Efficient Metadata Management for Cloud Computing applications

Abhishek Verma Shivaram Venkataraman Matthew Caesar Roy Campbell
{verma7, venkata4, caesar, rhc} @illinois.edu
University of Illinois at Urbana-Champaign

Abstract

Cloud computing applications require a scalable, elastic and fault tolerant storage system. In this paper, we describe how metadata management can be improved for a file system built for large scale data-intensive applications. We implement Ring File System (RFS), that uses a single hop Distributed Hash Table, found in peer-to-peer systems, to manage its metadata and a traditional client server model for managing the actual data. Our solution does not have a single point of failure, since the metadata is replicated and the number of files that can be stored and the throughput of metadata operations scales linearly with the number of servers. We compare against two open source implementations of Google File System (GFS): HDFS and KFS and show that our prototype performs better in terms of fault tolerance, scalability and throughput.

1. Introduction

The phenomenal growth of web services in the past decade has resulted in many Internet companies having the requirement of performing large scale data analysis like indexing the contents of the billions of websites or analyzing terabytes of traffic logs to mine usage patterns. A study into the economics of computing [1] published in 2003, revealed that due to relatively higher cost of the transferring data across the network, the most efficient computing model is to move computation near the data. As a result, several large scale, distributed, data-intensive applications [2, 3] are used today to analyze data stored in large datacenters.

The growing size of the datacenter also means that hardware failures occur more frequently and that applications need to be designed to tolerate such failures. A recent presentation about a typical Google datacenter reported that up to 5% of disk drives fail each year and that every server restarts at least twice a year due to software or hardware issues [4]. With the size of digital data doubling every 18 months [5], it is also essential that applications are designed to scale and meet the growing demands.

Distributed data storage has been identified as one of the challenges in cloud computing [6]. An efficient distributed file system needs to:

1. provide large bandwidth for data access from multiple concurrent jobs
2. operate reliably amidst hardware failures
3. be able to scale to many millions or billions of files and thousands of machines

The Google File System (GFS) [7] was proposed to meet the above requirements and has since been cloned in open source projects like Hadoop Distributed File System (HDFS) [8] and Kosmos File System (KFS) [9] that are used by companies like Yahoo, Facebook, Amazon, Baidu, etc. The GFS architecture comprises of a single GFS *master* server which stores the metadata of the file system and multiple slaves known as *chunkservers* which store the data. Files are divided into chunks (usually 64 MB in size) and the GFS master manages the placement and data-layout among the various chunkservers. The GFS master also stores the metadata like filenames, size, directory structure and information about the location and placement of data in memory. One of the direct implications of this design is that the size of metadata is limited by the memory available at the GFS master. This architecture was picked for its simplicity and works well for hundreds of terabytes with few millions of files [10]. With storage requirements growing to petabytes, there is a need for distributing the metadata storage to more than one server.

Clients typically communicate with a GFS master only while opening a file to find out the location of the data and then directly communicate with the chunkservers to reduce the load on the single master. A typical GFS master is capable of handling a few thousand operations per second [10] but when massively parallel applications like a MapReduce [2] job with many thousand mappers need to open a number of files, the GFS master becomes overloaded. As datacenters grow to accomodate many thousands of machines in one location, distributing the metadata operations among multiple servers would be necessary to increase the throughput. Though the probability of a single server failing in a datacenter is low and the GFS master is continuously monitored, it still remains a single point of failure for the system. Having multiple servers to handle failure would increase the overall reliability of the system and reduce the downtime visible to clients.

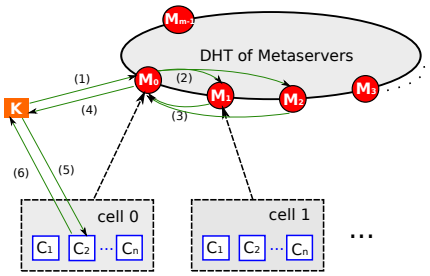

Figure 1. Architecture of RingFS

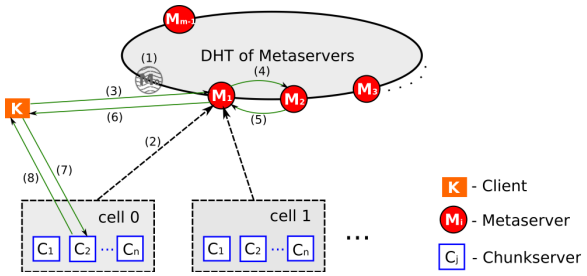

Figure 2. Tolerating metaserver failures

Handling metadata operations efficiently is an important aspect of the filesystem as they constitute up to half of file system workloads [11]. While I/O bandwidth available for a distributed file system can be increased by adding more data storage servers, scaling metadata management involves dealing with consistency issues across replicated servers.

Motivated by the above limitations, our goal is to design a distributed file system for large scale data-intensive applications that is fault tolerant and scalable while ensuring a high throughput for metadata operations from multiple clients. We propose RingFS, a filesystem where the metadata is distributed among multiple replicas connected using a Distributed Hash Table (DHT). Metadata for all the files in a directory is stored at one primary server, whose location is determined by computing a hash of the directory name, and then replicated to its successors.

The major contributions of our work include:

1. Rethinking the design of metadata storage to provide fault tolerance, improved throughput and increased scalability for the file system.
2. Studying the impact of the proposed design through a mathematical analysis and simulations
3. Implementing and deploying the file system on a 16-node cluster and comparison with HDFS and KFS.

The rest of this paper is organized as follows: Section 2 gives a background of the architecture of the existing distributed file system and its limitations. We describe the design of our system in Section 3 and analyze its implications in Section 4. We then demonstrate the scalability and fault tolerance of our design through simulations followed by implementation results in Section 5. We discuss possible future work and conclude with Section 6.

2. Related Work

Metadata management has been implemented in systems like NFS, AFS by statically partitioning the directory hierarchy to different servers. This, however, requires an administrator to manually assign subtrees to each server but enables clients to easily know which servers have the metadata for a give file name. Techniques of hashing a file name or the parent directory name to locate a server have been previously discussed in file systems like Vesta [12] and Lustre [13]. Ceph [14], a petabyte scale file system, uses a dynamic metadata distribution scheme where subtrees are migrated when the load on a server increases. Hashing schemes have been found to be inefficient while trying to satisfy POSIX directory access semantics as this would involve contacting more than one server. However studies have shown that most cloud computing applications do not require strict POSIX semantics [7] and with efficient caching of metadata on the clients, the performance overhead can be overcome. Filesystems like PAST [15] and CFS [16] have been built on top of DHTs like Pastry and Chord but concentrate on storage management in a peer to peer system with immutable files. A more exhaustive survey of peer-to-peer storage techniques for distributed file systems can be found here [17].

3. Design

Our architecture consists of three types of nodes: *metaservers*, *chunkservers* and *clients* as shown in the Figure 1. The metaservers store the metadata of the file system whereas the chunkservers store the actual contents of the file. Every metaserver has information about the locations of all the other metaservers in the file system. Thus, the metaservers are organized in a single hop Distributed Hash Table (DHT). Each metaserver has an identifier which is obtained by hashing its *MAC* address.

Chunkservers are grouped into multiple cells and each cell communicates with a single metaserver. This grouping can be performed in two ways. The chunkserver can compute a hash of its *MAC* address and connect to the metaserver that is its successor in the DHT. This makes the system more self adaptive since the file system is symmetric with respect to each metaserver. The other way is to configure each chunkserver to connect to a particular metaserver alone. This gives more control over the mapping of chunkservers to metaservers and can be useful in configuring geographically distributed cells each having its own metaserver.

The clients distribute the metadata for the files and directories over the DHT by computing a hash of the parent path present in the file operation. Using the parent path implies that the metadata for all the files in a given directory is present at the same metaserver. This makes listing the

contents of a directory efficient and is commonly used by MapReduce and other cloud computing applications.

3.1 Normal operation

We demonstrate the steps involved in the creation of a file, when there are no failures in the system. The sequence of operations shown in Figure 1 are:

1. Client wishes to create a file named */dir1/dir2/filename*. It computes a hash of the parent path, */dir1/dir2*, to determine that it has to contact metaserver M_0 for this file operation.
2. Client issues a create request to this metaserver which adds a record to its *metatable* and allocates space for the file in Cell 0.
3. Before returning the response back to the client, M_0 sends a replication request to r of its successors, M_1, M_2... in the DHT to perform the same operation on their *replica_metatable*.
4. All of the successor metaservers send replies to M_0. Synchronous replication is necessary to ensure consistency in the event of failures of metaservers.
5. M_0 sends back the response to the client.
6. Client then contacts the chunkserver for the actual file contents.
7. Chunkserver finally responds with the file.

Thus, in all r metadata Remote Procedure Calls (RPCs) are needed for a write operation. If multiple clients try to create a file or write to the same file, consistency is ensured by the fact that these mutable operations are serialized at the primary metaserver for that file.

The read operation is similarly performed by using the hash of the parent path to determine the metaserver to contact. This metaserver directly replies with the metadata information of the file and where the chunks for the file are located. The client then communicates directly with the chunkservers to read the contents of the file. Thus, read operations need a single metadata RPC.

3.2 Failure and Recovery

Let us now consider a case now where metaserver M_0 has failed . The chunkservers in cell 0, detect the failure through heartbeat messages and connect to the next server M_1 in the DHT. When a client wishes to create a file its connection is now handled by M_1 in place of M_0. As most of the server failures in a datacenter notify a system administrator, we assume that the failed server will come back shortly and hence replicate the metadata to only $r-1$ servers for this request. M_1 also allocates space for the file in cell 0 and manages the layout and replication for the chunkservers in cell 0.

Once M_0 recovers, it sends a request to its neighboring metaservers M_1, M_2 ... to obtain the latest version of the metadata. On receipt of this request, M_1 sends the metadata which belongs to M_0 and also closes the connection with the chunkservers in cell 0. The chunkservers now reconnect to M_0 which takes over the layout management for this cell and verifies the file chunks based on the latest metadata version obtained.

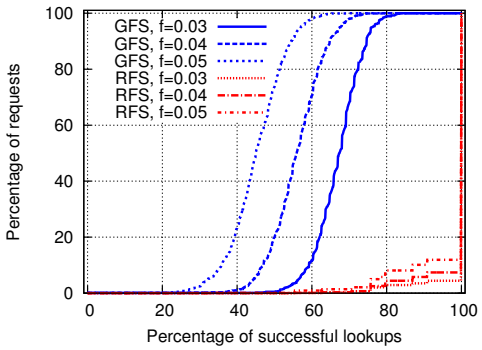

Figure 3. CDF of number of successful lookups for different failure probabilities

4. Analysis

In this section, we present a mathematical analysis comparing the design of GFS and X with respect to the fault tolerance, scalability, throughput and overhead followed by a failure analysis.

4.1 Design Analysis

Let the total number of machines in the system be n. In GFS, there is exactly 1 metaserver and the remaining $n-1$ machines are chunkservers that store the actual data. Since there is only one metaserver, the metadata is not replicated and the filesystem cannot survive the crash of the metaserver.

In RFS, we have m metaservers that distribute the metadata r times. RFS can thus survive the crash of $r-1$ metaservers. Although a single Remote Procedure Call (RPC) is enough for the lookup using a hash of the path, r RPCs are needed for the creation of the file, since the metadata has to be replicated to r other servers. Since m metaservers can handle the read operations, the read metadata throughput is m times that of GFS. Similarly, the write metadata throughput is m/r times that of GFS, since it is distributed over m metaservers, but replicated r times. This analysis is summarized in Table 1.

4.2 Failure Analysis

Failures are assumed to be independent. This assumption is reasonable because we have only tens of metaservers and they are distributed across racks and potentially different clusters. We ignore the failure of chunkservers in this analysis since it has the same effect on both the designs and simplifies our analysis. Let $f = 1/MTBF$ be the probability that the meta server fails in a given time, and let R_{GFS} be

Metric	GFS	RFS
Metaserver failures that can be tolerated	0	$r-1$
RPCs required for a read	1	1
RPCs required for a write	1	r
Metadata records	X	$X \cdot m/r$
Metadata throughput for reads	X	$X \cdot x$
Metadata throughput for writes	X	$X \cdot y$

Table 1. Analytical comparison of GFS and RFS

the time required to recover it. The file system is unavailable for $R_{GFS} \cdot f$ of the time. For example, if the metaserver fails once a month and it takes 6 hours for it to recover, then the file system availability is 99.18%.

Let m be the number of metaservers in our system, r be the number of times the metadata is replicated, f be the probability that a given server fails in a given time t and R_{RFS} be the time required to recover it. Note that R_{RFS} will be roughly equal to $r \cdot R_{GFS}/n$, since the recovery time of a metaserver is proportional to the amount of metadata stored on it and we assume that the metadata is replicated r times. The probability that any r consecutive metaservers in the ring go down is $mf^r(1-f)^{m-r}$. If we have $m = 10$ metaservers, $r = 3$ copies of the metadata and $f = 0.1$ per 3 days, then this probability is 0.47%. However, a portion of our file system is unavailable if and only if all the replicated metaservers go down within the recovery time of each other. This happens with a probability of $F_{RFS} = m \cdot f \cdot \left(\frac{f \cdot R_{RFS}}{t}\right)^{r-1} \cdot (1-f)^{m-r}$, assuming that the failures are equally distributed over time. The file system is unavailable for $F_{RFS} \cdot R_{RFS}$ of the time. Continuing with the example, the recovery time would be 1.8 hours and the availability is 99.9994%.

5. Experiments

In this section, we present experimental results obtained from our prototype implementation of RingFS. Our implementation is based on the KFS implementation and modified the metadata management data structures and added the ability for metaservers to recover from failures by communicating with its replicas. To study the behavior on large networks of nodes, we also implemented a simulation environment.

All experiments were performed on sixteen 8-core HP DL160 (Intel Xeon 2.66GHz CPUs) with 16GB of main memory, running CentOS 5.4. The MapReduce implementation used was Hadoop 0.20.1 and was executed using Sun's Java SDK 1.6.0. We compare our results against Hadoop Distributed File System (HDFS) that accompanied the Hadoop 0.20.1 release and Kosmos File system (KFS) 0.4. A single server is configured as the metaserver and the other 15 nodes run the chunkservers. RFS is configured with 3 metaservers and 5 chunkservers connecting to each of them. The metadata is replicated three times.

5.1 Simulation

Fault tolerance of a design is difficult to measure without a large scale deployment. Hence, we chose to model the failures that occur in datacenters using a discrete iterative simulation. Each metaserver is assumed to have a constant and independent failure probability.

The results show that RFS has better fault tolerance than the single master (GFS) design. In the case of GFS, if the metaserver fails, the whole filesystem is unavailable and the number of successful lookups is 0 till it recovers after some time. In RFS, we configure 10 metaservers and each fails independently. The metadata is replicated on the two successor metaservers. Only a part of the filesystem is unavailable only when three successive metaservers fail. Figure 3 shows plot of the CDF of the number of successful lookups for GFS and RFS for different probabilities of failure. As the failure probability increases, the number of successful lookups decreases. Less than 10% of the lookups fail in RFS in all the cases.

5.2 Fault Tolerance

The second experiment demonstrates the fault tolerance of our implementation. A client sends 150 metadata operations per second and the number of successful operations is plotted over time for GFS, KFS and RFS in Figure 6. HDFS achieves a steady state throughput, but when the metaserver is killed, the complete filesystem become unavailable. Around $t = 110s$, the metaserver is restarted and it recovers from its checkpointed state and replays the logs of operations that couldn't be checkpointed. The spike during the recovery happens because the metaserver buffers the requests till it is recovering and batches them together. A similar trend is observed in the case of KFS, in which we kill the metaserver at $t = 70s$ and restart it at $t = 140s$.

For testing the fault tolerance of RFS, we kill one of the three metaservers at $t = 20s$ and it does not lead to any decline in the throughput of successful operations. At $t = 30s$, we kill another metaserver, leaving just one metaserver leading to a drop in the throughput. At $t = 60s$, we restart the failed metaserver and the throughput stabilizes to its steady state.

5.3 Throughput

The third experiment demonstrates the metadata throughput performance. A multithreaded client is configured to spawn a new thread and perform read and write metadata operations at the appropriate frequency to achieve the target qps. We then measure how many operations complete successfully each second and use this to compute the server's capacity. Figure 4 shows the load graph comparison for HDFS, KFS and RFS. The throughput of RFS is roughly twice that of HDFS and KFS and though the experiment was conducted with 3 metaservers, the speed is slightly lesser due to the replication overhead.

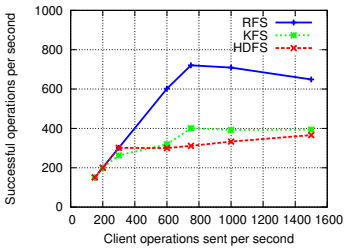

Figure 4. Comparison of throughput under different load conditions

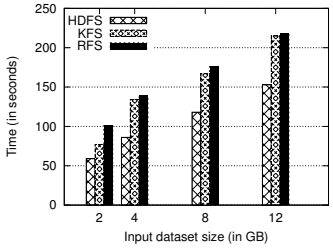

Figure 5. MapReduce application - Wordcount

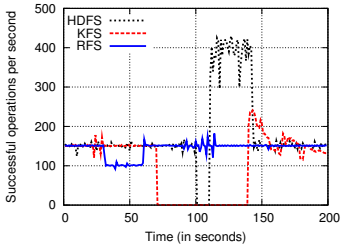

Figure 6. Fault tolerance of HDFS, KFS and RFS

5.4 MapReduce Performance

We ran a simple MapReduce application that counts the number of words on a wikipedia dataset and varied the input dataset size from 2GB to 16GB. We measured the time taken for the job to compute on all three file system and a plot of the same is shown in Figure 5. We observed that for a smaller dataset the overhead of replicating the metadata did increase the time taken to run the job, but on larger datasets the running times were almost the same for KFS and RFS.

6. Conclusion

We presented and evaluated RingFS, a scalable, fault-tolerant and high throughput file system that is well suited for large scale data-intensive applications. RFS can tolerate the failure of multiple metaservers and it can handle a large number of files. We have shown how the idea of using a single hop Distributed Hash Table to manage its metadata from Peer-to-peer systems can be combined together with the traditional client server model for managing the actual data. Our techniques for managing the metadata can be combined with other filesystems.

References

[1] J. Gray, "Distributed computing economics," *Queue*, vol. 6, no. 3, pp. 63–68, 2008.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 59–72.

[4] J. Dean, "Large-Scale Distributed Systems at Google: Current Systems and Future Directions," 2009.

[5] J. Gantz and D. Reinsel, "As the economy contracts, the Digital Universe expands," *IDC Multimedia White Paper*, 2009.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[8] "Hadoop," http://hadoop.apache.org/.

[9] "Kosmos file system," http://kosmosfs.sourceforge.net/.

[10] M. K. McKusick and S. Quinlan, "GFS: Evolution on Fast-forward," *Queue*, vol. 7, no. 7, pp. 10–20, 2009.

[11] D. Roselli, J. Lorch, and T. Anderson, "A comparison of file system workloads," in *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 2000.

[12] P. Corbett and D. Feitelson, "The Vesta parallel file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 3, pp. 225–264, 1996.

[13] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*. Citeseer, 2003.

[14] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[15] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *Proc. HotOS VIII*. Citeseer, 2001, pp. 75–80.

[16] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 202–215, 2001.

[17] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell, "A survey of peer-to-peer storage techniques for distributed file systems," in *ITCC*, vol. 5. Citeseer, pp. 205–213.