2015 IEEE 39th Annual International Computers, Software & Applications Conference

Distributed Metaserver Mechanism and Recovery Mechanism Support in Quantcast File System

Su-Shien Ho[1], Chun-Feng Wu[1], Jiazheng Zhou[1],
Wenguang Chen[2], Ching-Hsien Hsu[3], Hung-Chang Hsiao[4], Yeh-Ching Chung[1]

[1]Department of Computer Science, National Tsing Hua University, Taiwan
[2]Department of Computer Science and Technology, Tsinghua University, Beijing, China
[3]Department of Computer Science and Information Engineering, Chung Hua University, Taiwan
[4]Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan

{sushien, cfwu, jzhou}@sslab.cs.nthu.edu.tw,
cwg@tsinghua.edu.cn, chh@chu.edu.tw, hchsiao@csie.ncku.edu.tw, ychung@cs.nthu.edu.tw

***Abstract*—With the need of data storage increases tremendously nowadays, distributed file system becomes the most important data storage system in cloud computing. In distributed file system development, there are many researchers work hard to refine the architecture to provide scalability and reliability. In our work, we propose a distributed metaserver system including metaserver scale-out, metadata replication, metaserver recovery, and metaserver management recovery mechanisms. In our experiments, the proposed system can increase the capacity of metadata and increase the reliability by fault tolerance mechanism. The overhead of read/write data is very little in the proposed system as well.**

Keywords-Distributed File System*; *Metadata; Metaserver*; *Fault Tolerance; QFS

1. INTRODUCTION

Big data is the most popular area in cloud computing. Many distributed file systems serve for big data to store the large-scale data, including HDFS[24], GFS[12], and QFS[17]. There is a metaserver in these distributed file systems, but this single node server limits the system scalability and metadata reliability. As the scale of storage cluster becomes much larger today, we need to improve the reliability of metadata management. We construct a lightweight and high-level metaserver management in QFS. The concept of this paper can also be applied to those distributed file systems with a single metaserver.

We propose a distributed metaserver system based on QFS and accomplish the following contributions:

- High-level metaserver management.
- Building a scalable metaserver cluster.
- Recovering failed metaservers.
- Recovering failed metaserver management.

The rest of the paper is organized as follows. In Section 2, we show some preliminaries and related work. Section 3 introduces the Quantcast File System. The proposed distributed metaserver system is presented in Section 4. The experiments and analysis are shown in Section 5. Section 6 concludes our work and shows future work.

2. RELATED WORK

The distributed file system (DFS) is one of the most important components in cloud computing today. Google File System (GFS)[12] and Hadoop Distributed File System (HDFS)[2] are the most famous and the greatest DFSs. The block-based storage and object-based storage[16] become more and more popular. (TABLE 1)

TABLE 1. DFSs with or without metadata management nodes.

Architecture	System
DFS with metadata management nodes	GFS [12] (file level) HDFS [2] (file level) QFS [17] (block level) KFS [5] (block level) Ceph DFS [26] (object level) MooseFS [9] (object level) Lustre FS [23] (object level)
DFS without metadata management nodes	Amazon S3[1] (object level) Facebook Cassandra[15] (object level) OpenStack Swift[10] (object level)

2.1. Hadoop Distributed File System (HDFS) and Ceph Distributed File System

The big data analysis with MapReduce is the popular in cloud computing nowadays. Because MapReduce framework is popular, its storage system HDFS gets a lot of attentions and many research works want to improve the storage system performance, reliability, and scalability.

The Ceph DFS is the most representative object-based storage system today. The Ceph is a high scalability and high performance file system[26]. Ceph consists of metaserver cluster and object storage cluster.

2.2. *Metadata Indexing Methods*

Dynamic sub-tree partitioning[28] indexing method is used in Ceph DFS for multiple metaservers. Dynamic sub-tree partitioning can distribute the metadata to several metaservers with load balancing.

2.3. *Related Research Work*

There are more and more research works focusing on metadata accessing time, metadata balancing in metaserver cluster, and metadata reliability since the object-based storage system is the mainstream storage system today. Many companies and researchers attempt to improve the single metaserver to multiple metaservers to increase scalability and reliability in the storage system. They are shown in literatures [3], [6], [7], [8], [11], [14], [19], [22], and [27].

3. QUANTCAST FILE SYSTEM

Quantcast file system (QFS) [17] is developed in the frame of the Kosmos File System (KFS) [5] which is an open-source distributed file system implemented in C++. KFS has three parts: client library (a set of commands), metaserver, and chunk server. Figure 1 shows Quantcast file system architecture. QFS improves read/write performance by parallel reading/writing from chunk servers.

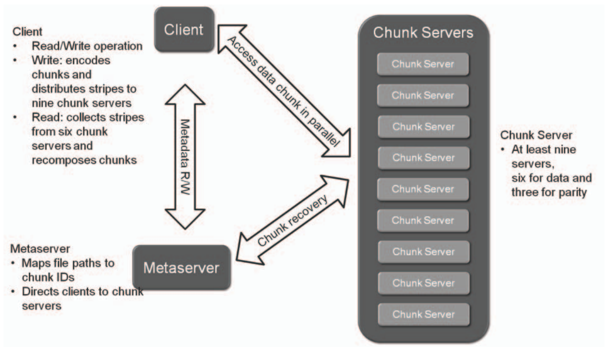

Client
• Read/Write operation
• Write: encodes chunks and distributes stripes to nine chunk servers
• Read: collects stripes from six chunk servers and recomposes chunks
Client
Access data chunk in parallel
Metadata R/W
Chunk recovery
Metaserver
• Maps file paths to chunk IDs
• Directs clients to chunk servers
Metaserver
Chunk Servers
Chunk Server
Chunk Server
• At least nine servers, six for data and three for parity

Figure 1. The Quantcast file system architecture.

3.1 *Metaserver*

The QFS metaserver keeps all directory and file structure of the file system, i.e. metadata. The metaserver monitors redundancy block and recreates missing data. In QFS, a file will be encoded into six chunk data and three redundancy parity blocks and distributed to nine chunk servers. They are encoded with Reed-Solomon code, annotated as Reed-Solomon 6+3. We can lose 3 chunks of data at most.

3.2 *Chunk Server*

Each chunk server stores chunks as files on the local file system and each chunk file is named as [*file-id*].[*chunk-id*].*version.* The chunk servers' primary work is accepting connections from clients and doing some recovery work guided from metaserver.

3.3 *Client*

The client contains many read/write operation commands. Client can use this set of commands to deal with basic read and write, and Reed-Solomon encoded read and write. There are many works ([4], [13], [18], [20], [21], [25]) about Reed-Solomon code. Figure 2 shows an example of RS(6,3). Figure 3 shows the encoding method in QFS client.

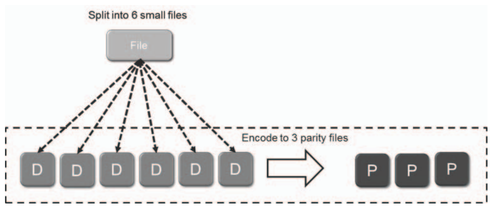


Figure 2. An example of RS(6,3).

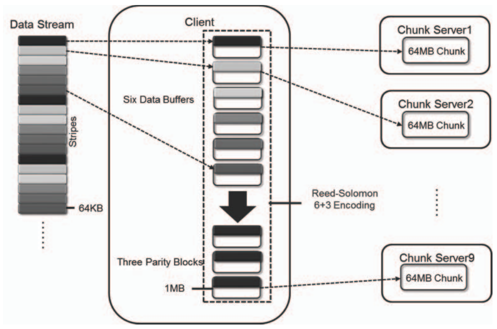


Figure 3. QFS client encodes data into Reed-Solomon 6+3 code.

4. DISTRIBUTED METASERVER MECHANISM AND RECOVERY MECHANISM

We want to build a distributed metaserver system with high reliability and scalability by adding fault tolerance mechanism.

4.1 Overview

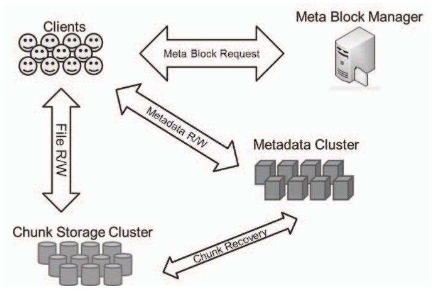
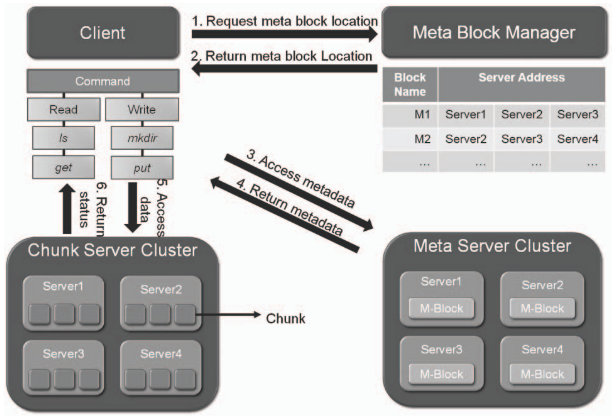


Figure 4. The architecture of distributed metaserver system in QFS.

This system consists of three main components: client, Metadata Block Manager (MBM), and server cluster (see Figure 4). Client interacts with MBM to get the block location, and then client will get the block name and server addresses.

4.2 System Architecture and Components Implementation

Figure 5 shows the details of every component. We will discuss them as follows.

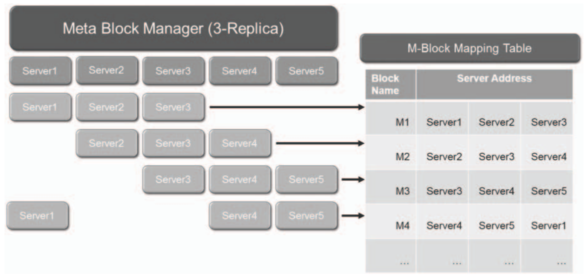


Block Name	Server Address		
M1	Server1	Server2	Server3
M2	Server2	Server3	Server4
...	...	...	...

Figure 5. The architecture of distributed metaserver system in QFS.

The metadata block (M-Block) is a file which contains numerous metadata records. Each metadata record contains some information about file or directory attributes, including *name*, *mode*, *user*, *group*, *creation time*, *metadata id*, and etc.

Metadata Block Manager (MBM) is a lightweight service. It only keeps a small M-Block mapping table, creating mapping from M-Block name to server addresses. MBM is the most important component in the metaserver cluster. The role of MBM is a scheduler. The mechanism of MBM is shown in Figure 6. MBM assigns M-Block to three servers with 3 replicas setting.



Block Name	Server Address		
M1	Server1	Server2	Server3
M2	Server2	Server3	Server4
M3	Server3	Server4	Server5
M4	Server4	Server5	Server1
...	...	...	...

Figure 6. MBM uses Round Robin scheduling to build the mapping table.

4.2.1 Metadata Replication Mechanism

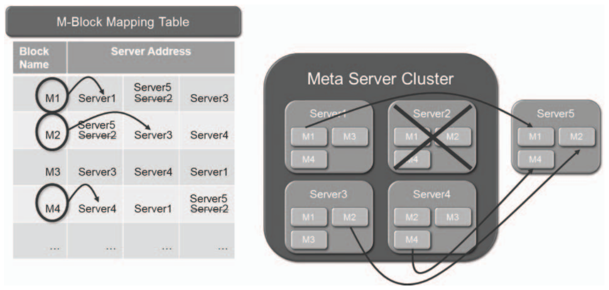
To introduce our metadata replication mechanism, we refer to the data replication mechanism in Hadoop distributed file system (HDFS) [24] and Ceph distributed file system [26]. we get a pair of M-Block mapping to several metaserver addresses before writing metadata to metaserver. We will check status of all metaservers before starting to write. After checking the status is good, client will send metadata to multiple metaservers and each metaserver will write the metadata into M-Block.

4.2.2 Fault Tolerance Mechanism

Our system supports *N*-replica mechanism, so we can construct an *n*+(*N*-1) metadata system. That is, the system can support at most (*N*-1) servers' failure. During this period, the system works but it is not in a good condition.

4.2.3 Metaserver Recovery Mechanism

We have to recover metaservers before all metaservers fail. If all metaservers fail, we cannot get metadata from this DFS, and this system is crashed.



Block Name	Server Address		
M1	Server1	Server5 Server2	Server3
M2	Server5 Server2	Server3	Server4
M3	Server3	Server4	Server1
M4	Server4	Server1	Server5 Server2
...	...	...	...

Figure 7. The metaserver recovery mechanism.

Figure 7 shows that the metadata recovery mechanism. We will show how to recover one metaserver by the following steps. Suppose Server2 is broken.

Step 1. New metaserver Server5 wishes to replace the failed metaserver Server2. Server5 sends *recovery* request to MBM.

Step 2. MBM will look up all M-Blocks owned by failed metaserver Server2. In Figure 7, MBM will find M1, M2 and M4 originally locate in Server2.

Step 3. MBM will look up the record of M-Block M1, M2, M4 found in step2. MBM will find the metaservers Server1, Server3 and Server4 for the replica of M1, M2, and M4, respectively.

Step 4. MBM will send <*M-Block name*, *location*> pairs to new metaserver Server5. The content of message is <M1, Server1>, <M2, Server3>, <M4, Server4>.

Step 5. New metaserver Server5 will get M-Blocks according to received message. Server5 will get M1 from Server1, M2 from Server3 and M4 from Server4.

4.2.4 Metadata Block Manager Recovery Mechanism

Metadata Block Manager is the most important component in our system. If MBM crashes, we can not perform any operations. Figure 8 shows the MBM recovery mechanism. New MBM should send recovery request to each metaserver, and each metaserver will return several <*M-Block name*, *location*> pairs to new MBM. Then new MBM receives return messages from metaservers, and new MBM will use these pairs to rebuild M-Block mapping table.

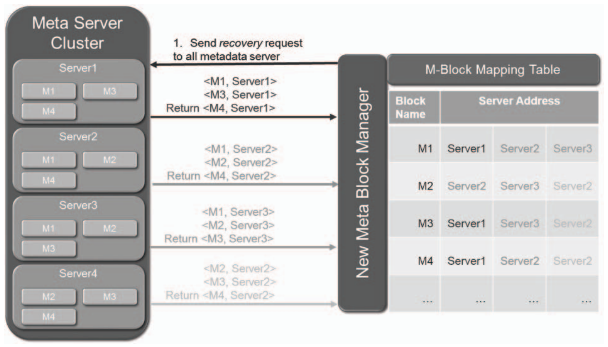


Figure 8. The MBM recovery mechanism.

5. Experiments and Analysis

5.1. Experimental environment

Our experimental environment consists of nine server nodes. In these nine nodes, there are one Meta Block Manager, seven nodes for metaservers, and nine nodes for chunk servers.

5.2. Metadata Capacity vs. System Capability

We use the default 3-replica mechanism in metaserver cluster and use four to seven servers to observe the variation of metadata capacity. We limit the memory size of 200MB to simulate the hardware memory size in the node. Figure 9 shows the results of the experiments.

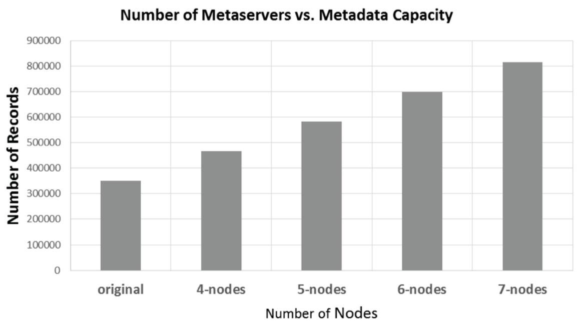


Figure 9. The number of metaservers vs. metadata capacity.

We use *mkdir* operation to put metadata to metaservers. We can find that the number of metadata records (capacity) increase 33% in the 3-replica mechanism when we add one more metaserver to the system. With the proposed distributed metaserver system, we can increase the capability of the original QFS.

5.3. Client Performance

We will test four main commands in our system, and compare the performance with the original QFS. We choose *mkdir*, *ls*, *get* and *put* for experiments.

5.3.1 mkdir Command

mkdir only requires communication with the metaserver. The size of metadata is fixed for one command, so we can accurately calculate the execution times per second.

In Figure 10, no-replica stands for the original QFS. Since client should communicate with MBM, and metadata should write to multiple metaservers due to the replication mechanism in our system, we can see the performance decreases between no-replica and 3-replica. After the replication mechanism is used, the overhead of execution time is just related to the number of replicas (i.e., write times), so the result shows the linear performance degradation between 3-replica to 6-replica.

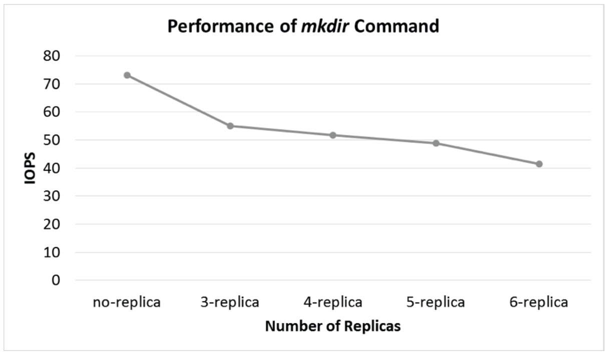


Figure 10. The performance of *mkdir* command.

5.3.2 ls Command

ls command will get all metadata in a directory. We will list a directory with ten thousand records one hundred times. Figure 11 shows the result of *ls* performance. We can see that the performance decreases dramatically between no-replica and 3-replica, since we add replication mechanism (3-replica) into the original QFS (no-replica). The rest of result is similar to that of *mkdir* command. The performance of *ls* command is the worst in our system when comparing with that in original QFS.

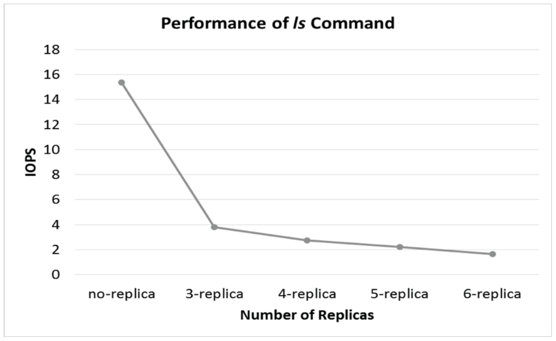


Figure 11. The performance of *ls* command.

5.3.3 put Command

In our system, *put* means write files into the file system, and we will test the write throughput. We test four groups, including 1MB, 10MB, 100MB, and 500MB. Each group is tested with five settings (no-replica, 3-replica, 4-replica, 5-replica, and 6-replica).

Figure 12 shows that the performance of small files is the worst. With the number of replicas increases, the performance will degrade in each group since we should write more metadata to metaservers. Writing metadata many times will cause the overhead in our system, the proportion of the overhead is getting smaller when the file size is getting larger. Therefore, the overhead is not obvious with large file size.

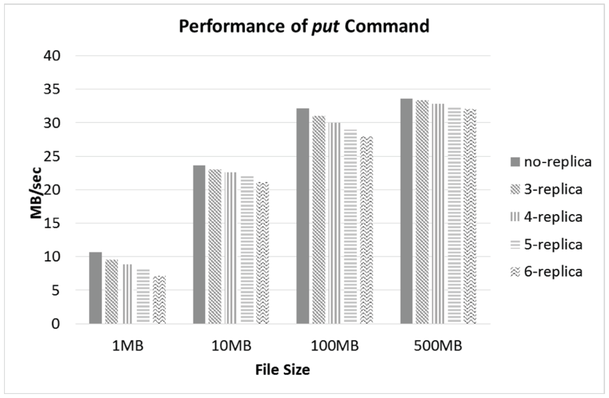


Figure 12. The Performance of *put* Command.

5.3.4 get Command

get command means read files from the file system, and we will test the read throughput. We will choose test data as shown in section 5.3.3.

Figure 13 shows that the performances of *get* command are close among all settings for a given data size, and the overhead is not obvious. Because it only needs to check if the metadata exists in the metaservers in B+ tree in memory, it spends very little time checking it.

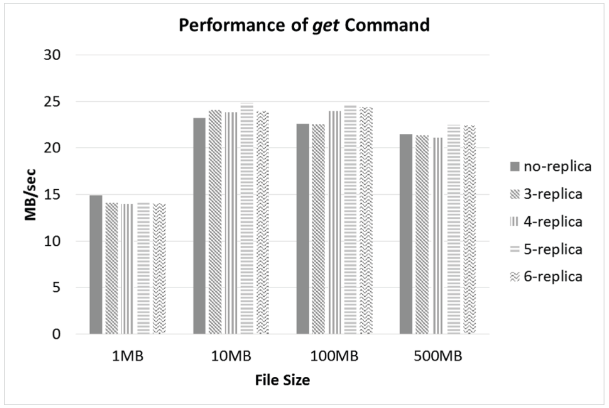


Figure 13. The performance of *get* command.

5.4. Fault Tolerance Mechanism vs. System Performance

In *n*-replica mechanism system, we allow *n*-1 nodes crash in our system. Figure 14 shows that the performance of read files will not be affected when some metaservers crash. We will get metadata from next healthy metaserver when we cannot communicate with the current metaserver.

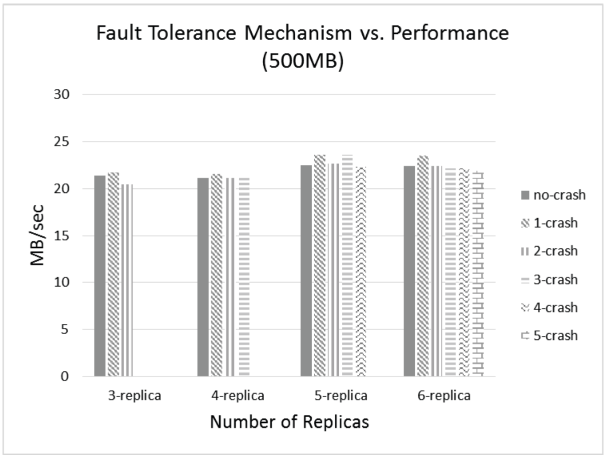


Figure 14. Fault tolerance mechanism vs. system performance with 500MB files.

6. Conclusions and Future Work

For distributed file system, data and metadata reliability becomes more important than before. We construct a lightweight and high-level metaserver management in QFS. We provide a distributed metaserver system with replication and recovery mechanisms. We keep the characteristics of parallel reading/writing and erasure coding from QFS in our system. The proposed method only introduces little overhead of communication with multiple metaservers, and we can build a high reliable fault-tolerant system. When we read/write files, the throughput is nearly not affected by replication mechanism. Our work is the first one to improve the scalability and reliability based on QFS. The concept of this paper can also be applied to those distributed file systems with a single metaserver.

In the future, we will resolve the issue that metadata always occupy the memory in metaserver. We want to offload the metadata from memory but we will add a caching mechanism to enhance the access performance.

Acknowledgement

The work of this paper is partially supported by Ministry of Science and Technology under contract MOST 104-2218-E-007-002-.

References

1. *Amazon S3*. Available from: http://aws.amazon.com/s3/.
2. *Apache Hadoop*. Available from: http://hadoop.apache.org/.
3. *Apache Hadoop 2.1.1-beta*. Available from: http://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-hdfs/Federation.html.
4. *HDFS and Erasure Codes (HDFS-RAID)*.
5. *KFS*. Available from: https://code.google.com/p/kosmosfs/.
6. *LevelDB - A Fast And Lightweight Key/Value Database Library by Google*.
7. *MapR*. Available from: http://www.mapr.com.
8. *MapR Documentation*. Available from: http://doc.mapr.com/display/MapR/Home.
9. *MooseFS.*
10. *Openstack Swift*. Available from: http://docs.openstack.org/developer/swift/.
11. *Panasas Hardware Architecture*. Available from: http://www.panasas.com/products/hardware-architecture.
12. S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google File System*, in *ACM SIGOPS Operating Systems Review*. 2003. p. 29-43.
13. C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, *Erasure Coding in Windows Azure Storage*, in *USENIX Annual Technical Conference*. 2012. p. 15-26.
14. K. Kulkarni, K. Ren, S. Patil, and G. Gibson, *Giga+TableFS on PanFS: Scaling Metadata Performance on Cluster File System.* 2013.
15. A. Lakshman and P. Malik, *Cassandra - A Decentralized Structured Storage System*, in *ACM SIGOPS Operating Systems Review*. 2010. p. 35-40.
16. M. Mesnier, G.R. Ganger, and E. Riedel, *Object-Based Storage*, in *IEEE*. 2003. p. 84-90.
17. M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, *The Quantcast File System.* Proceedings of the VLDB Endowment, 2013: p. 1092-1101.
18. D.S. Papailiopoulos, J. Luo, A.G. Dimakis, C. Huang, and J. Li, *Simple Regenerating Codes: Network Coding for Cloud Storage*, in *INFOCOM, 2012 Proceedings IEEE*. 2012. p. 2801-2805.
19. S.V. Patil, G.A. Gibson, S. Lang, and M. Polte, *GIGA+: Scalable Directories for Shared File Systems*, in *Proceedings of the 2nd international workshop on Petascale data storage*. 2007. p. 26-29.
20. K.V. Rashmi, N.B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, *A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster*, in *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems*. 2013.
21. I. Reed and G. Solomon, *Polynomial Codes Over Certain Finite Fields*, in *Journal of SIAM*. 1960. p. 300-304.
22. K. Ren and G. Gibson, *TABLEFS: Enhancing Metadata Efficiency in the Local File System.*, in *USENIX Annual Technical Conference*. 2013. p. 145-156.
23. P. Schwan, *Lustre: Building a File System for 1,000-node*, in *Proceedings of the 2003 Linux Symposium*. 2003. p. 380-386.
24. K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The Hadoop Distributed File System*, in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, IEEE Computer Society: Washington, DC, USA. p. 1–10.
25. H. Weatherspoon and J.D. Kubiatowicz, *Erasure Coding vs. Replication: A Quantitative Comparison.* Peer-to-Peer Systems, 2002: p. 328-337.
26. S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn, *Ceph: A Scalable, High-Performance Distributed File System*, in *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, USENIX Association: USENIX Association. p. 307-320.
27. X. Xie, Y. Yang, and Y. Lu, *A Zones-Based Metadata Management Method for Distributed File System*, in *Trustworthy Computing and Services*. 2014. p. 169-175.
28. G. Zhou, Q. Lan, and J. Chen, *A Dynamic Metadata Equipotent Subtree Partition Policy for Mass Storage System*, in *Japan-China Joint Workshop*. 2007, IEEE. p. 29-34.