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 failures 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 environment. 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 paper "A Case for Redundant Arrays of Inexpensive Disks (RAID)" [1] for the demand of rising the reliability, capacity and speed of storage systems cost-efficiently. Instead of reading and writing data in one large disk drive, RAID organizes a series of inexpensive independent disk drives with certain patterns and offer a virtually unified disk drive. The idea of RAID achieved great performance with large capacity, wide width of input/output and low cost and hence gained popularity. Since 1970s, many versions of RAID have been raised according to requirements of different application scenarios and are widely used in various data protection applications. Most RAID versions implement error-detection drives and employ error-protection schemes called "parity" to provide fault tolerance ability in one or more disks. The most typical method is to generate a parity disk by simple XOR function [4], where failure of up to one disk can be tolerated by reconstructing the failed one from the rest of disk array. Other methods are designed for multiple failure tolerance, including Galois Field and Reed-Solomon error detection [3] [5], which is used in RAID-6. This report will give an introduction of RAID family and then focus on the methodology and implementation of RAID-6.

Among all levels of RAID, RAID-5 and RAID-10 are most widely used by industry manufacturers. RAID-10 combines RAID-1 [7]and RAID-0 [6] 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 only 50% and can tolerant, 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.

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 supports several basic functionalities, such as distributed data storage, failure detection and lost redundancy recovery. On top of the minimal implementation, we also developed several advanced features that can accommodate real files of arbitrary size and support various user-defined configurations for practical applications. The rest of the report is organized as follows: Section II presents an overview of our project. Section III investigates the mathematics and principle of RAID-6 system. Section IV introduces the implementation details of our designed system. Section V presents the experiment results of different configurations. The implementation of this project is concluded in Section VI.

II. Overview

The RAID-6 distributed storage system is developed in Python 3.7 under Linux environment. Instead of using existing

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 on Nov 23, 2019.

finite field libraries, we create our own Galois Field **GF**(2^8) operations based on the tutorial document of Reed-Solomon Coding in [2], [3]. 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

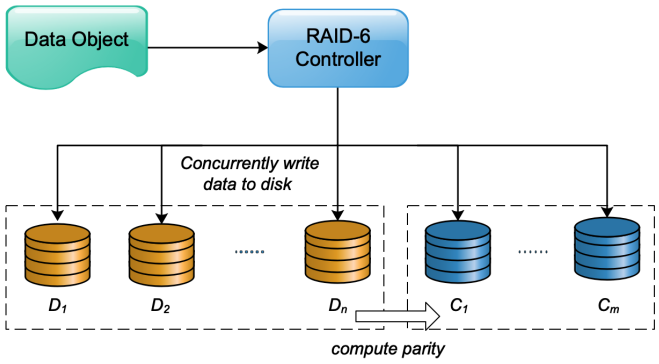

Fig. 1: RAID-6 distributed storage system

III. Problem Specification

The RAID-6 theory is based on the Galois Field mathematics and the Reed-Solomon codes. This section will provide relevant definitions and principles in the implementation of a RAID-6 distributed storage system. The framework of a RAID-6 system is shown in Figure 1. 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 represented 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 operations. For example, $A \oplus B = A \ominus B$

Multiplication: The multiplication ($\otimes$) is performed by bitwise AND, and can be simplified using two logarithms tables. 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 fundamental 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 are much more difficult and harder to implement. Multiplying two field elements is to multiply their corresponding polynomials. To simplify this calculation, we use two lookup 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.

Algorithm 1: Setup logarithm tables

Input: $w = 8$, $modulus = 0b100011101$
Output: two logarithm tables

```
max = 1 << w
b = 1
gflog, gfilog = malloc(max)
for log = 0 to max do
    gflog[b] = log
    gfillog[log] = b
    b = b << 1
    if b & max then
        b = b ^ modulus
```

- *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.

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 failure, 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 encounter 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} row 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 Disks: N = 6
- Number of Data Disks: D = 4
- Number of Parity Disks: C = 2
- $D_1 \sim D_6$ denote 6 disks
- Chunk Size: 16 Bytes

Furthermore, one could change the configuration such as number of parity disks or chunk size to extend RAID6 system. On chunk size, when there are mainly large files to be stored, it is recommended to use a larger chunk size.

To fully distribute our data in all disks, we will shift our data chunk and parity chunk in one direction. Each stripe of data includes D(which is 4 in our configuration) raw data chunk: d_{ij} where i is chunk index and j is stripe index, and C(which is 2) parity chunk: p_{ij}. And the order of data chunk and parity chunk will shift, which could average the IO pressure on all disks. If we do not shift the store location and assign some disks to only store parity data, they may be a big bottleneck if their IO speed is much slower than others.

	D_1	D_2	D_3	D_4	D_5	D_6
S_1	d_{11}	d_{21}	d_{31}	d_{41}	p_{11}	p_{21}
S_2	p_{22}	d_{12}	d_{22}	d_{32}	d_{42}	p_{12}
S_3	p_{13}	p_{23}	d_{13}	d_{23}	d_{33}	d_{43}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

TABLE I: RAID6 data distribution

B. Basic Functions

We first implemented some basic functions: data segmentation, writing, reading and recovering. With these functions, we could store a file in an abstract RAID6 system, and then read or recover it.

On data segmentation, every time we will read 4 chunks of data whose size should be $Chunksize \times NumberOfDataDisks = 16 \times 4 = 64Bytes$ from file. And then by applying Reed-Solomen algorithm, we could calculate the corresponding parity data chunks to compose a completed stripe.

On writing, we need to shift it according to its stripe number. After successfully writing into RAID6, the total number of stripes and file information like file name and format should be recorded as well.

On reading and recovering, the most important part is to correctly index each chunk to decide its order in stripe. While reading, to make it faster, we could skip the parity data chunks if we do not need to check the completeness of file. While recovering, we could only compute the missing parts rather than computing all chunks.

C. Other Functions

In addition to basic functionalities, we also developed several advanced features in our RAID-6 distributed storage system. The advanced features we have implemented are summarized as fellows.

1) Accommodate Real Files of Arbitrary Size: Previously, the data object we used for test is a synthetically generated string in .txt file. To make it available for practical storage, we have tried different data in other format. Zero padding method is included to ensure that files in different are equally distributed with respect to the chunk size of each disk. The padding method added a series of bit 0 at the end of the files and can be clipped when rebuilding data from non-corrupted disks.

2) Support User-Defined Configurations : In the test phase, the RAID-6 system is specifically configured with 6 data disks and 2 checksum disks as the conventional example. To make it more flexible, we separately take the configuration file as a Python class to let users define their preferred choice for practical implementation. The experiments of different configurations with different performance is discussed in Section V .

V. Experiments

In this experiment, we change the chunk size to **4 Bytes**, **8 Bytes**, ..., and **256 Bytes**. The test file we use is a jpg with size of 805 KB. We measure the reading, writing and recovering time in the configuration of 4 data disks and 2 parity disks. Following figures are the experiment result:

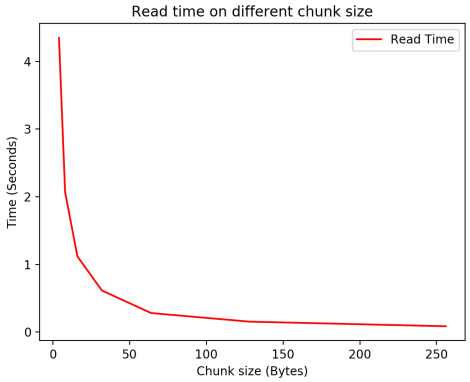

Fig. 2: Read time on different chunk size, file size is 802 KB

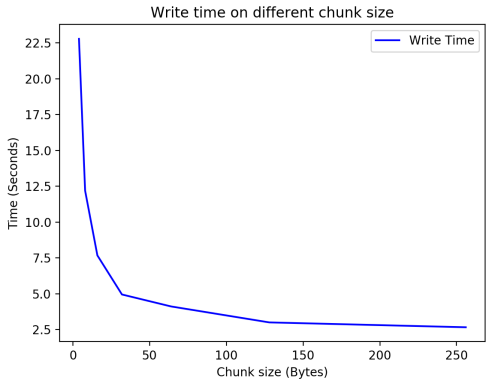

Fig. 3: Write time on different chunk size, file size is 802 KB

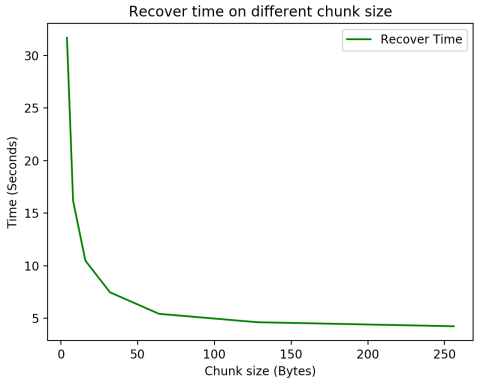

Fig. 4: Recover time on different chunk size, file size is 802 KB

In the second experiment, we use the txt file with size of 512 Bytes, and measure the reading, writing and recovering time.

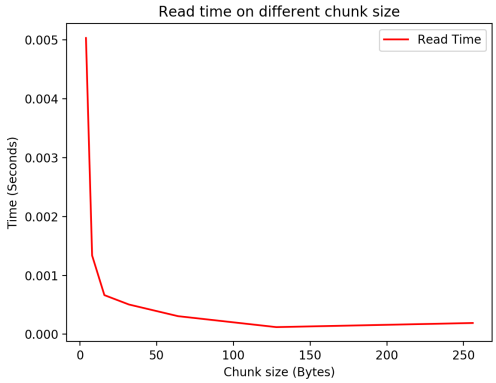

Fig. 5: Read time on different chunk size, file size is 512 Bytes

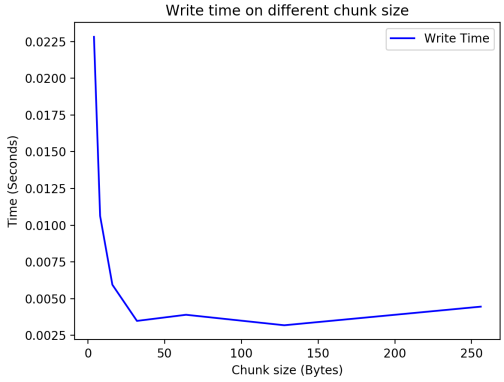

Fig. 6: Write time on different chunk size, file size is 512 Bytes

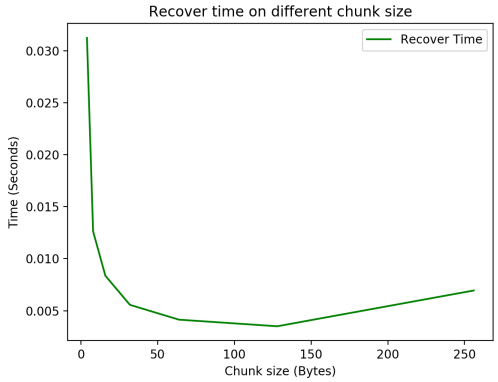

Fig. 7: Recover time on different chunk size, file size is 512 Bytes

From the results, we can see that with the chunk size becomes larger, the reading, writing and recovering time all decrease. This is because data is divided into larger pieces, which reduces the times of reading or writing the file(IO). However, in second experiment, when the chunk size is too large compared with the file size, the performance might be worse. So the chunk size should depend on how large you files mainly are. If you store many tiny files, smaller chunk size could save space because of less padding. On the contrary, larger chunk size should be choosed if there are many large files because of less IO time.

VI. Conclusion

We have successfully developed a RAID-6 based distributed storage system in Python 3.7 under Linux environment. Instead of using existing finite field libraries, we create our class for basic operations in Galois Field. Based on the mathematics of Reed-Solomen Coding, the RAID-6 system is able to support basic functionalities such as distributed data storage, failure detection and lost redundancy recovery. On top of the minimal implementation, we also develop several advanced features to make the system more flexible and scalable. To make it available for implementation, we open-source our codes at GitHub for further test and discussion.

Acknowledgment

We would like to thank Prof. Anwitaman Datta for teaching the course CE7490 Advanced Topics in Distributed Systems and the abundant resources provided in his lecture.

References

[1] Patterson, David A. and Gibson, Garth and Katz, Randy H., *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, 3rd ed.

[2] H. Peter Anvin, *The mathematics of RAID-6*, 1st ed, 20, Dec, 2011.

[3] J. S. Plank, *A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems*, University of Tennessee, July, 1996. http://web.eecs.utk.edu/~jplank/plank/papers/CS-96-332.html

[4] M. Blaum, J. Brady, J. Bruck, and J. Menon.*EVENODD: An optimal scheme for tolerating double disk failure in RAID architectures.* IEEE Transactions on Computers, 44(2):192-202, 1995

[5] L. Xu, and J. Bruck. *X-Code: MDS array codes with optimal encoding.* IEEE Transactions on Information Theory, 45(1):272-276, 1999.

[6] BITTON, D. AND GRAY, J. 1988. *Disk shadowing.*In Very Large Database Conference XIV. Morgan Kaufmann, San Mateo, Calif., 33 1–338.

[7] CHEN, P. M., GIBSON, G., KATZ, R., AND PATTERSON, D. A. 1990. *An evaluation of redundant arrays of disks using an Amdahl 5890.* In Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling ofComputer Systems. ACM, New York. The first experimental evaluation of RAID.

[8] MENON, J. and CORTNEY, J. 1993. *The architecture of a fault-tolerant cached RAID controller* In Proceedings of the 20th International Svmposum on Computer Architecture IEEE, New York, 76–86.

[9] Lm, E. K. AND KATZ, R, H, 1991.*Performance consequences of parity placement m disk arrays* In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operatzng Systems(ASPLOS-ZV). IEEE, New Yorkj 190-199