MRFS: A Distributed Files System with Geo-replicated Metadata

Jiongyu Yu, Weigang Wu, Di Yang, and Ning Huang

Department of Computer Science, Sun Yat-sen University, Guangzhou 510006, China
{yujiongy,yangdi5}@mail2.sysu.edu.cn, wuweig@mail.sysu.edu.cn

Abstract. Distributed file system is one of the key blocks of data centers. With the advance in geo-replicated storage systems across data centers, both system scale and user scale are becoming larger and larger. Then, a single metadata server in distributed file system may lead to capacity bottleneck and high latency without considering locality. In this paper, we present the design and implementation of MRFS (Metadata Replication File System), a distributed file system with hierarchical and efficient distributed metadata management, which introduces multiple metadata servers (MDS) and an additional namespace server (NS). Metadata is divided into non-overlapping parts and stored on MDS in which the creation operation is raised, while namespace and directory information is maintained in NS. Such a hierarchical design not only achieves high scalability but also provides low-latency because it satisfies a majority of requests in local MDS. To address hotspot issues and flash crowds, the system supports flexible and configurable metadata replication among MDSs. Evaluation results show that our system MRFS is effective and efficient, and the replication mechanism brings substantial local visit at the cost of affordable memory overhead under various scenarios.

Keywords: Distributed file system, Metadata management, data replication.

1 Introduction

With the emergence and development of large-scale geo-replicated application, distributed file system, as a general storage infrastructure, has attracted more and more attentions in the past years. One of the key challenges in distributed file system lies in big data processing. Data has become of greater importance, and storage demand also has an explosive growth, which has increased exponentially exceeding petabytes and getting close to exabytes in certain applications [1].

Consequently, high scalability and providing low-latency response have been two critical factors in the design of distributed file system for geo-replicated applications. Since metadata transactions account for over 50% of all file system operations [2], most modern distributed file systems decouple the metadata transactions from actual data accesses so as to achieve scalability and availability. Dedicated metadata server (MDS) is deployed to process metadata transactions while storage nodes are to store actual data. Metadata management then becomes a critical issue in file systems.

X.-h. Sun et al. (Eds.): ICA3PP 2014, Part II, LNCS 8631, pp. 273–285, 2014.
© Springer International Publishing Switzerland 2014

However, in most of existing distributed file systems, there is usually only single MDS node, which is prone to be a bottleneck if the number of files is very large [3]. Though a few designs introduce distributed metadata model, it is costly to maintain a global and consistent namespace. Besides, these works consider only single datacenter, which are not suitable for systems that spread across multi-datacenters, i.e. geo-replicated systems.

To solve the issues discussed above, this paper presents a two-tiered metadata management scheme with metadata across multiple metadata servers (MDSs). We separate the file metadata and namespace information and store them in MDS (metadata server) and NS (namespace server), respectively. File metadata is initially stored in the MDS where the creation operation is raised. We call such MDS as primary MDS of the metadata in the rest of the paper. Based on the rule of locality and visit pattern of applications, we assume that a majority of client requests are satisfied in the primary MDS and therefore will not cost high network latency among different datacenters. In addition, multiple MDSs can serve requests simultaneously and potentially improve the performance and concurrency. Moreover, NS maintains a global and consistent namespace which supports fast response for directory lookup and modification requests from clients. Such operation is supposed to cause high delay in other designs since the namespace is scattered among different MDSs and involved entries should be located and merged upon each query.

We propose to extend MooseFS, a well-known open source distributed file system, by modifying the entire metadata module and introducing a new role, namespace server. More precisely, we refer to the basic blocks of MooseFS, like communication mechanism and client module, but recode the whole metadata service module, i.e., redesign the data structure of file metadata and namespace information and implement the replication mechanism. The metadata of the file system is dynamically partitioned into non-overlapping parts upon clients' creation request. Namely, each metadata node is in charge of one subset of the whole metadata in file system. It should be clarified that, in the context of MRFS, clients are applications or front-end servers that issue read/write operations on behalf of real world users. All processes and interaction with NS hiding behind the primary MDS are transparent to clients.

On the other hand, to alleviate flash crowds and hotspot issues, we implement a flexible and configurable replication mechanism. Popular metadata entries are replicated to other MDSs that query them frequently during a pre-defined period. The replication threshold and time interval are both configurable to meet diverse requirements. With replication, high-latency access across MDSs is reduced and load balance among the overall system is enhanced.

MRFS is tested in real deployment. Experiments have been conducted under several scenarios to validate our design and evaluate the performance of the new file system. The results show that our design is effective and efficient. Besides, the newly-added replication mechanism largely reduces the across-datacenter communication at the cost of affordable memory overhead.

The rest of the paper is organized as follows. We briefly review existing works on metadata management for distributed file systems in Section 2. Section 3 describes the design and implementation of MRFS. The experiments and results are reported in Section 4. In the last section, we conclude the paper and discuss about future works.

2 Related Work

According to the metadata server type, we categorize existing works on metadata management into three classes, i.e., centralized metadata management, distributed metadata management, and implicit metadata management.

Most of popular and famous distributed file systems, i.e., HDFS [4], GoogleFS [5] and MooseFS [6], use a centralized metadata server. The advantage of such design lies in easy implementation and management. However, the drawback is also obvious. Since metadata is maintained in main memory, as the scale of file count increases to extremely large, it may become a bottleneck and limit the scalability of overall system.

Quite a number of distributed metadata management schemes have been introduced to solve the problems of centralized ones. With static subtree partitioning, metadata is divided in to non-overlapped parts and distributed into individual MDSs by system administrator. This approach is simple and relatively efficient and used by many famous implementations like Coda [7] and Sprite [8]. However, it may face workload imbalance among MDSs. Besides, when namespace need to be re-divided, system administrator is involved again.

Hashing-based namespace partitioning removes the issue of unbalanced workloads in static partitioning. In general, path name of file and directory is hashed and then assigned to corresponding servers, i.e., Lazy Hybrid [9]. System can quickly locate the requested metadata utilizing the path name and hashing function. This approach causes tremendous overhead when node is added to or deleted since system should re-calculate the hashing-function and relocate most of the metadata. Traversal of a directory is also inefficient in such design.

Dynamic subtree partitioning [10] is proposed to address the load imbalance problem in static partitioning, i.e., Ceph [11]. The metadata of the whole file system is partitioned by hashing directories near the root of the directory hierarchy, each of which is undertaken by a node in a MDS cluster. By migrating heavily loaded metadata automatically and overlapping popular parts, the load among different MDSs can be balanced dynamically. However, because of the existence of overlapped metadata, the maintenance of the consistency between different MDSs becomes more significant and critical, and consequently, the system becomes very complex and costly to realize and execute. Moreover, balancing load will cause metadata redistribution when the user access pattern or the MDS set changes. This results in additional overhead.

Hierarchical Bloom-filter Array (HBA) is an approach based on bloom filter [12]. In HBA, each metadata server constructs a bloom filter to store the path name of metadata that it hosts. Exploiting the temporal access locality, HBA uses another bloom filter to store some frequently accessed path. Although bloom filter is space-efficient, it returns a probabilistic answer and cannot guarantee the location of a file.

In the third class of metadata management, there is no dedicated metadata sever at all. GlusterFS [13] is a representative of this class, which replaces the metadata module, i.e. MDS, with an elastic hash algorithm. That is, there is in fact no explicit MDS, and client is in charge of locating data according to file's absolute path. Therefore, the bottleneck and single point of failure issues in server side is eliminated too, and high scalability and parallelism is simply achieved. However, such approach also has trouble when traversing a directory and maintaining the consistency of namespace. Additionally, lack of specialized MDS causes more workloads and responsibility at client nodes.

As for the replication of metadata, the Hadoop extension by MapR Inc. [14] is the only existing work to the best of our knowledge. In this system, metadata is replicated like common data to achieve for high availability and better performance. Our work differs from the work of MapR Inc. in the overall system architecture and the management method of replication management.

3 The Design and Implementation of MRFS

3.1 Overview of MRFS

MRFS (Metadata Replication File system) is mainly composed of four components: Metadata Server (MDS), Namespace Server (NS), Client and Chunk Server (CS). Several MDSs distribute in different geography locations, and store actual file metadata. On the other side, there is only one single NS maintaining a global namespace and managing the whole file system. Clients connect and conduct operations to their primary MDSs. Chunk servers are nodes that provide storage for file data.

MRFS aims to provide low latency for a majority of client requests of metadata under different scenarios. It is assumed that clients have higher interest in metadata that they created, which means that requests are more probably satisfied in clients' primary MDS. A small portion of requests cannot be handled locally and therefore primary MDS inquires NS for the location of that metadata and then forward the request to corresponding MDS containing the requested metadata. In case of special states, like hotspots or flash crowds, replication is used to reduce such across-MDS interactions noticeably, and in consequence, improve the overall performance and load balance.

Taking advantage of the locality of metadata and client behavior, such design avoids the inherently existing drawbacks of other methods like static subtree partitioning and hashing design, and provides low-latency access for clients in most situations. Moreover, by means of replication, MRFS addresses the work balance issue.

In the rest of this section, we present the details of the design and implementation of MRFS. The architecture of MRFS is shown in Fig 1, which is the basis of the following description.

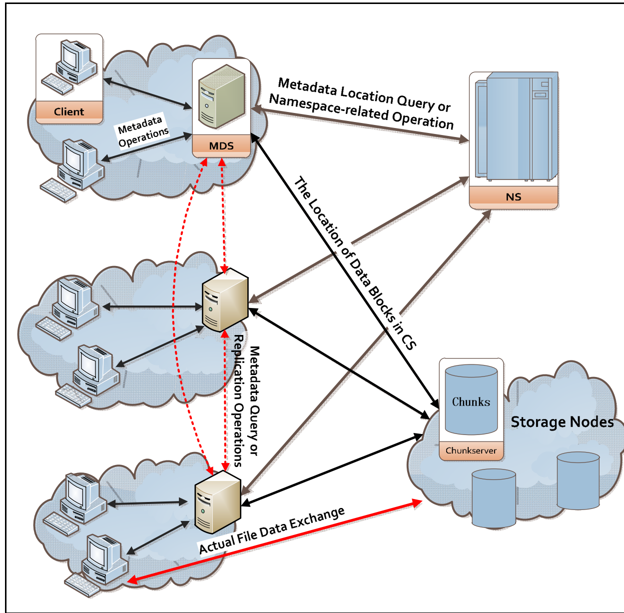

Fig. 1. The architecture of MRFS

3.2 The Client Module of MRFS

The client of MRFS is built on FUSE [15], a loadable kernel module that provides library API to create a file system in user space. We implement interfaces that are essential to build a practical file system. These interfaces are listed in table 1.

Table 1. Implemented interfaces in MRFS

Name	Description
fsinit	initiating process for file system
getattr	retrieve attributes of file or directory
create	create a file
unlink	remove a file
mkdir	create a directory
rmdir	remove a directory
read	read a file
write	write a file
readdir	read a directory, i.e., *ls* operation
chmod	change modes of file or directory

When initiated, client connects and registers with its primary MDS, which has a lowest latency for client requests. Client communicates with its primary MDS through a long-lived TCP connection. Such design eliminates the process of figuring out the

closest MDS and initializing a connection, like looking up the routing table and then connecting to that MDS, upon receiving any request from FUSE.

After client process is mounted on a certain directory, all operations conducted in this directory are transmitted to the client process through FUSE module. Client identifies the type of command and then sends a corresponding message directly to the primary MDS via the connection already established. It should be noted that FUSE automatically call *getattr* function to acquire the file attribute for existence and access privilege check. Only when it returns with a success code, actual operation can be executed onwards.

Two types of operations should be considered individually. First type is only-metadata-involved operations like create and unlink. The primary MDS is in charge of handling the whole workflow. Such operations come to an end when client receives the execution result and/or the requested metadata information.

The second type is operation that involves actual data of files. Client retrieves the storage location of data blocks from MDS at first, and then interacts with corresponding storage nodes for real data read/write.

3.3 The Namespace Server of MRFS

The namespace server (NS) maintains a global and consistent directory tree in main memory. All metadata servers connect to NS when starting up, and forward all namespace-related operations, i.e., creating or removing a file, to NS. Since NS uses a single-thread model, we don't need to worry about the annoying consistency issues. By novelly separating the namespace from traditional metadata service, unlike other distributed metadata management, MRFS is able to provide much more efficient and straightforward response for directory query, i.e., ls operation.

Additionally, NS stores the mapping between metadata entry of each file and its primary MDS, identified by the absolute path name of files. Therefore, MDS can acquire the location of every metadata entry along with its path name. Though absolute path may cause extra memory overhead, the overall system is benefited from its faster locating and traversal of the directory. As a workaround, we can use the prefix-compression algorithm on path name to reduce memory usage, at the cost of longer delay of processing.

To realize replication mechanism, NS also records the replicas information. With this, NS provides more flexibility and useful functionalities, like restricting the total number of replicas and computing the popularity of entries.

3.4 The Metadata Server of MRFS

The MDS module in MooseFS is originally designed as a central node that bundles metadata and namespace service together. So we extend it to support distributed metadata model. There are multiple metadata servers in system and each of them is in charge of managing a part of the metadata. The metadata of the whole file system is divided into non-overlapping parts in accordance with the client's creation operation. Fig 2 and Fig 3 illustrate the creation process and the consequent namespace.

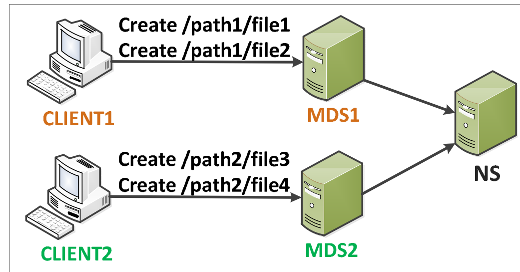

Fig. 2. Clients create different files

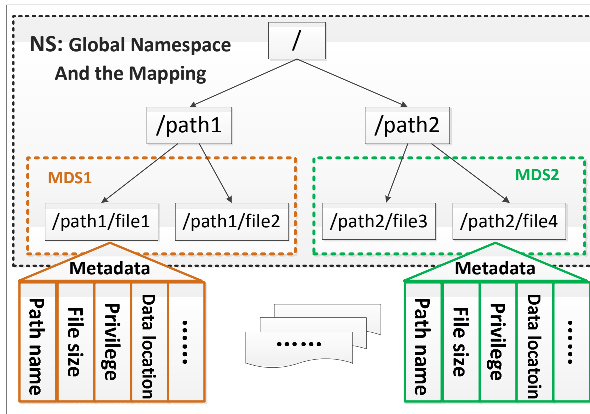

Fig. 3. Metadata construction in server side

Client1 and client2 create files respectively and send command to their own primary MDSs. MDS stores that metadata entry is called Host MDS, i.e., MDS1 is the Host MDS of */path/file1*. To complete the command, MDS should inform NS of the creation operation, so that new files are added into the namespace. As a result, namespace and file metadata are stored in NS and MDS separately. For MDS, it constructs a hash table in memory to store all local file metadata, so as to accelerate the query speed.

Each MDS serves multiple clients simultaneously. When a request from client is coming, primary MDS scans the hash table to check whether the requested entry exists locally. If so, MDS returns directly; or else, it should forward the query to NS for a global query. If the path exists in other MDS, NS will return its location. Then primary MDS connects to the Host MDS (for the first time of connection) and queries for the actual metadata. This procedure introduces an extra RTT, but such situation is supposed to be rare.

In case that there are hotspot issues or flash crowds, we implement the replication mechanism among MDSs. It uses a server-initiated model. Host MDS pushes the copy of popular metadata entries to other MDSs that have queried the metadata beyond a configurable threshold in a specific time interval. Replicas are distributed in different MDSs to improve the workload balance for servers and a low latency for

clients. When creating a new replica, Host MDS keeps track of the replica MDS for subsequent updates. To simplify the design, metadata updates can only be executed in the Host MDS. Whenever the metadata is modified, the updates will be pushed to all available replicas by Host MDS.

MRFS removes the stale replicas automatically to avoid unlimited increase of replicas. Each MDS maintains the visit information for replicas. At intervals of a configurable period, each MDS scans all existing replicas and removes those that are old enough and under the deletion threshold. To guarantee the fairness and decrease the impact of history information, the concept of decay is applied in our design. Each decay period the history visit count is decayed at a rate, whose value can be set according to different requirements.

At last, all processes in server side are transparent to client. The only thing clients concern about is the execution result or the returned metadata information.

4 Experiments and Results

4.1 Experiment Setup

We deploy four machines as MDSs, each of which is with 1G main memory and running Ubuntu 12.04 Server. Along with each MDS, there is a client running at the same node and connecting to the MDS process. Besides, the NS is deployed at another node with 8G main memory and running Ubuntu 12.10 Server. The local-disk file system at each node is ext4.

4.2 Experiment Results

We use four different metrics to measure the performance of metadata service. Firstly, we measure the memory overhead of MDS and NS without replicas. Then we create replicas in one MDS on purpose to measure the replica's impact on memory usage. Secondly, the execution time of creation in different situation is measured. At last, we use NumPy [16] and Python to simulate the visit pattern of web applications. By this, we can measure the efficiency of replicas and the performance enhancement it brings.

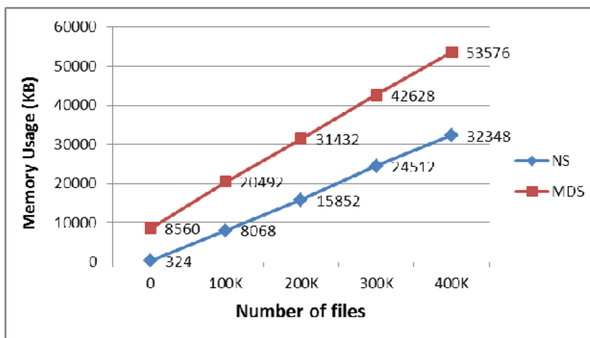

Fig. 4. Memory usage of NS and MDS process changes as the number of created files increases

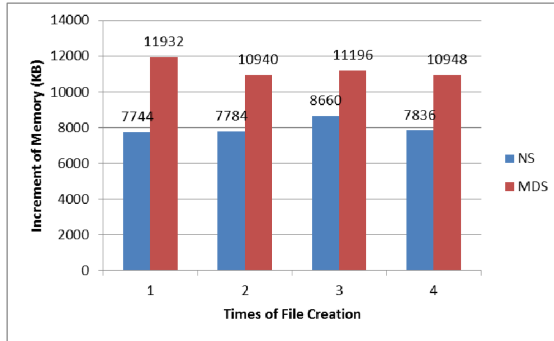

Fig. 5. Increment of memory usage of NS and MDS process in each creation operation

Fig 4 and Fig 5 show the memory usage and increment in NS and MDS along with the increasing of the number of created files. Obviously, the memory usage increase linearly with the file count. Each entry in MDS costs about 110 bytes, while that of NS is about 30% less. This can be explained because NS doesn't store the file information but the directory tree and mapping. Although MDS may take up more memory, in real environment, there are multiple MDS in different locations, workload will be distributed among them and MDS is unlikely to have a bottleneck in memory. It should be clarified that the reason why MDS takes much more memory than NS is that process allocates the memory to hash table in advance.

We run another experiment with two MDS and a NS to demonstrate the impact of replicas. Initially, we create 100K files in MDS2, and then make MDS1 create all files' replica locally. The *dmap* command is used to monitor the memory usage of each process in different stages. Fig 6 shows some features of MRFS. The creation in one MDS won't affect other MDS, which means MDS is able to work individually. The replica creation will introduce extra memory overhead in all three machines. NS increases a tiny amount of memory as it only keeps track of the replica MDS's ip for further use. As the Replica MDS, MDS1 increases about 85% of the primary metadata copy in MDS2. This is because Replica MDS doesn't store the information of remote visit but local visit. As the Primary MDS, MDS2 need to record the replica MDS's information, and therefore its memory usage increases about 20% compared to the original.

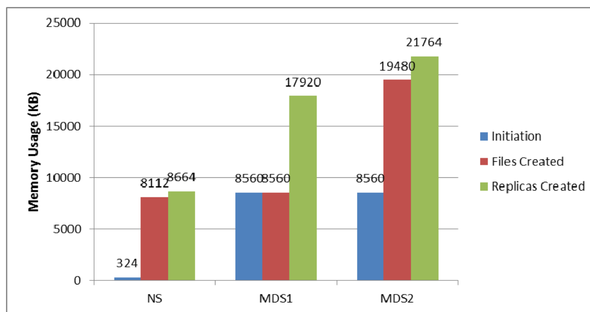

Fig. 6. Comparison of memory usage before and after creating replicas

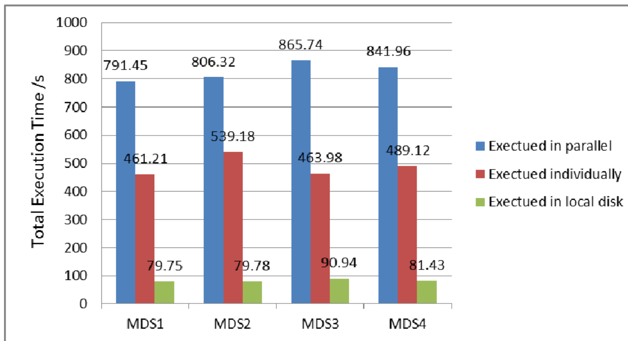

Fig. 7. Total execution time of creating 100K files in four MDS

Each MDS applies the creation of 100K files for three times. More precisely, creation is firstly operated in parallel and then individually and lastly executed in local file system. Fig 7 shows that local file system provides the best performance with the total execution time of 80 seconds. For MRFS, due to the network latency, the elapsed time is about 5 times slower than local file system. Averagely, each creation operation costs about 5ms. When executed in parallel, the average execution time is a little longer, for the reason that each MDS should interchange with NS to finish the operation and the single-thread design of NS limits the throughput. Fortunately, all metadata and namespace information are stored in main memory and this inherent advantage leads to a fast processing. Therefore, network latency takes up most of the elapsed time, and even in parallel mode, the average time of 8ms for each creation is efficient enough for a distributed file system. This result meets our design expectation.

To measure the effectiveness of replication, we firstly create 500 disjoint files in each MDS, and then use Python script and the NumPy library to simulate the access pattern of metadata. Each Client executes 40K access operations through their Primary MDS, and every operation is carried out at a time interval of 10ms.

The local/remote access ratio is set in the script, i.e., ratio=0.2 means that local access takes up 20% of all access while remote access of other MDS takes up 80%. All accesses conform to the Pareto Distribution, which is a power-like distribution and can be used as a model for many read-world problems [17][18]. For MRFS, that means only a small part of metadata is involved with a majority of accesses. Another parameter is the threshold of replica creation which can be configured in MDS module. In our experiment, threshold parameter is set as 5, 10 and 20. We also count the number of created replicas, and calculate the average hit number of all replicas. This metric can show the overall efficiency of the metadata replication.

We can calculate the hit ratio of generated replicas among all accesses that could not be satisfied by local metadata. The hit rates and the number of created replicas are plotted in Fig 8 and Fig 9 respectively. Firstly, we can see obviously the hit ratio is decreasing as the threshold number increases. This is simply because fewer replicas are generated and more requests are forward to remote MDS. In the worst case (ratio=0.8 and threshold=20), the hit ratio drops to 48.44%. However, since the total remote accesses account for 20% of all accesses, the actual forward operations take up only 10.29%. Secondly, with the value of ratio increases, which reflects more

requests are handled locally, the percentage of replica hit falls on the contrary. The reason is that with the same value of threshold, fewer remote accesses will lead to fewer replicas according to the feature of Pareto distribution, and consequently a greater number of requests will be forward to other MDS.

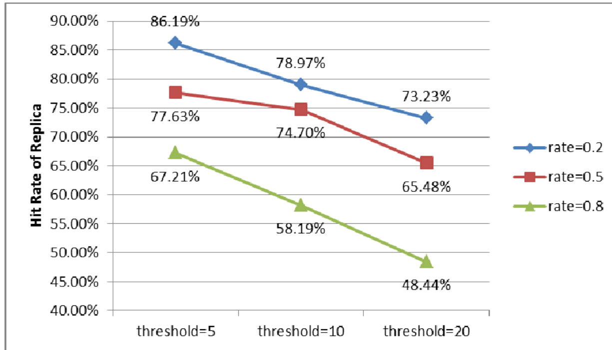

Fig. 8. Hit rate of replicas with various parameters

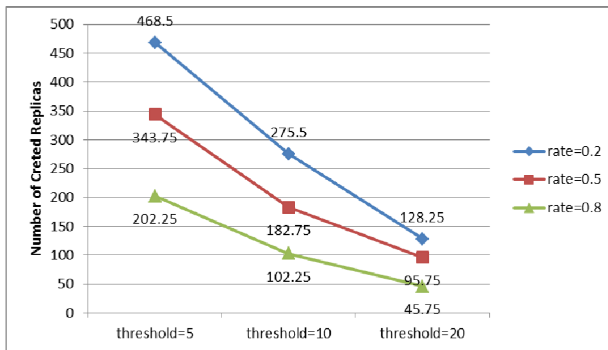

Fig. 9. Number of generated replicas with various parameters

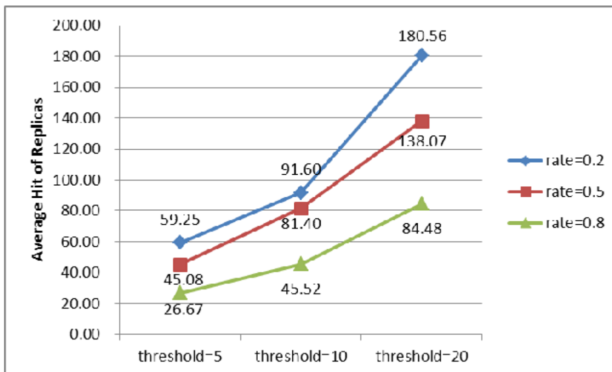

Fig. 10. Average hit count of replicas with various parameters

Besides the hit ratio of replicas, we use the average hit count to measure the efficiency of replicas. As can be observed in Fig 10, one replica serves more visit requests while the total number of replicas declines. This can be explained by the basic properties of Pareto distribution, that is a majority of access only involve with a small amount of metadata. Therefore, the most popular part of replicas will take over more requests than others.

From the discussions and comparisons above, we can see that MRFS performs efficiently and effectively in metadata service in various scenarios. And the replication mechanism largely reduces the cross-MDS visit at the cost of affordable memory overhead. Besides, we can take advantage of the flexible configuration to achieve the balance between the overall performance and memory usage.

5 Conclusion and Future Work

Distributed file system plays a key role in distributed computing, especially in cloud computing systems with high requirement of storage volumes and performance. High scalable and effective metadata service is still a challenging issue in the design and implementation of distributed file systems. We design and implement a real distributed file system MRFS with novel metadata management, which takes advantage of two-tiered architecture and separates the metadata and namespace service. To reduce the latency and alleviate the hotspot issues and flash crowds, an efficient and flexible replication mechanism is implemented as well. Experiments show that the file system can process a majority of file operations with low latency. Moreover, the distribution of metadata service provides a higher scalability and efficiently serves clients scattered at different places. Last but not least, replicas bring substantial performance improvement at little expense of memory loads.

In future, we will improve our system in several ways. First, high availability of metadata will be introduced and implemented, which will make the system more robust. Second, we will consider new approach like prefix-compression algorithm to reduce the memory overhead in namespace server. Third, new replica placement strategies will be considered to improve the overall efficiency while cutting down more memory usage.

Acknowledgement. This research is partially supported by National Natural Science Foundation of China (No. 61379157), Guangdong Natural Science Foundation (No. S2012010010670), and Pearl River Nova Program of Guangzhou (No. 2011J2200088)

References

1. Leung, A.W., Shao, M., Bisson, T., Pasupathy, S., Miller, E.L.: Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. In: FAST, vol. 9, pp. 153–166 (2009)
2. Roselli, D.S., Lorch, J.R., Anderson, T.E.: A Comparison of File System Workloads. In: USENIX Annual Technical Conference, General Track, pp. 41–54 (2000)

3. Traeger, A., Zadok, E., Joukov, N., Wright, C.P.: A nine year study of file system and storage benchmarking. ACM Transactions on Storage (TOS) 4(2), 5 (2008)
4. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. IEEE (2010)
5. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. In: ACM SIGOPS Operating Systems Review, vol. 37(5), pp. 29–43. ACM (2003)
6. MooseFS, `http://www.moosefs.org`
7. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on Computers 39(4), 447–459 (1990)
8. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM Transactions on Computer Systems (TOCS) 10(1), 26–52 (1992)
9. Brandt, S.A., Miller, E.L., Long, D.D., Xue, L.: Efficient metadata management in large distributed storage systems. In: 2013 IEEE 10th International Conference on Mobile Ad-Hoc and Sensor Systems, pp. 290–290 (2003)
10. Weil, S.A., Brandt, S.A., Miller, E.L., Maltzahn, C.: CRUSH: Controlled, scalable, decentralized placement of replicated data. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 122. ACM (2006)
11. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D., Maltzahn, C.,, C.: A scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. USENIX Association (2006)
12. Zhu, Y., Jiang, H., Wang, J.: Hierarchical bloom filter arrays (hba): A novel, scalable metadata management system for large cluster-based storage. In: 2004 IEEE International Conference on Cluster Computing, pp. 165–174 (2004)
13. GlusterFS, `http://www.gluster.org`
14. MapR, `http://www.mapr.com`
15. FUSE, `http://fuse.sourceforge.net`
16. NumPy, `http://www.numpy.org`
17. Arnold, B.C.: Pareto distribution. John Wiley & Sons, Inc. (1985)
18. Reed, W.J.: The Pareto, Zipf and other power laws. Economics Letters 74(1) (2001)