RAID-6 based distributed storage system

Bernos Guillaume, Bindics Gergery, Finck Quentin Nicolas

***Abstract*—RAID-5 systems are commonly deployed for data protection in most business environments. However, RAID-5 systems only tolerate a single drive failure, and the probability of encountering latent defects of drives approaches 100 percent as disk capacity and array width increase. RAID-6 significantly outperforms the other RAID levels in disk-failure tolerance due to its ability to tolerate arbitrary two concurrent disk failures in a disk array. The underlying parity array codes have a significant impact on RAID-6's performance. The RAID-6 theory is based on Maximum Distance Separable (MDS) coding, as well as Galois Field (GF) mathematics. In this article, we propose an implementation of RAID-6 — a reliable distributed storage system.**

***Keywords*—redundant array of independent disks (RAID), parity, galois field (GF), Maximum Distance Separable (MDS).**

I. INTRODUCTION

The rising demand for capacity, speed and reliability of storage systems has expedited the acceptance and deployment of Redundant Array of Independent Disks (RAID) systems. RAID distributes data blocks among multiple storage devices to achieve high bandwidth input/output and uses one or more error-correcting drives for failure recovery. There are different RAID levels available for different data protection requirements. This paper focuses on RAID-6 systems and technology.

RAID techniques [1] are widely used in modern storage systems to achieve high performance and reliability. By maintaining redundant information within an array of hard disks, RAID can protect data against one or more disk failures. Among the commonly used RAID levels, RAID-1, RAID-4, and RAID-5 can tolerate exactly one single-disk failure; another disk failure will lead to permanent data loss. RAID-10 and RAID-1 tolerate multiple failures by mirroring disks into pairs, but two concurrent disk failures in any mirroring pair will also lead to permanent data loss; moreover, their storage efficiency, at only 50%, is quite low. RAID-6 is specified to tolerate any two concurrent disk failures in a disk array, thus providing a higher level of data reliability.

While RAID-5 and RAID-10 have been most commonly deployed in production data centers, RAID-6 has received increasing attention from the academia and industry recently and is poised to be more widely deployed in data centers to increase data reliability and integrity. The reason behind this increasing interest in the RAID-6 architecture is twofold. On the one hand, recent findings from real world by researchers [2] have reported that partial or complete disk failure rates are actually much higher than previously and commonly estimated, which suggests an urgent need to significantly improve the reliability of RAID systems. On the other hand, while the number and capacity of disks have been growing almost exponentially [3], individual disk failure rates remain largely unchanged, which means that in supercomputing data centers, where there are thousands of hard disks and two or more concurrent disk failures are no longer rare, the ability to tolerate double disk failures becomes ever more important.

RAID-6's double-disk-failure recovery ability is implemented through the underlying erasure codes, such as Reed-Solomon [4] and EVENODD [5]. The performance of a RAID-6 array is largely determined by the erasure code used. In general, we measure a RAID-6 code's performance in terms of its computational complexity, update complexity and storage efficiency. Computational complexity is proportional to the CPU computational overhead during construction and reconstruction. Update complexity indicates the average number of parity blocks affected by an update (write) of a single data block [6, 7]. For RAID-6, every data block is protected by at least two distinct parity blocks on average, so the optimal update complexity is 2. The update complexity can significantly influence the write performance of RAID-6, especially for small writes. On the other hand, storage efficiency is also an important metric, which measures the percentage of storage space used by the parity blocks to protect the data blocks. The smaller the percentage is, the higher the storage efficiency is.

The remainder of this paper is organized as follows. In Section II, we present our system architecture. The section III describes the implementation process. Section IV shows the experiments conducted and the insights discovered. In section V, we propose some solutions to optimize the computation operations. In section VI, we conclude the paper.

II. SYSTEM ARCHITECTURE

A. Environment

Here we present our technical environment. To implement RAID-6, we use the programming language Python version 3.6. We use the pyfinite package for dealing with finite fields and related mathematical operations.

B. Architecture

In our implementation, the RAID-6 system can store any data objects (images, txt, etc). The RAID-6 model is updated when it receives a request from the controller as shown in the following figure.

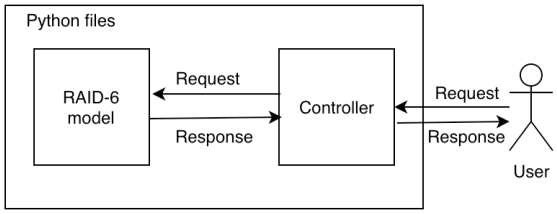

Figure 1 - System architecture

III. Implementation Process

Here is a description of the implementation of a RAID-6 system.

A. Overview of RAID-6

RAID-6 is a specification for storage systems with k + 2 nodes to tolerate the failure of any two nodes. Logically, a typical RAID-6 system appears as depicted in Figure 1. There are k + 2 storage nodes, each of which holds B bytes, partitioned into k data nodes, D0,...Dk-1 and two coding nodes P and Q. The entire system can store kB bytes of data, which are stored in the data nodes. The remaining 2B bytes of the system reside in nodes P and Q and are calculated from the data bytes. The calculations are made so that if any two of the k + 2 nodes fail, the data may be recovered from the surviving nodes.

Figure 2 - Logical overview of a RAID-6 system [10]

Actual implementations optimize this logical configuration by setting B to be smaller than each disk's capacity, and then rotating the identity of the data and coding devices every B bytes. Here is an explanation on how to generate P and Q check values when a stripe of data is being written or updated on to disks. If P and Q elements were always mapped to the same two disks, then every write operation serviced by the array would require updating of the same two disks. To avoid the bottleneck, P and Q check values are rotated through the disks in the array as follows. Figure 3 shows one possible layout of P and Q elements in a RAID-6 array. For stripe 0, drive 0 through 4 store data blocks but drives 5 and 6 store P and Q parity, respectively. For stripe 1, drives 1 through 3 and 6 store data blocks and drives 4 and 5 store P and Q parity respectively. The rotation continues until P and Q arrive back at their original positions, and the sequence repeats.

Stripe	0	1	2	3	4	5	6
0	$D_{(0,0)}$	$D_{(0,1)}$	$D_{(0,2)}$	$D_{(0,3)}$	$D_{(0,4)}$	$P_{(0)}$	$Q_{(0)}$
1	$D_{(1,0)}$	$D_{(1,1)}$	$D_{(1,2)}$	$D_{(1,3)}$	$P_{(1)}$	$Q_{(1)}$	$D_{(1,4)}$
2	$D_{(2,0)}$	$D_{(2,1)}$	$D_{(2,2)}$	$P_{(2)}$	$Q_{(2)}$	$D_{(2,3)}$	$D_{(2,4)}$
3	$D_{(3,0)}$	$D_{(3,1)}$	$P_{(3)}$	$Q_{(3)}$	$D_{(3,2)}$	$D_{(3,3)}$	$D_{(3,4)}$
4	$D_{(4,0)}$	$P_{(4)}$	$Q_{(4)}$	$D_{(4,1)}$	$D_{(4,2)}$	$D_{(4,3)}$	$D_{(4,4)}$
5	$P_{(5)}$	$Q_{(5)}$	$D_{(5,0)}$	$D_{(5,1)}$	$D_{(5,2)}$	$D_{(5,3)}$	$D_{(5,4)}$
6	$Q_{(6)}$	$D_{(6,4)}$	$D_{(6,0)}$	$D_{(6,1)}$	$D_{(6,2)}$	$D_{(6,3)}$	$P_{(6)}$

Figure 3 - RAID 6 Data Blocks on Drives [9]

B. Features implemented

The features implemented for our RAID-6 system are summarized as follows.

- Store and access abstract "data objects" across storage nodes for fault-tolerance.
- Include mechanisms to determine failure of storage nodes.
- Carry out rebuild of lost redundancy at a replacement storage node : recovery of P and Q drive failure, recovery of Q and single data drive failure, recovery of P and single data drive failure, recovery of dual data drive failure.
- 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, using a more full-fledged implementation).

One important feature that we did not have time to implement is mid-data block writing. By storing different files on the same chunk, we could have further improved the performances of this RAID-6 implementation.

C. Explanation

The different functions are explained below, but also commented in the full implementation:

- ***__init__***: Allow the user to define certain characteristics of RAID-6 such as the CHUNK_SIZE or the number of disks. It will also reinitialize any previous disk created.
- ***increase_disk_index***: Simple function allowing to increase the disk index to know where to write next.
- ***update_disk_info***: Simple function allowing to update the disk info to know if a block is full.
- ***store_parity***: Store the parity of an index thanks to the parity file.
- ***write_data***: Write data to RAID-6 disks with the associated name. It will create a temporary file in disks.
- ***write_data_from_file***: Write data to RAID-6 with the associated name from the file. If we want to write the file to a specific chunk, it gives a list in chunk_to_write. Offset allows to write from a certain part of the file (update).
- ***is_P_index***: Determine if an index is the P_index since P is stored in a cyclic way.
- ***is_Q_index***: Determine if an index is the Q_index since Q is stored in a cyclic way.
- ***actual_disk_index***: Give the current index of a disk since disks are stored in a cyclic way.
- ***read_one_chunk***: Read one index (all the disks), reading data and recovering disk loss. Some disks can be excluded while recovering data. Self recovering can be turned off in order to accommodate reading incomplete index.
- ***read_data***: Read data to console given an index, disk and length to read. It is not supposed to be used by end-user.

- ***read_data_to_file***: Read data to a file given an index, disk and length to read. It is not supposed to be used by end-user. Data can be added at the end of the file.
- ***recovering_disks***: Allow to recover up to two deleted disks. It uses the parity file to do all the computations.
- ***delete_data***: Delete data based on their name. Data will still be on disk but can be rewritten on.
- ***update_data_from_file***: Update data stored based on their name. It will compare data to only store changed data.
- ***get_data_from_name***: Get the data from their name.
- ***get_data_to_file_from_name***: Write data to a file from their name.

IV. Experiment Results

We conduct experiments on the relevant following criteria to evaluate our RAID-6 storage system:

- Save a file
- Read a file
- One disk recovery
- Two disks recovery
- Delete and write to update a file
- Update a mutable file

These criteria will be measured on two axes: time and chunk size.

A. *Save a file*

Here we measure the execution time to save a file over the increase of the chunk size. In this experiment, we use a picture as a file.

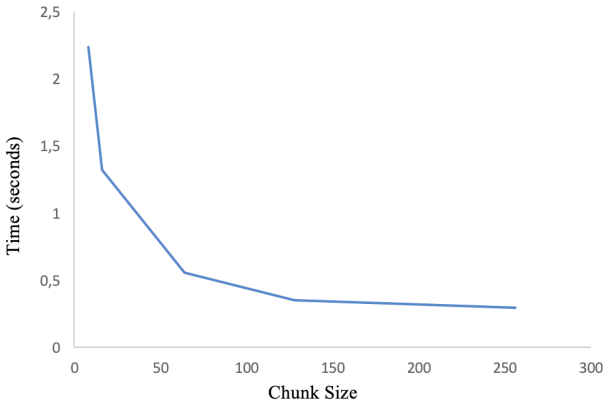

Figure 4 – Execution time to save a file

As shown in the Figure 4, the time to save a file decreases when the chunk size increases. It is because we need less time to open and close files on which we currently work. For instance, it decreases from 2.23 seconds (chunk size is 8) to 0.29 seconds (chunk size is 256).

B. *Read a file*

We measure the execution time to read a file over the increase of the chunk size. In this experiment, we use a picture as a file also.

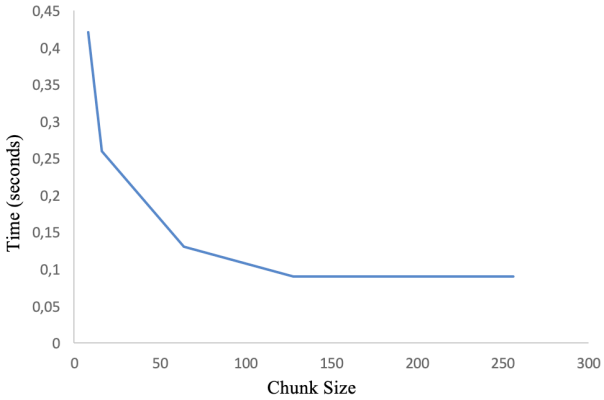

Figure 5 - Execution time to read a file

As shown in the Figure 5, we notice that the time to read a file decreases when the chunk size increases. For instance, it decreases by 78.57% from chunk size 8 to chunk size 256.

C. *One disk recovery*

We measure the execution time to perform a one disk recovery over the increase of the chunk size.

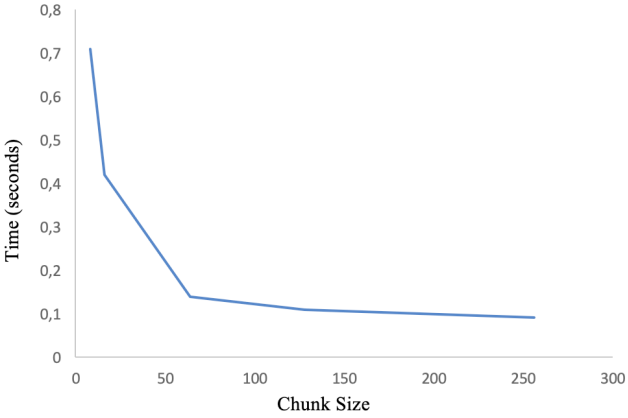

Figure 6 - Execution time for one disk recovery

As shown in the Figure 6, we notice that the time to perform a disk recovery decreases when the chunk size increases. It is because there are less and less computation operations on P and Q when the chunk size increases.

D. *Two disks recovery*

We measure the execution time to perform a two disks recovery over the increase of the chunk size.

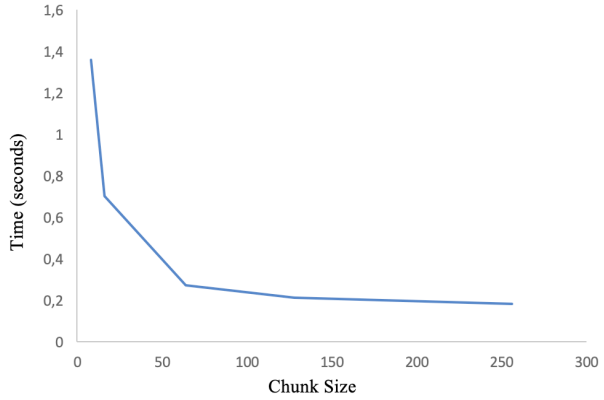

Figure 7 - Execution time for two disks recovery

As shown in the Figure 7, we notice that the time to perform a two disks recovery decreases when the chunk size increases. It decreases from 1.36 seconds (chunk size is 8) to 0.18 seconds (chunk size is 256). It is also noticeable that the execution time is approximately the double compared to the execution time for a single disk recovery. It is because there are more computation operations on P and Q for a two disks recovery.

E. *Delete and write to update a file*

We measure the execution time to perform a delete and write operation to update a file over the increase of the chunk size.

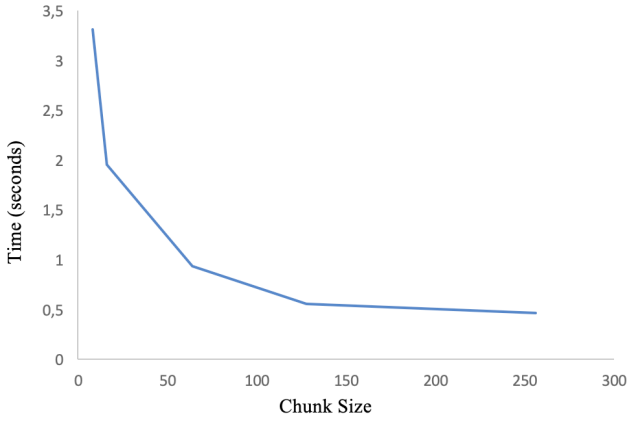

Figure 8 - Time to update a file (Delete/Write)

As shown in the Figure 8, we notice that the time to perform a delete and write operation to update a file decreases when the chunk size increases.

F. *Update a mutable file*

We measure the execution time to perform an update on a mutable file over the increase of the chunk size.

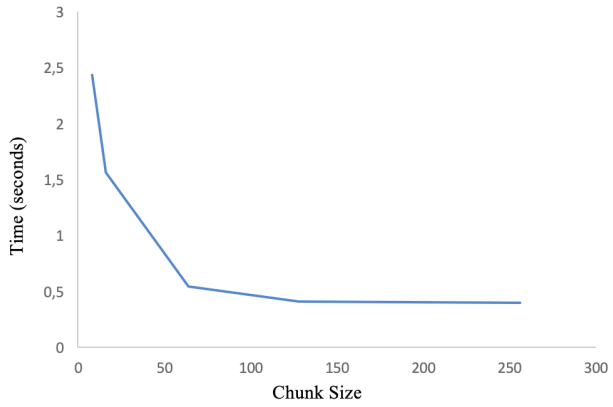

Figure 9 - Time to update a mutable file

As shown in the Figure 9, the time to perform an update on a mutable file decreases when the chunk size increases. We notice that the execution time is faster for an update on a mutable file compared to do an update by deleting and writing a file. Indeed, the execution time decreases by 26.59% (chunk size is 8) and by 13.04% (chunk size is 256) when executing an update on a mutable file. It is because there are less disk computation operations when updating a mutable file.

V. Optimize the Computation Operations

A. *Our contribution*

We could multithread the implementation in order to take full advantage of every core of the computer since Python is not multi-threaded by default.

B. *Parity array coding*

Reed-Solomon coding is a powerful erasure coding technique with strong error-correcting capability. It uses Galois Field arithmetic during encoding and decoding. Galois Field addition is equivalent to XOR, but its multiplication is much more complicated, usually involves a table-lookup operation that alleviates the computation intensity. Unlike Reed-Solomon coding, parity array coding depends solely on XOR operations during encoding and decoding. Thus, parity array coding is more preferable for RAID storage systems and could be a way for optimization.

C. *P-Code*

In the literature, there are many solutions proposed to improve RAID-6. One of them is P-Code [8], a vertical code, in which its parities are calculated only with XOR operations and spread evenly across component disks. Unlike horizontal codes that concentrate parity blocks on extra dedicated parity disks, this feature of P-Code results in balanced disk accesses during the write-dominated periods. P-Code complements the other two optimal RAID-6 codes: X-code and the tweaked RDP. P-Code is an MDS code with optimal storage efficiency, optimal construction and reconstruction computational complexity, and optimal update complexity.

D. *Acceleration with the Intel IOP333*

The computation operations could be optimized by using the Intel IOP333 [9]. It combines the Intel XScale core with powerful new features to create Intel's latest and most powerful intelligent I/O processor. The main enhancement in the IOP333's Application Acceleration Unit (AAU) is the ability to accelerate the GF multiplication for RAID-6 implementations.

The GF multiplication is an important operation when computing the check value and when recovering one or two failed data drives. The AAU in the IOP333 relieves the CPU from doing time-consuming GF multiplication, thereby increasing the overall system performance. The AAU performs GF multiplications in conjunction with the XOR. The AAU multiplies source data blocks with GF field elements before the XOR operation when the P+Q RAID-6 feature is enabled. Additionally, the chain descriptor in the AAU allows the user to chain several RAID-6 computations together and offloads the RAID-6 computation to hardware instead of burdening the CPU.

VI. Conclusion

A RAID-6 storage system provides enhanced data protection for critical user data. But, it is challenging to implement a RAID-6 system due to its complexity. In this paper, a RAID-6 implementation was presented. Additionally, experiments were conducted and optimization of computation operations was discussed.

ACKNOWLEDGMENT

We thank the professor Anwitaman DATTA for teaching the course CE7490 Advanced Topics in Distributed Systems. This work was done during an exchange semester at Nanyang Technological University in 2018.

REFERENCES

[1] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD,* ACM, pp. 109-116, June 1988.

[2] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *FAST '07,* San Jose, CA, February 2007.

[3] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *FAST '07,* San Jose, CA, February 2007.

[4] J. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience,* 27(9):995-1012, 1997.

[5] 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.

[6] J. Hartiline. *R5X0: An efficient high distance parity-based code with optimal update complexity.* Research Report # RJ10322 (A0408-005), IBM Research Division, 2004.

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

[8] C.Jin, H.Jiang, D.Feng and L.Tian. *P-Code: A New RAID-6 Code with Optimal Properties*, 2009.

[9] Intel, *Intelligent RAID-6 Theory Overview and Implementation*, 2005

[10] James S. Plank. *A New MDS Erasure Code for RAID-6*, September, 2007.