RAID-6 Based Distributed Storage System

Aye Phyu Phyu Aung, Seima Saki Suriyasekaran, Nick Zhang Shihui

***Abstract*—With the development of information technology, data availability and integrity have become increasingly important so as how to safely store the information without losing it due to unpredictable events such as corruptions. Redundant Array of Independent Disks (RAID) is a data storage virtualization technology that combines multiple physical drives into one logical storage unit thereby increasing performance and reliability. By using RAID, users can tolerate a certain amount of the drive failure. Among all the RAID implementations, RAID6 is one of the most commonly used architectures in the industry due to its ability to tolerate up to 2 disks failures. In this project of the course CE7490, we aim to implement and optimize RAID6 to store the data files across virtual drives in a distributed manner.**

I. Introduction

RAID (Redundant Array of Inexpensive Disks) distributes data blocks among multiple storage disks to achieve high performance and reliability. They use one or more error correction drives to recover the data in case of a failure. Data is distributed across the disks in several ways which are identified by different RAID levels [1]. RAID 0 consists of striping but no mirroring and parity, RAID 1 consists of data mirroring, RAID10 creates a striped set from a series of mirrored drives. RAID2, RAID3, RAID4 and RAID5 consists of a single parity block generated in different ways in each of these levels. These systems tolerate only single failure and the probability of more disks failing at the same time is very high when the capacity of the disks increase. To address this issue RAID6 [2] was implemented to provide better data protection. RAID6 consists of striping and two parity blocks, providing fault tolerance unto two failed disks.

RAID6 technology is based on Maximum Distance Separable(MDS) [3] coding as well as Galois Field(GF) or Finite field mathematics [4] to encode data on drives and recover it later when a failure occurs. MDS codes and GF are discussed in detail in Section II.

This project is to study and implement RAID6 technology and we have accomplished the following tasks:

1) Implement the Galois Field function
2) Implement RAID6
    - *Write* Operation (text, pictures (jpg, png))
    - *Read* Operation
    - Recovery up to two disks with parity P and Q after detecting corrupted disks while read operation
    - 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 (than just 6+2)
3) Experiments and analysis on RAID6 functionalities

The rest of the report is organized as follows. Section II describes the system architecture of RAID6 including MDS Codes, Galois Field. Section III describes the implementation steps of RAID6, read, write, update, data recovery and optimization, followed by experiments and analysis of results in Section IV. Finally, we conclude the report in Section V.

II. System Architecture

A. MDS Codes

Reed-Solomon codes [5] are one type of MDS codes which we have used in our implementation of this project. The Reed-Solomon Encoding method creates a Codeword of size n symbols totally consisting of k data symbols and $2t = n - k$ parity symbols calculated from the data symbols.

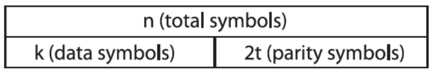


Fig. 1: Reed-Solomon Code $RS(n, k)$

A Reed-Solomon code referred as $RS(n, k)$ as shown in Figure 1. Each symbol consists of s-bits. The MDS codes used in RAID6 uses P+Q parities and is related with the Reed-Solomon codes when $2t = 2$ so the system is able to handle two erasures. For a symbol size s the maximum allowed code word length is $n = 2^s - 1$. A P+Q RAID6 implementation forms a codeword by taking a single byte from each of k data disks and calculating two parities P and Q and storing it in the P and Q disks respectively. A sample implementation of RAID6 with 5 disks totally is shown in Figure 2. Here $n = 5, k = 3$ with two parities P and Q.

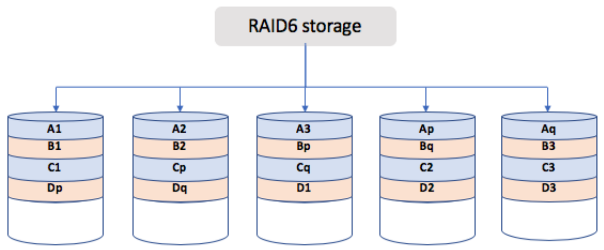


Fig. 2: RAID6 Example

The concept of two parity and using it for data recovery is similar to solving two linear equations for the unknown values, hence supporting the recovery of a maximum of two unknown values. But solving the linear equations which consist of integers is inconvenient as the range of the integers in them can be high. To solve this issue *Galois Field(GF)*, a type of finite field arithmetic, is used where we choose a prime number

n and change the meaning of arithmetic operations such as addition, subtraction, multiplication and division such that the resulting values wrap around n.

B. Galois Field

A GF is a set of values that contains a finite number of elements and we are interested here in a particular GF of size 2^8 and have elements from 0 to $2^8 - 1$. As mentioned in **Section II** the arithmetic operations are different in GF. The GF elements are enumerated as shown in [2] and the software implementation generates two lookup tables named *gflog and gfilog*. The arithmetic operations are summarized below:

- Addition and Subtraction are XOR operations denoted as $(\oplus)$.
- Multiplication denoted as $(\cdot)$ of two elements a and b follow the equation:

$$a \cdot b = gfilog[gflog[a] + gflog[b]] \tag{1}$$

- Division denoted as $(\div)$follows the equation

$$a \div b = gfilog[gflog[a]{-}gflog[b]] \tag{2}$$

C. Parity Generation in RAID6

The first parity block, named P block, is calculated by the following formula.

$$\mathbf{P} = \mathbf{D_0} \oplus \mathbf{D_1} \oplus \mathbf{D_2} \oplus \ldots \oplus \mathbf{D_{n-1}} \tag{3}$$

The second parity block, named Q block, requires much more computation. It is defined as the following formula.

$$\mathbf{Q} = g^0 \cdot \mathbf{D_0} \oplus g^1 \cdot \mathbf{D_1} \oplus g^2 \cdot \mathbf{D_2} \oplus \ldots \oplus g^{n-1} \cdot \mathbf{D_{n-1}} \tag{4}$$

Here $(\cdot)$ is the multiplication operation defined in Galois Field $\mathbf{GF}(2^8)$ with $\mathrm{m} = 100011101$ (corresponding to the irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$) as the modulus. g is the generator and g^i is defined as $\underbrace{g \cdot g \ldots g}_{i}$.

Handling One Disk Failures: For data disk or P disk, we calculate the XOR results of other disks excluding Q using Equation3. For Q disk, we recompute using Equation (4).
Handling Two Disk Failures: To recover one data disk and Q disk is easy, we first compute the XOR parity using Equation 3 to get the data; then recompute Q with Equation 4 accordingly.

If one data disk $\mathbf{D}_x$ and P parity are corrupted, we use the Equation 5 to retrieve the data.

$$\mathbf{D}_x = (\mathbf{Q}_x + \mathbf{Q}) \cdot g^{-x} \tag{5}$$

Where Q_x is computed as if $\mathbf{D_x} = \{00\}$ and for $\mathbf{GF}(2^8)$ we have $g^{-x} = g^{255-x}$. Afterwards, we can get P data using Equation 3.

If we lose 2 data block, namely $\mathbf{D}_x$ and $\mathbf{D}_y$, we use the following to recover them.

$$A = g^{y-x} \cdot (g^{y-x} + \{01\})^{-1} \tag{6}$$

$$B = g^{-x} \cdot (g^{y-x} + \{01\})^{-1} \tag{7}$$

$$\mathbf{D}_x = A \cdot (\mathbf{P} + \mathbf{P}_{xy}) + B \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) \tag{8}$$

$$\mathbf{D}_y = (\mathbf{P} + \mathbf{P}_{xy}) + \mathbf{D}_x \tag{9}$$

Here $\mathbf{P}_{xy}$ and $\mathbf{Q}_{xy}$ are computed parity as if $\mathbf{D}_x = \{00\}$ and $\mathbf{D}_y = \{00\}$.

III. Implementation

In this section, we will discuss our implementation of RAID6. We implement our project in Python-3.7. Our implementation is available at our GitHub Repository.

We implemented RAID6 by using folders in a file system to simulate the independent disks. During the initialization, we define the number of disks N_d and the chunk size to accommodate the real files of arbitrary size. The disks include $N_d - 2$ data disks, 1 P parity disk and 1 Q parity disk. Here, we allow $N_d \geq 8$.

We create a temporary file on the disk when we write a file and store the information of each written file with starting index, starting disk, offset and length of data against the file name.

A. Write Operation

The *write* operation firstly divides the data according to the $CHUNK\_SIZE$ and store the data to the N_d disks along with the calculated P and Q parity. For the incomplete chunks, we fill in trialling zeros for the correct parity calculation.

B. Read Operation

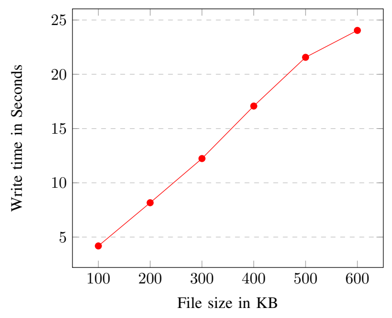
In each *read* operation, we specify the starting index of the data, starting disks, offset and the length of the data which we retrieve from $FILE\_INFO$. After that, we try to read one chunk of data. The chunk data are appended and write to the output unless a disk fails where we start the recovering process.

C. Data Recovery

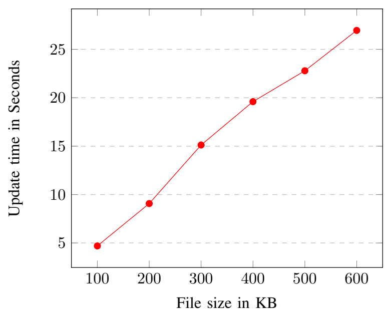
We recover up to 2 corrupted disks given the corrupted disk number is known. The data recovery covers the following scenarios:

1) One Disk Corruption
   - Recover data with parity P and Q
   - Recover the corrupted parity P
   - Recover the corrupted parity Q
2) Two Disks Corruption
   - Recovery of 2 chunks of data with parity P and Q
   - Recovery of 1 chunk of data and parity P
   - Recovery of 1 chunk of data and parity Q
   - Recovery of parity P and parity Q

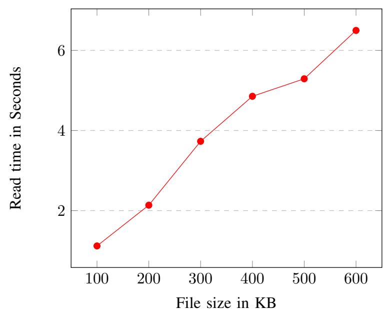
During the *read* operation, the corrupted disks are found out such that on failure to open a file chunk, we append $(chunk\_index + current\_disk) \mod N_d$ as the failed disk. Then, we start the auto-recovery process by passing the index(s) of the corrupted disk(s). After recovery, it continues



(a) Write operation



(b) Update operation



(c) Read operation

Fig. 3: Time taken for Write, Update, Read operations with 8 disks and chunk size of 128 bytes

the *read* operation of the stored data to output the data on the console or output file.

On one disk corruption, we recover the data and parity P by the formula Eq. (3). In case of parity Q, we recover by the formula Eq.(4).

On two disks corruption, we use the formulas Eq.(4)- Eq.(7) for recovering 2 chunks of data, Eq. 5 for recovering a chunk of data followed by Eq. 3 for parity P, Eq. 3 and Eq. 4 for a chunk of data and parity Q and re-calculating the parity bits if both P parity and Q parity are corrupted.

D. Support of mutable files

We can support the update of files to be written under the same file name. We compare the changed file with the original data copy stored, excluding the P and Q values. Once the different data is found, we replace it using the changed value and re-calculate the P and Q values. In the case of larger files are updated, we check if the following chunk is already written. If already written, the updated data is written is two places i.e,

$$\{'starting\_index' : previous\_starting\_index,\\ 'starting\_disk' : previous\_starting\_disk,\\ 'offset' : 0, 'length' : old\_size\}$$

and

$$\{'starting\_index' : index\_after\_already\_written,\\ 'starting\_disk' : disk\_after\_already\_written,\\ 'offset' : 0, 'length' : update\_data\_size - old\_size\}$$

E. Support of larger set of configurations

We can also support configuration set larger than 6+2 configuration. Here, we tested that the implementation support up to 64 disks (62+2) configuration. For primitive polynomials for the finite field, we refer to Conway Polynomials list [6] the values of which we obtain from Conway polynomials for finite fields by Frank Luebeck's Webpage [1]. We can assume that the implementation will be able to support a larger number of disks if we provide the primitive polynomials for the respective configurations.

[1]http://www.math.rwth-aachen.de/ Frank.Luebeck/data/ConwayPol/index.html

IV. Experiments

A. Experiment Setup

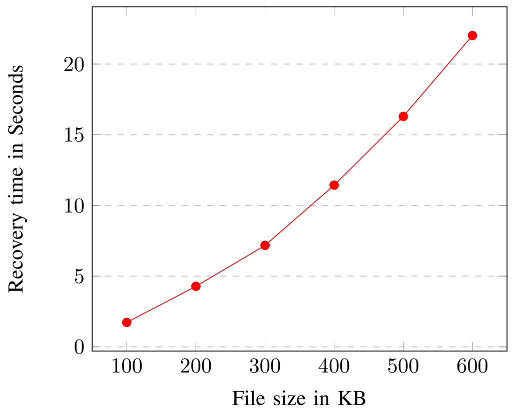
For this project, we are conducting our experiments on a Windows PC with the following specifications: Intel i5-6600k CPU, 16GB of DDR4 RAM and 256GB of SATA SSD storage.

B. Results

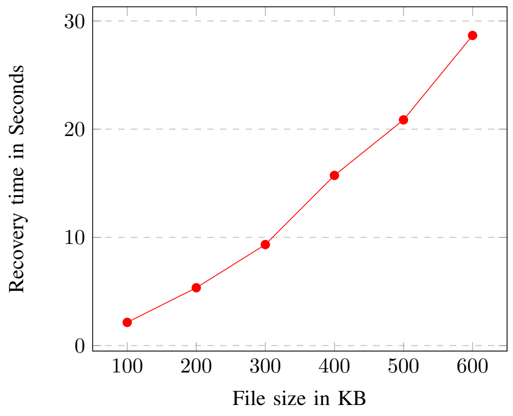
We conducted experiments to measure the time taken to read, write and update files of varying sizes, ranging from 100kb to 600kb. In Figure 3(a), 3(b) and 3(c), we can see there is a linear relationship between the time taken for all three operations and the increasing file sizes. We have also experimented with 1 and 2 disks failures to test the fault tolerance level of RAID-6 technology.

In Figure 4(a), the time taken for recovery 1 deleted disk increases exponentially with larger file sizes. The same can be inferred for recovery 2 deleted disks in Figure 4(b). Lastly, we also varied the number of disks and different chunk sizes to see its effect on the time taken for Read and Write operations.

In Figure 5(a) and 5(b), we have fixed the chunk size to be 128 bytes while varying the number of disks from 8 to 64. We can see that the time taken for Write operation increases as the number of disks increases and the inverse can be said for Read operation as the time taken decreases as the number of disks increases. Understandably, these results react in tandem with the mathematical complexity of calculating P and Q parities for different number of disks. In Figure 5(c) and 5(d), we have fixed the number of disks to be 8 while varying the chunk sizes from 128 bytes to 1024 bytes. We can see that, for both read and write operations, the time taken has drastically reduced as the chunk size increases. This is also understandable because there will be less computations for the chunks as the chunk size increases and the number of chunks decreases.

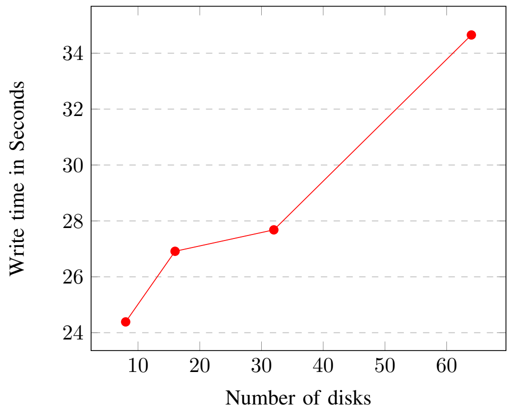


(a) Recovery time for deleting 1 disk

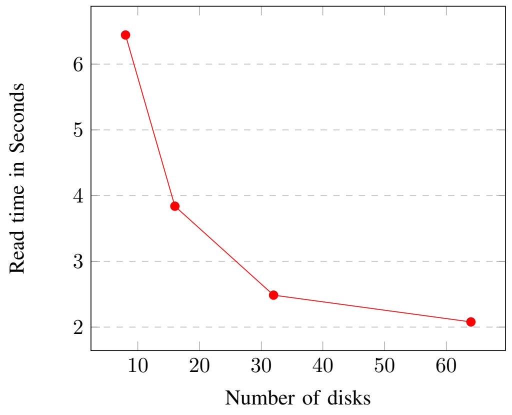


(b) Recovery time for deleting 2 disks

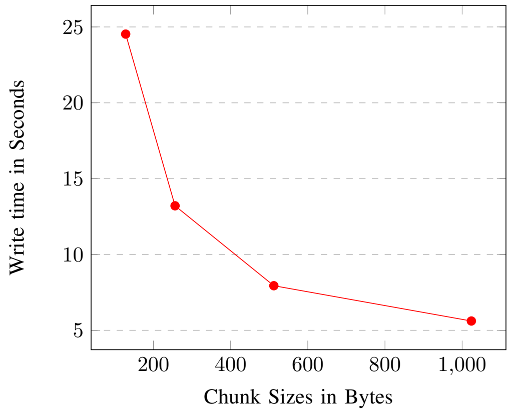
Fig. 4: Time taken for recovering 1, 2 missing disks with 8 disks and chunk size of 128 bytes



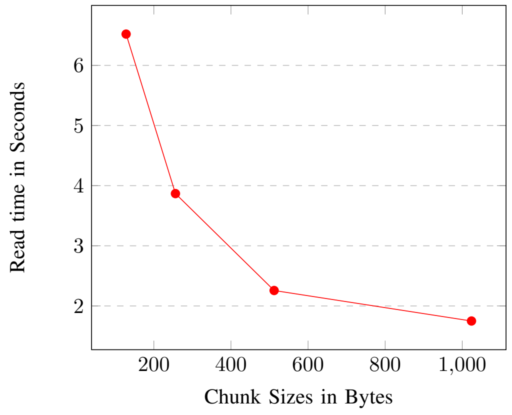
(a) Write time for with different number of disks and chunk size of 128 bytes



(b) Read time for with different number of disks and chunk size of 128 bytes



(c) Write time for with 8 disks and different chunk sizes



(d) Read time for with 8 disks and different chunk sizes

Fig. 5: Time taken for varying number of disks and chunk sizes

V. Conclusion

A. Lessons Learnt During the Project

In the project, we have gained the knowledge of how the RAID6 implementation calculates and stores the parity P and Q and how it can achieve the fault tolerance up to 2 disks failures. As RAID6 uses error checking code with 2 parity bits for several data chunks, this requires more computing resources while it is more space-efficient than replication such as RAID1 and RAID10 since it only generates two additional blocks as parity bits for several blocks of data to tolerate 2 disks failures.

We also learn that we can use RAID 6 with many drives more than $6 + 2$ to tolerate two disk corruptions with small space overhead. However, it will require more calculation, since we need to calculate P and Q with aggregation values over many drives. Same with chunk size where the smaller chunk size saves space allowing less trailing zeros, it takes longer time due to the need of opening and closing the files in the file system to read data. Thus, it is the best to configure the number of disks, chunk size for RAID6 implementation according to the actual situation to achieve the best performance.

B. Conclusion

In the world of big data technology and storage, RAID levels have created a new form of reliability standard for data storage in multiple general purpose disk drives. RAID-6, in particular, extends over RAID5 by adding an additional parity block which raises the fault tolerance level up to 2 disks failure. In the course of studying and implementing RAID-6 technology in our project, we have successfully replicated the capabilities of RAID-6 in the form of distributing data across file folders in Windows OS/Mac OS to emulate actual RAID-6 operations. These capabilities include Write, Read, Update Files and recovery up to two disks in Raid-6 configurations. We have also experimented the time taken for Read and Write operations with varying number of disks and chunk sizes. This helps us to understand and visualise the relationship between the time taken for such operations and the trade-off between chunk sizes and disks.

References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, “A case for redundant arrays of inexpensive disks (raid),” in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’88. New York, NY, USA: ACM, 1988, pp. 109–116. [Online]. Available: http://doi.acm.org/10.1145/50202.50214

[2] Intel. (2005) Intelligent raid 6 theory overview and implementation. [Online]. Available: http://ecee.colorado.edu/~ecen5653/ecen5653/papers/RAID-6-PQ-30812202.pdf

[3] T. Hurley, D. Hurley, and B. Hurley, “Maximum distance separable codes to order,” *CoRR*, vol. abs/1902.06624, 2019. [Online]. Available: http://arxiv.org/abs/1902.06624

[4] E. Horowitz, “Modular arithmetic and finite field theory: A tutorial,” in *Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation*, ser. SYMSAC ’71. New York, NY, USA: ACM, 1971, pp. 188–194. [Online]. Available: http://doi.acm.org/10.1145/800204.806287

[5] S. B. Wicker, *Reed-Solomon Codes and Their Applications*. Piscataway, NJ, USA: IEEE Press, 1994.

[6] L. S. Heath and N. A. Loehr, “New algorithms for generating conway polynomials over finite fields,” 1998.