RAID-6 Based Distributed Storage System

Meng Shen, Ruihang Wang, Yihang Li

***Abstract*—RAID-6 is an extension of RAID-5 by adding another parity block. It is designed to tolerate any two concurrent disk failure and maintain a relatively high storage efficiency. The mathematics behind RAID-6 is Galois Field theory and Reed-Solomon Coding. In this project, we create our own finite field operation library and use it to develop a RAID-6 based distributed storage system in Python 3.7 under Linux envrionment. In addition to basic functionalities, we also developed several advanced features in the system that can accommodate real files of arbitrary size and support user defined configurations. To make it available for implementation, we open-source our codes with read-me documents at https://github.com/GuluDeemo/CE7490-RAID6**

***Index Terms*—RAID-6, Galois Field, Reed-Solomon codes, Python.**

I. Introduction

THE concept of RAID (Redundant Array of Independent Disks) was raised in the paper ”A Case for Redundant Arrays of Inexpensive Disks (RAID)” for the demand for rising the reliability, capacity and speed of storage systems. Instead of reading and writing data in one disk drive, RAID achieves larger width of input/output, and hence improves the performance significantly. Many levels of RAID have been raised since 1970s and are widely used in different data protection requirements. Most RAID implement error-detection drives and employ error protection schemes called ”parity” to provide fault tolerance in one or more disks. Most use simple XOR, but RAID-6 uses two separate parities based respectively on addition and multiplication in a Galois Field and Reed-Solomon error detection. This report focused on RAID-6.

Among all levels of RAID, RAID-5 and RAID-10 are most widely used by industry manufacturers. RAID-10 combines RAID-1 and RAID-0 by dividing data into several parts and mirroring each part into pairs. RAID-10 is designed to tolerate multiple disk failure except two corresponding disks in the mirroring pairs. However, the storage efficiency is at only 50%, which is very low. RAID-5 consists of block-level striping with distributed parity, and the parity information is saved among drives. When facing failure, the loss data can be calculated and rebuilt by distributed parity. However, RAID-5 can only tolerate one disk failure. RAID-6 is designed to tolerate any two concurrent disk failure and maintain a relatively high storage efficiency.

Meng shen and Ruihang Wang are with the School of Computer Science Engineering, Nanyang Technological University, Singapore, e-mail: (meng005@e.ntu.edu.sg, ruihang001@e.ntu.edu.sg).

Yihang Li is with the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore, e-mail: (leah9704@gmail.com).

Project submitted Nov 23, 2019.

RAID-6 is a private raid-level standard proposed concurrently by several large enterprises. This RAID level is developed on the basis of RAID-5, while unlike RAID-5, there is not only a parity area for sibling data on the drive, but also a parity area for each data block. Two independent parity systems use different algorithms and the data is very reliable. Even if two disks fail at the same time, it does not affect the use of the data, which enhances the disk's fault tolerance. With a RAID-6 based distributed storage system, it is possible to mitigate most of the problems associated with RAID-5.

RAID-6 achieves faster reading performance and higher fault tolerance. All these advantages make larger RAID groups more practical, especially for high-availability systems, as larger-capacity drives take longer to restore. The larger the drive capacities and the larger the array size, the more important it becomes to choose RAID-6 instead of RAID-5. However, RAID-6 is not as popular as other levels of RAID for practical application because of its complicated system and expensive RAID controller. For most small businesses which do not require high security level of data preservation, it is better to use RAID-5 economically. While for businesses demand higher security level of data preservation like data center, it is necessary to implement RAID-6.

In this report, we present a reliable distributed storage system based on RAID-6 under Python 3.7 environment. The mathematical operation in Galois Field is well investigated and created as a library to support RAID-6 controller through practical implementation. The RAID-6 based system support several basic functionalities, such as distributed data storage, failure detection and lost redundancy recovery. On top of the minimal implementation, we also investigate ... The rest of the report is organized as follows: ...

II. Overview

The RAID-6 distributed storage system is developed in Python 3.7 under Linux environment. Instead of using existing finite field libraries, we create our own Galois Field **GF**(2^8) operations based on the tutorial document of Reed-Solomon Coding in [**?**]. The only dependency we use in this project is *NumPy* for basic matrix operations. The functionalities we have developed for this RAID-6 system are summarized as follows:

- Store and access abstract “data objects” across storage nodes using RAID-6 for fault-tolerance.
- Include mechanisms to determine failure of storage nodes.
- Carry out rebuild of lost redundancy at a replacement storage node.
- Accommodate real files of arbitrary size, taking into account issues like RAID mapping, etc.

- Support mutable files, taking into account update of the content, and consistency issues.
- Support larger set of configurations

III. Problem Specification

The RAID-6 theory is based on the Galois Field mathematics and the Reed-Solomon codes. This section will provide relevant definations and principles in the implementation of a RAID-6 distributed storage system. First, let $D_1, D_2, ..., D_n$ denote n *storage disks* and $C_1, C_2, ..., C_m$ denote m *checksum disks*. Each of this disk holds the same storage capacity. The objective is to define the calculation of C_i such that if any m of $D_1, D_2, ..., D_n, C_1, C_2, ..., C_m$ corrupt. The contents of the corrupted devices can be rebuilt from the non-corrupted devices.

A. Arithmetic over Galois Field

Fields with 2^w elements under closed operations like addition and multiplication are called *Galois Fields* (denoted as **GF**(2^w)). The elements are integers from zero to $2^w - 1$. For example, the field **GF(2)** can be represent as the set $0, 1$. A small field would limit the number of disks possible while a large field would require extremely large tables. For RAID-6, we choose the commonly used field of **GF**(2^8), which allows for a maximum of 255 data disks. The operations in this finite field are illustrated as below:

Addition: The addition field operator ($\oplus$) is performed by bitwise XOR.

Subtraction: The addition and subtraction ($\ominus$) are the same operation. For example, $A \oplus B = A \ominus B$

Multiplication: The multiplication ($\otimes$) is performed by bitwise AND, and can be simplified using logarithms table. The details of setting up this table will be presented later.

Division: Division is defined as multiplication with an inverse like $A/B = A \otimes B^{-1}$

Based on the definitions of the foundamental operations, the following basic rules should be obeyed in this field:

- Addition is commutative: $A \oplus B = B \oplus A$
- Addition is associative: $(A \oplus B) \oplus C = A \oplus (B \oplus C)$
- Multiplication is commutative: $A \otimes B = B \otimes A$
- Multiplication is associative: $(A \otimes B) \otimes C = A \otimes (B \otimes C)$
- Distributive law: $(A \oplus B) \otimes C = A \otimes C \oplus B \otimes C$

Here, we further explain how to perform multiplication and division in **GF**(2^8). The multiplication and division in this field is much more difficult and harder to implement. Multiplying two field elements is to multiply their corresponding polynominals. To simplify this calculation, we use two look up logarithm tables with length of each equals $2^w - 1$. These tables are defined as *gflog* and *gfilog* and can be generated using Algorithm 1.

- *gflog[]*: This table is defined for the indices 1 to 255, which maps the index to its logarithm in the Galois Field.
- *gfilog[]*: This table is defined for the indices 0 to 254, which maps the index to its inverse logarithm in the Galois Field.

Obviously, *gflog[gfilog[i]] = i*, and *gfilog[gflog[i]] =i*. With these two tables, we can easily multiply two non-zero elements of **GF**(2^8) by adding their logs and then taking the inverse log as given by:

$$A \otimes B = gfilog[gflog[A] + gflog[B]] \tag{1}$$

$$A \otimes B^{-1} = gfilog[gflog[A] - gflog[B]] \tag{2}$$

When w is small, these two tables can accelerate multiplication and division in the Galois Field.

Algorithm 1: Setup logarithm tables

```
Input: $w = 8$, $modulus = 0b100011101$
Output: two logarithm tables
1 max = $1 << 8$
2 $b = 1$
3 gflog, gfilog = malloc(max)
4 for $log = 0$ to $max$ do
5     gflog[$b$] = $log$
6     gfillog[$log$] = $b$
7     $b$ = $b << 1$
8     if $b$ & $max$ then
9         $b$ = $b$ ^ $modulus$
```

B. Reed-Solomen Coding

By using Reed-Solomen Coding, we could use more parities to back up our data, not just 2. It could be very helpful when one wants to achieve better data security.

1) Calculating Parities: Suppose we have N data disks and M parity disks. To calculate the parity P_i, we define function F_i:

$$P_i = F_i(d_1, d_2, ..., d_n) = \sum_{j=1}^{n} d_j f_{i,j} \tag{3}$$

If we treat data and parity as vectors, where $D = [d_1, d_2, ..., d_i]$ and $P = [p_1, p_2, ..., p_i]$, we could get a simpler equation:

$$P = FD \tag{4}$$

To calculate P, first, we need to generate the $M \times N$ Vandermonde matrix F where $f_{i,j} = j^{i-1}$

$$F = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & N \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2^{M-1} & \cdots & N^{M-1} \end{bmatrix} \tag{5}$$

In this way, we could easily get our parities P by applying a matrix multiplication on Galois Field.

2) Recovering Data: To recover our data from disk faliure, we need to define matrix $A = \begin{bmatrix} I \\ F \end{bmatrix}$ and $E = \begin{bmatrix} D \\ P \end{bmatrix}$, where I is a $N \times N$ identity matrix and matrix F, D, P are defined before. So we get the equation:

$$AD = E \tag{6}$$

When a disk or disks encount failure, we could delete the corresponding row in A and E. For instance, we have one failed data disk D_i and one failed parity disk P_j, then we need to delete i_{th} row in I and j_{th} row in F to get a new A'. Likewise, we also need to delete i_{th} row in D and j_{th} rwow in P to get a new E'. Then the equation becomes:

$$A'D = E' \tag{7}$$

What we want is actually D, so the problem becomes to solve the following equation:

$$D = A'^{-1}E' \tag{8}$$

The way we use to calculate the inverse matrix A'^{-1} is Gaussian Elimination, the only difference is we do it in Galois Field. Once we calculate our data D, then by using the method mentioned before: $P = FD$, we could update our parity P if it's necessary.

IV. Implementation

A. System Configuration

The configuration of the RAID-6 based distributed storage system is shown below

- Number of Disk: N = 6
- Number of Data Disk: D = 4
- Number of Checksum Disk: C = 2
- D_1 D_4 denote 4 data disks
- C_1 C_4 denote 2 data disks

V. Conclusion

The conclusion goes here.

Appendix A
Proof of the First Zonklar Equation

Appendix one text goes here.

Appendix B

Appendix two text goes here.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.