, with parity for each row and column. Such parity distributions cannot reliably handle more than 1 erasure in one dimension and 3 in two dimensions. Unfortunately for $m$ parity bits, there is some failure arrangement of $k$ data disks $< m$, that the system cannot recover from.

One mechanism for efficient recovery of up to two erasures is the "EVENODD" parity scheme. The algorithm can not be scaled beyond two erasures so we don’t consider it a good extensible erasure code, but for two is it more efficient than the typical Reed-Solomon mechanism to be described below.

3.3 Hamming Codes

One of the earliest error correcting codes, Hamming Code is a linear error-correcting code named for its inventor Richard Hamming. This code can detect up to 2 contiguous bit errors and correct a single-bit error[HC]. For every $m$ parity bits used, $2^m - m - 1$ data bits can be so protected[HC]. The above [10,8]-code is in fact a "doubly-shortened" Hamming code with a more efficient decoder than a normal Hamming code. Hamming codes are used in ECC memory and form the basis for the rarely used RAID 2 system, where it can recover from one drive failure and recover corruption when and only when the corresponding data and parity is good. Hamming codes, while interesting, probably are not a good candidate for an extensible erasure code.

4 The Reed-Solomon Algorithm

Developed by Irving S. Reed and Gustave Solomon, the Reed-Solomon coding scheme is a linear block code that does not work on bits, but symbols (read bytes), and is widely used to detect and correct for errors in CD and DVD media. As a error detection and correction code, given
additional symbols added to data, Reed-Solomon can detect up to \( m \) errors and correct up to \( \lfloor m/2 \rfloor \) errors. As an erasure code it can recover up to any \( m \) erasures\[P96\], making it an optimal erasure code, and so it is of particular interest to us. As a linear block code, Reed-Solomon is a \([n, k, n - k + 1]\) code where \( n \) is the length of the code (the number of data elements plus the number of check symbols we wish to add (i.e. \( n + m \) in our case), \( k \) is the dimension of the code which in our case is \( n \) or the number of data elements, and \( n - k + 1 \) is the Hamming distance of the code.

Originally Reed-Solomon is viewed as a code where an overdetermined system is created by oversampling the input and then encoding the message as coefficients of a polynomial (over a Finite Field). Recovery of the original message remains possible so long as there are at least as many remaining points in the system as were in the original message.[RS]

Traditionally however RS is viewed as a cyclic code where the message polynomial is multiplied by a cyclic generator polynomial to produce an output polynomial where the encoding check symbols are the coefficients. These coefficients can then be used to check for errors with polynomial division with the generator polynomial. Any non-zero remainder indicates an error. The remainder can then be used to solve a system of linear equations known as syndrome decoding (a simple version of which we demonstrated above) often using the Berlekamp-Massey algorithm to find the error locations and magnitudes by which it may be able to reconstruct the original message.[RS]

Because we are most interested in using Reed-Solomon as an erasure code, which implies that the error locations are known we can employ a simpler method of encoding and decoding using a properly formed information dispersal matrix (one that is formed from manipulating a Vandermonde matrix so that it is invertible via Gaussian Elimination). First we will discuss Galois Fields, in which all arithmetic will take place.

4.1 Galois Fields

For many codes, Reed-Solomon included, it is necessary to perform addition, subtraction, multiplication and particularly division without introducing rounding errors or a loss of precision, and since the resulting codes often need to be in a specific integer range – modulus arithmetic would seem to be required. However in normal modulus arithmetic, division is problematic. Rings of size prime would solve that issue, but fail to allow us to efficiently use our data storage. Arithmetic in a Galois Field where \( q = 2^h \) however provides us with a way to accomplish these feats.

Galois Fields or Finite Fields (denoted as \( GF(q) \) or \( \mathbb{F}_q \) respectively) are fields that contain a finite set of elements \( q \) (which is called its order). The order of a field must be a prime power where \( q = p^h \), \( p \) being a prime and \( h \) a positive integer, and the identity elements 0 and 1 must exist.[H96]

Galois Fields have the following properties:[H96]

(i) \( \mathbb{F} \) is closed under addition and multiplication, i.e. \( a + b \) and \( a \cdot b \) are in \( \mathbb{F} \) whenever \( a \) and \( b \) are in \( \mathbb{F} \).
(ii) Commutative laws: $a + b = b + a$, $a \cdot b = b \cdot a$.

(iii) Associative laws: $(a + b) + c = a + (b + c)$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

(iv) Distributive law: $a \cdot (b + c) = a \cdot b + a \cdot c$.

(v) $a + 0 = a$ for all $a$ in $F$.

(vi) $a \cdot 1 = a$ for all $a$ in $F$.

(vii) For any $a$ in $F$, there exists an additive inverse element $(-a)$ in $F$ such that $a + (-a) = 0$.

(viii) For any $a \neq 0$ in $F$, there exists a multiplicative inverse element $a^{-1}$ in $F$ such that $a \cdot a^{-1} = 1$.

Because of (vii) and (viii) we have subtraction and division respectively, understanding that $a - b = a + (-b)$ and $a \div b = a \cdot b^{-1}$ for $b \neq 0$.[H96]

We also have the following properties:[H96]

(i) $a \cdot 0 = 0$ for all $a$ in $F$.

(ii) $a \cdot b = 0 \Rightarrow a = 0$ or $b = 0$. (Thus the product of two non-zero elements of a field is also non-zero.)

The values of the field or residue class ring are represented by polynomials, composed of primitive elements, having degree less than a generator polynomial $g(x)$ which is irreducible, i.e. there do not exist polynomials $a(x), b(x) \in F$ both with degree less than $g(x)$, such that $g(x) = a(x)b(x)$. [vL99]

For our purposes $p$ will always be 2, $h$ will be either 8 or 16. The limiting factor is the necessity of maintaining tables used for calculating the results of multiplication and division. Beyond 16, tables would become too large to efficiently store. Addition and subtraction of polynomials over $GF(2^h)$ is essentially the xor operation on the two elements. Thus: $a + b = a \oplus b$, $a - b = a \oplus b$.

Multiplication in a field is essentially polynomial multiplication. To optimize multiplication and division we’ll employ a set of two tables that contain the log and inverse log of each element. To multiply, the logs of the two elements are added together (mod $2^h$) and then the inverse log of that value is the result. To divide, the logs are subtracted instead.

To generate our tables, we use the algorithm listed in figure 4.1. Genpoly is the generator polynomial value $g(x)$ expressed as a vector in binary form. For example, in $GF(2^4)$, $g(x) = x^4 + x + 1$, and as a vector (1,0,0,1,1). Substituting 2 for $x$, we get $2^4 + 2 + 1 = 19$. For $GF(2^8)$, $g(x) = x^8 + x^4 + x^3 + x^2 + 1$ or 0b10001101.
b = 1;
for (log = 0; log < (1<<h)-1; log++) {
    glog[b] = log;
    gilog[log] = b;
    b = b << 1;
    if (b & (1 << h)) b = b ^ genpoly;
}

Figure 1: Code for generating logarithm tables[P96].

Once we have produced our log-tables, multiplication becomes:

\[ r = gilog \left[ (glog[a] + glog[b]) \mod 2^h \right] \]

And division becomes:

\[ r = gilog \left[ (glog[a] - glog[b]) \mod 2^h \right] \]

For \( a, b > 0 \). If \( a \) or \( b \) equals 0, then 0 is the given result, \( b = 0 \) in division indicates an error.

Table 4.1 shows an example for \( GF(2^4) \). Note that the first rows iterate through the primitive elements. Starting with \( x^4 \) we have the value obtained from the generator polynomial \( g(x) = x^4 + x + 1 \). Given the equation \( x^4 + x + 1 = 0 \), we see that \( x^4 = -x - 1 \), but since \((-\alpha) = \alpha \) (\( \alpha \) is its own additive inverse) we can re-write as \( x^4 = x + 1 \). Following polynomials can be computed by multiplying the previous by \( x \), thus \( x^5 = x \cdot x^4 = x(x + 1) = x^2 + x \), and so on. The vectors so computed form the basis for the inverse log table.

<table>
<thead>
<tr>
<th>polynomial</th>
<th>vector</th>
<th>decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((0 0 0 0))</td>
<td>0</td>
</tr>
<tr>
<td>(x^0)</td>
<td>1 ((0 0 0 1))</td>
<td>1</td>
</tr>
<tr>
<td>(x^1)</td>
<td>(x) ((0 0 1 0))</td>
<td>2</td>
</tr>
<tr>
<td>(x^2)</td>
<td>(x^2) ((0 1 0 0))</td>
<td>4</td>
</tr>
<tr>
<td>(x^3)</td>
<td>(x^3) ((1 0 0 0))</td>
<td>8</td>
</tr>
<tr>
<td>(x^4)</td>
<td>(x + 1) ((0 0 1 1))</td>
<td>3</td>
</tr>
<tr>
<td>(x^5)</td>
<td>(x^2 + x) ((0 1 1 0))</td>
<td>6</td>
</tr>
<tr>
<td>(x^6)</td>
<td>(x^3 + x^2) ((1 1 0 0))</td>
<td>12</td>
</tr>
<tr>
<td>(x^7)</td>
<td>(x^3 + x + 1) ((1 0 1 1))</td>
<td>11</td>
</tr>
<tr>
<td>(x^8)</td>
<td>(x^2 + 1) ((0 1 0 1))</td>
<td>5</td>
</tr>
<tr>
<td>(x^9)</td>
<td>(x^3 + x) ((1 0 1 0))</td>
<td>10</td>
</tr>
<tr>
<td>(x^{10})</td>
<td>(x^2 + x + 1) ((0 1 1 1))</td>
<td>7</td>
</tr>
<tr>
<td>(x^{11})</td>
<td>(x^3 + x^2 + x) ((1 1 1 0))</td>
<td>14</td>
</tr>
<tr>
<td>(x^{12})</td>
<td>(x^3 + x^2 + x + 1) ((1 1 1 1))</td>
<td>15</td>
</tr>
<tr>
<td>(x^{13})</td>
<td>(x^3 + x^2 + 1) ((1 1 0 1))</td>
<td>13</td>
</tr>
<tr>
<td>(x^{14})</td>
<td>(x^3 + 1) ((1 0 0 1))</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1: Galois Field for \( GF(2^4) \).
4.2 Information Dispersal Matrix

To create the check-sums necessary for our Reed-Solomon encoding, we use a \((n + m) \times n\) information dispersal matrix, created by altering a standard Vandermonde matrix until the first \(n\) rows are the \((n \times n)\) identity matrix. Once we have a properly created dispersal matrix, any sub-matrix formed by deleting \(m\) rows of the matrix is invertible via Gaussian Elimination.

A Vandermonde matrix is defined as:

\[
\begin{bmatrix}
0^0 (= 1) & 0^1 (= 0) & 0^2 (= 0) & \cdots & 0^n (= 0) \\
1^0 & 1^1 & 1^2 & \cdots & 1^{n-1} \\
2^0 & 2^1 & 2^2 & \cdots & 2^{n-1} \\
3^0 & 3^1 & 3^2 & \cdots & 3^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
(n + m - 1)^0 & (n + m - 1)^1 & (n + m - 1)^2 & \cdots & (n + m - 1)^{n-1}
\end{bmatrix}
\]

The above Vandermonde matrix has the property that any sub-matrix formed by deleting \(m\) rows, is invertible. By using elementary transformations of the above matrix, we can maintain its rank, and thereby maintain its invertible property.\[PD03\] Thus we can derive our information dispersal matrix \(A\) through the following elementary operations on the Vandermonde matrix until the first \(n\) rows are the identity matrix.\[PD03\]

- Any column \(C_i\) may be swapped with column \(C_j\).
- Any column \(C_i\) may be replaced by \(C_i \cdot c\), where \(c \neq 0\).
- Any column \(C_i\) may be replaced by adding a multiple of another column to it: \(C_i = C_i + c \cdot C_j\), where \(j \neq i\) and \(c \neq 0\).

The algorithm for constructing our information matrix \(A\) is as follows \[PD03\] (note that all arithmetic is done in the Galois Field):

- Given index \(i < n\), we will transform row \(i\) into an identity row without altering the other identity rows. If the \(i\)-th element of row \(i\) is zero, find a column \(j\) such that \(j > i\) and the \(j\)-th element of row \(i\) is non-zero and swap columns \(i\) and \(j\). Since \(j > i\), swapping columns will not alter the first \(i - 1\) rows of the matrix. Additionally such a column is guaranteed to exist; otherwise the first \(n\) rows of the matrix would not compose an invertible matrix.
- Let \(f_{i,i}\) be the value of the \(i\)-th element of row \(i\) and let \(f_{i,i}^{-1}\) be its multiplicative inverse. Since \(f_{i,i} \cdot f_{i,i}^{-1} = 1\) and since \(f_{i,i} \neq 0\), \(f_{i,i}^{-1}\) is guaranteed to exist. If \(f_{i,i} \neq 1\), replace column \(C_i\) with \(f_{i,i}^{-1} \cdot C_i\).
- Now \(f_{i,i} = 1\). For all columns \(j \neq i\) and \(f_{i,j} \neq 0\), replace column \(C_j\) with \(C_j - f_{i,j} \cdot C_i\), where \(f_{i,j}\) is the \(j\)-th element in row \(i\).
- Repeat until the first \(n\) rows are identity rows.
Once completed we have our invertible dispersal matrix $A$ which we can invert using Gaussian Elimination.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>log[i]</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>10</td>
<td>3</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>13</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>invlog[i]</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>11</td>
<td>5</td>
<td>10</td>
<td>7</td>
<td>14</td>
<td>15</td>
<td>13</td>
<td>9</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Logarithm tables for $GF(2^4)$

An example [PD03], we construct $A$ for $n = 3$, $m = 3$ over $GF(2^4)$. We first construct the $6 \times 3$ Vandermonde matrix over $GF(2^4)$.

$$
\begin{bmatrix}
0^0 & 0^1 & 0^2 \\
1^0 & 1^1 & 1^2 \\
2^0 & 2^1 & 2^2 \\
3^0 & 3^1 & 3^2 \\
4^0 & 4^1 & 4^2 \\
5^0 & 5^1 & 5^2
\end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 5 \\ 1 & 4 & 3 \\ 1 & 5 & 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}
$$

Row 1 is already an identity row, so we move on to row 2. To convert row 2, we note that $f_{2,1} = f_{2,2} = f_{2,3} = 1$, so we need to replace $C_1$ with $(C_1 - C_2)$ and $C_3$ with $(C_3 - C_2)$. Resulting in (a). All that is left is to convert row 3. Since $f_{3,3} \neq 1$, we replace $C_3$ with $6^{-1}C_3 = 7C_3$ resulting in (b), then finally replace $C_1$ with $(C_1 - 3C_3)$ and $C_2$ with $(C_2 - 2C_3)$ to finally yield our desired $A$ (c).

$$
\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 2 & 6 \\ 2 & 3 & 6 \\ 5 & 4 & 7 \\ 4 & 5 & 7 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 2 & 1 \\ 2 & 3 & 1 \\ 5 & 4 & 6 \\ 4 & 5 & 6 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 15 & 8 & 6 \\ 14 & 9 & 6 \end{bmatrix}
$$

4.3 The Reed-Solomon Algorithm

Given a field of size $2^h$, words will be up to $h$ bits in size and up to $2^h - 1$ data words can be used in the system. Thus for $GF(2^8)$ we operate on words with values between 0 and 255 and can have up to 255 devices. For large arrays of disks beyond 255 devices, we would need to use a larger field, such as $GF(2^{16})$ which would let us handle up to 65535 disks while operating on words 2 bytes in size at a time.

The general idea of the algorithm is to generate a set of check-sums that are linearly independent from one another. Given the data set $D$ composed of data elements $\{d_1, d_2, \ldots, d_n\}$, we generate the check-sum word $c_i$ by applying a function $F_i$ to the data $D$, i.e. $c_i = F_i(d_1, d_2, \ldots, d_n)$.

In this algorithm, $F$ is the last $m$ rows of the invertible matrix formed from a Vandermonde
matrix as shown above. The first \( n \) rows are identity matrix, so data elements multiplied through them result in the appropriate data element for that row. The remaining \( m \) rows are the linearly independent rows used to generate the check-sum words from the data. Given our invertible matrix \( A = \begin{bmatrix} I \\ F \end{bmatrix} \), our data \( D \), we have the equation to generate an array \( E \) where \( E = \begin{bmatrix} D \\ C \end{bmatrix} \) as:

\[
A \cdot D = E
\]

Or as a matrix:

\[
\begin{bmatrix}
1 & 0 & \ldots & 0 \\
0 & 1 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & 1 \\
f_{1,1} & f_{1,2} & \ldots & f_{1,n} \\
\vdots & \vdots & \ddots & \vdots \\
f_{m,1} & f_{m,2} & \ldots & f_{m,n}
\end{bmatrix}
\begin{bmatrix}
d_1 \\
d_2 \\
d_3 \\
\vdots \\
c_1 \\
\vdots \\
c_m
\end{bmatrix}
= \begin{bmatrix}
d_1 \\
d_2 \\
d_3 \\
\vdots \\
c_1 \\
\vdots \\
c_m
\end{bmatrix}
\]

The check-sum words can naturally be computed directly from the last \( m \) rows of the \( A \) matrix. When any of the data words changes, then each check-sum word must be updated to reflect the change. This can be done efficiently by subtracting out the portion of the check-sum corresponding to the old data word and adding in the new amount, so in this way we do not need to re-inspect the other data words to affect the change. Given a new data word \( d'_j \) which changes \( d_j \), we compute:

\[
c'_i = c_i + f_{i,j}(d'_j - d_j),
\]

where \( c'_i \) is the new updated check-sum for each \( c_i \).[P96]

To perform data recovery, we use the identity matrix \( I \) and the matrix \( D \) and replace the rows in \( I \) and \( D \) where there are missing data words with the first available check-sum row from \( A \) and check-sum word from \( E \). For example, if data word \( d_2 \) was missing, we would replace row 2 of the identity matrix with the first check-sum row from \( A \) (row \( n + 1 \)) and replace the value for \( d_2 \) in \( D \) with \( c_1 \). For each successive missing data element we repeat this process using the next available check-sum row until we have replaced all the missing data elements or run out of check-sum rows (in which case we cannot recover the data.) Once we have done this we have the new matrices \( A' \) and \( E' \). To recover the data we solve the following equation to find \( D \):

\[
A' \cdot D = E'
\]

By inverting the matrix \( A' \) using Gaussian Elimination we yield:

\[
D = (A')^{-1} \cdot E'
\]
4.4 An RS example

An example for \( n = 3 \) and \( m = 3 \), \( D = \{8, 5, 10\} \) in the field \( GF(2^4) \). We first construct \( A \) by altering a Vandermonde matrix in the way described above to yield:

\[
A = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & 1 & 1 \\
15 & 8 & 6 \\
14 & 9 & 6
\end{bmatrix}
\]

We use rows 4-6 to calculate our check-sum digits \( c_1, c_2 \) and \( c_3 \) as follows:

\[
c_1 = (1 \cdot 8) \oplus (1 \cdot 5) \oplus (1 \cdot 10) \\
= 8 \oplus 5 \oplus 10 \\
= (1000) \oplus (0101) \oplus (1010) = (0111) = 7
\]

\[
c_2 = (15 \cdot 8) \oplus (8 \cdot 5) \oplus (6 \cdot 10) \\
= 1 \oplus 14 \oplus 9 \\
= (0001) \oplus (1110) \oplus (1001) = (0110) = 6
\]

\[
c_3 = (14 \cdot 8) \oplus (9 \cdot 5) \oplus (6 \cdot 10) \\
= 9 \oplus 11 \oplus 9 \\
= (1001) \oplus (1011) \oplus (1001) = (1011) = 11
\]

Now suppose we lose the data elements \( d_1, d_3 \) and check-sum word \( c_2 \). We can still recover our data words (and once we have, we can recompute our missing check-sum word). To do so we take our identity matrix and replace the rows 1 and 3 corresponding to data words \( d_1 \) and \( d_3 \) with the two remaining check-sum rows, and then invert it, like so:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix} \Rightarrow A' = \begin{bmatrix}
1 & 1 & 1 \\
0 & 1 & 0 \\
14 & 9 & 6
\end{bmatrix} \Rightarrow (A')^{-1} = \begin{bmatrix}
4 & 10 & 15 \\
0 & 1 & 0 \\
5 & 11 & 15
\end{bmatrix}, E' = \begin{bmatrix}
7 \\
5 \\
11
\end{bmatrix}
\]

Then to find \( d_1 \) and \( d_3 \) we multiply rows 1 and 3 of \( (A')^{-1} \) through matrix \( E' \) to yield:
\[ d_1 = (4 \cdot 7) \oplus (10 \cdot 5) \oplus (15 \cdot 11) \\
= 15 \oplus 4 \oplus 3 \\
= (1111) \oplus (0100) \oplus (0011) = (1000) = 8 \]

\[ d_3 = (5 \cdot 7) \oplus (11 \cdot 5) \oplus (15 \cdot 11) \\
= 8 \oplus 1 \oplus 3 \\
= (1000) \oplus (0001) \oplus (0011) = (1010) = 10 \]

Having obtained our data words, we can re-calculate the missing check-sum \( c_2 \) through the normal mechanism.

### 4.5 Complexity

Given \( n \) data elements, and \( m \) desired checksum words, the complexity of the algorithm is thus:

First addition and subtraction in the field is an xor operation, which is assumed to be \( O(1) \) in complexity – each bit pair in the operation is independent of all other bit-pairs, and so can be done efficiently in parallel. Multiplication and division are two table look-ups, a regular addition or subtraction with a possible modulus operation (which can be implemented as a comparison and optional addition/subtraction) followed by an additional table look-up, so three table look-ups and up to two addition/subtractions, so at most \( (3T \cdot 2h) \) operations = \( \Omega(h) \), where \( h \) is the number of bits input in the addition/subtraction and \( T \) is some table look-up constant.

The generation of the distribution matrix, requires first creating a Vandermonde matrix. The first and second column of which require no expensive operations to create. Each successive column requires a multiplication in the field of the previous column value by the row index for each row > 2. So given a \((n + m) \times n\) matrix, we require \((n + m - 1) \times (n - 2)\) field multiplications. Conversion to invertible form, requires up to \(2 \cdot (n + m - 1) \cdot n\) additional multiplications and xors. Since this matrix can be cached, and is only generated once, its computational complexity is mostly irrelevant.

To compute a single checksum word requires \( n \) field multiplications and \((n - 1)\) additions (xors) in the field. An update will require \( m \) check-sums to be updated, which requires two xors and a multiplication for each checksum.

To recover data requires a matrix inversion which is done by Gaussian Elimination, which has an arithmetic complexity of approximately \(2n^3/3\) operations or \(O(n^3)\). This operation needs to only be performed once prior to data restoration, so might be considered insignificant.

Recovery of up to \( m \) data words again only requires \( n \) field multiplications and \((n - 1)\) xor operations for each recovered data word.

Overall time complexity for Reed-Solomon grows with the square of the growth of data elements and check symbols in the system, or \(O(n \cdot m)\), \(m \leq n\), or \(O(n^2)\).
5 Low Density Parity Check

**Low-Density Parity Check (LDPC):** Used in high bandwidth applications, such as satellite transmissions and 10GigE, LDPC is capable of reconstructing a message very close to the noise threshold limit, as such it would seem to make a good candidate for both an error correction and erasure code.

**References**

[H96] Raymond Hill
*A First Course in Coding Theory*, From
Oxford University Press, New York, 1996

[vL99] J.H. van Lint
*Introduction to Coding Theory*, From
Springer-Verlag, Berlin/Heidelberg/New York, 1999

[P96] James S. Plank
*A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems*, From
University of Tennessee, 1996, CS-96-332,

[PD03] James S. Plank, Ying Ding,
*Note: Correction to the 1997 Tutorial on Reed-Solomon Coding*, From
University of Tennessee, 2003, CS-03-504,
http://www.cs.utk.edu/plank/plank/papers/CS-03-504.html

[A04] H. Peter Anvin (hpa@zytor.com),
The Mathematics of RAID-6, From
kernel.org, 2004,

[RS] Various Authors, *Reed-Solomon Error Correction*, From
http://en.wikipedia.org/wiki/Reed-Solomon_error_correction

[IIC] Various Authors, *Hamming Code*, From
http://en.wikipedia.org/wiki/Hamming_code