A Novel Scalable Architecture of Cloud Storage System for Small Files based on P2P

ZHANG Qi-fei, PAN Xue-zeng, SHEN Yan
College of Computer Science and Technology,
Zhejiang University,
Hangzhou 310027, China;
zhangfeezju@gmail.com

LI Wen-juan,
College of Qianjiang,
Hangzhou Normal University
Hangzhou, 310036, China
liellie@163.com

Abstract—**Scalability and Latency are the two important performance indicators for the distributed file system, and Google and Apache have achieved a great success with GFS and HDFS when operating big files, but the latency is too long when reading and writing small-size files, because the concurrent I/O can't work for small files, besides the master node is difficult to extend in the cloud storage system with Master/Slave structure. In this paper, we propose a distributed cloud storage system based on P2P, where a central route node is introduced to improve the resource query efficiency, so clients can find data using only one message compared with Chord's log(N). The central routing node only stores the status and routing information of all data nodes, which are indexed by the Trie Tree structure, so query time meets the requirement of online query. The data nodes store file's content and file's metadata thus the system is easy to extend because the master node no longer needs to store the metadata. Clients can also cache the routing information, so the read and write time is reduced according to the Locality Principle. Experiments show that the reading and writing time is significantly reduced compared with Hadoop HDFS.**

Keywords-P2P; the Small File; DHT; Chord Routing Algorithm; Cloud Storage System; Trie Tree

I INTRODUCTION

With the development of computational science, the large-scale parallel computers and large-scale parallel storage system are also renewed to meet the computing and storage requirements of various business systems. The development of parallel storage system is increasingly specialized to take as much utilization of underlying hardware as possible. Currently, the parallel storage system with a Master/Slaver structure has great advantage in reading and writing large-size files and the performance advantage is particularly evident when dealing with G-level even T-level files, such as Google's GFS(Google File System) [11] and Apache's Hadoop HDFS (Hadoop Distributed File System) [13]. But the concurrent I/O can't work for small-size files. According to a report from the US National Energy Research Scientific Computing Center in 2007, 99 percent files are smaller than 64MB and 43 percent files are smaller than 64 KB [1] in a file system with 1.3×107 files. Researchers of US Pacific Northwest National Laboratory have also come to the similar conclusion that 94 percent files are smaller than 64MB and 58 percent files are smaller than 64 KB [2] in a file system with 1.2×107 files. The conclusion is also confirmed by the data from climate, astronomy, biology and other fields, and the average size is 61 MB [3] of the 4.5×105 files from global atmospheric circulation patterns and the average size is 1 MB [4] of the 2×107 files from astronomical image and the average size is 190 KB [5] of the 3×107 files from human genome sequence map. So researchers begin to focus on the performance improvement for the small-size file in the distribute environment, and Shaikh [6] et al. presented an implementation of file stuffing on metadata server. The problem is that stuffing increases space pressure of the metadata server when more and more small files are stuffed. Carns [7] et al. implemented the similar stuffing method in PVFS2, and compared with the method in [6], the method in [7] only worked when the metadata server acted as data server at the same time, besides, only files that smaller than stripe size can benefit from their optimization. Hendricks [8] et al. studied the overhead due to the decoupling of clients and servers in

978-0-7695-4844-9/12 $26.00 © 2012 IEEE
DOI 10.1109/ClusterW.2012.27

distributed file system. They presented metadata pre-fetching method to mask the network roundtrip. Since the metadata is very small, the optimization can bring significant performance gains. Kuhn [9] et al. observed that not all applications required complete metadata for small file workloads. They implemented a special directory hint, no metadata to label the directory which contains small files. This method shows significant metadata performance improvements for small files but the directory restriction limits its applying scope to directory level. Mackey [10] et al. proposed the metadata compress method, which can improve the space utilization for metadata, but the method doesn't fundamentally solve the problem and introduces additional overhead simultaneously. The pre-fetching method can reduce the time overhead of a round-trip, but the method will introduce lots of extra time with the number of the small files increasing.

Firstly, we present the current mainstream distributed file system with Master/Slaver structure, and then we analyze its performance bottleneck when dealing with small files. After that, we propose a distribute file system for small file based on P2P in this paper, and we introduce the Trie data structure to store the routing information, which meets the requirements of online query. Clients can also cache the central routing information when the number of small files is very large. Experiments show that the performance of reading and writing small files is significantly improved compared with Hadoop HDFS.

II DISTRIBUTE STORAGE SYSTEM

A. Distribute Storage System with Master/Slaver structure

Currently, the cloud storage system mainly used the Master/Slaver structure, such as Google's GFS, Yahoo's HDFS and Amazon's S3 (Simple Storage Service) [12]. The typical structure is shown as Fig.2.1. The master server maintains directory structure, metadata and the mapping between the file name and block ID, and data blocks are stored on slaver servers. Clients can read and write the data block in parallel for large-size files after obtaining Slavers' IP and blocks' ID in step 3 and step 4. But the concurrent I/O could not work [15] for small files, and when designing a file system for small file with same capacity the master will become the bottleneck.

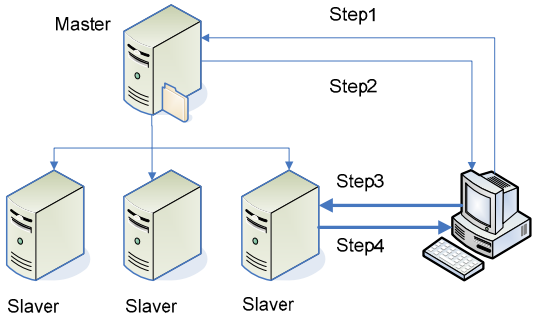


Figure 2.1 Google's GFS file system architecture

B. Distribute Storage System Based on P2P

In this paper we proposed a new distribute storage system architecture for small files with a central routing node, which is shown as Fig. 2.2.

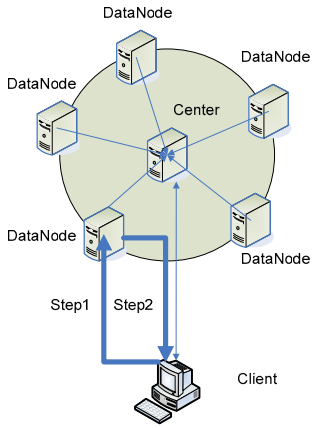


Figure 2.2 Distributed file system based P2P

The center node maintains the routing and status information of all data node, and the item is added or updated when the data node sends a heartbeat packet to the center node. By this way, the center node knows that the data node is online, the other hand, the center node will mark one node offline when the center node don't receive a heartbeat packet within a threshold time, and then the center node will initiate backup process (There are 3 copies for each file). All the data node form a ring, and the system uses current mainstream distribute hash table DHT [16] to distribute the files, which is also adopted by Amazon's Dynamo [14]. DHT enables that each node only needs to process the data falls between the current node and its

precursor node, and the consistence of hash function uniformly spreads data along the ring.

Currently, there are many routing algorithms based on DHT, such as Chord[17], CAN, Kademlia[18] et al., but Chord's complexity is O(log(N)), Kademlia's complexity is O(k), where N is the number of all data nodes and k is the number of the data set that is nearest to value key. Chord is completely decentralized and symmetric, which is different from our system. In our system we introduced a central routing node. The center node stores all nodes' routing and status information, which is showed as Tab.1 that clients can pre-fetch the routing information when the number of the routing table item is not huge; and client can also cache the routing information when the routing data is very large. It's shown as Fig.2.2 that clients can directly read or write the data on the specific data node. But, the Center Node will become the bottleneck of the system with the number of small files increasing. There will be a few problems as follow:

(1) The query time complexity is $O(\log(n))$ when the data set is orderly and the query time complexity is O(n) when the data set is unordered, so it's impossible to handle the high concurrent request.

(2) Too much maintain cost. The time complexity is up to $O(n)$ for inserting and deleting nodes. The number of failed nodes is proportional to the system scale, and the maintenance cost greatly increases the system burden when the system extends.

Table 1 Items of central routing node

Hash Value	Status	IP Address
12ABC23ADFKBCJKDFJAF	On	10.2.1.2
72ABC23ADFKBCJKDFJAF	Off	10.7.1.2
EFAER223ADFK12JKD78A	On	10.9.9.2

III THE INDEX STRUCTURE OF CENTRAL ROUTING NODE BASED ON TRIE TREE

In this paper, a new index structure based on Trie tree is proposed to solve the problem that the query time is too long and the system is difficult to extend. The Trie tree that is known as word search tree or key tree is a tree structure, which is a variant of hash tree, and it is usually used as statistics in search engine system or sorting a large number of strings. Trie is more efficient than Hash Table for Trie tree has less character comparison.

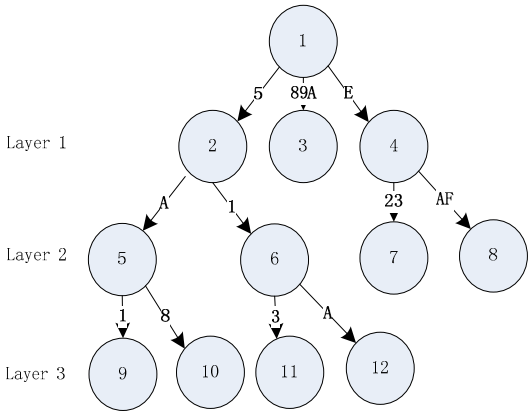


Figure 3.1 The shape of Trie tree

Firstly, several symbols are defined in order to better describe the performance of our system.

Definition 1: the number of all different characters in hash value is m, and the character set in the hash value is denoted by $\mathrm{N} = \{N_1, ...N_m\}$, where $|N| = m$ and symbol "||" denotes the length of the string. The symbol n denotes the number of items in hash table.

Definition 2: the length of the hash value is l, and the string of the hash value is denoted by $T = T[1...l]$, where $|T| = l$.

Definition 3: the length of the search string is l, and the search string is denoted by $P = P[1...l]$, where $|P| = l$.

A. Time Complexity

The Trie tree is built with the hash value shown in Tab.1, and the hash value has a length of 20 Bytes and status flag is one Byte, and IP address is 4 Bytes.

1) Time complexity of construction Trie tree

There is only one node called root node at initial in Trie tree, and it begins to traverse from root node when structuring a Trie tree in the memory, and a new node will be added when the string is not in Trie tree. Average children number of each node is $m/2$, and average height

of Trie tree is $l/2$, so the average number to compare for a new string is less than $(l/2)*(m/2)$, so we can get results as follow:

(1) The query time complexity is $O(n*(\log m)*l/2)$ when the all nodes' children are ordered.

(2) Time complexity is $O(n*(m/2)*(l/2))$ when the all nodes' children are unordered.

2) Time complexity of querying

The substance is that using the Trie tree can reduce the useless comparison. The search begins from root node and only m-times comparisons is needed at most at each layer because that there are m children at most to each node, so the average comparison time is $(m/2)*(l/2)$ for a Trie tree with height of l. We can get results as follow:

(1)Time complexity is $O((\log m)*l/2)$ when the all nodes' children are ordered;

(2)Time complexity is $O((m/2)*(l/2))$ when the all nodes' children are unordered.

3) Time complexity of adding

It's similar to construct a Trie tree for adding a node, and it begins from the root node and searches the node having same prefix with pattern P. The IP address stored in the matching node will be renewed if the pattern P matches all the string along the nodes of the Trie tree, otherwise we add a new node. We can get results as follow:

(1)Time complexity is $O(m*l/2)$ when the all nodes' children are ordered.

(2)Time complexity is $O(\log(m)*l/2)$ when the all nodes' children are unordered.

4) Time complexity of dealing failed node

It's similar to construct a Trie tree for deleting a failed node, and it begins from root node and searches the node having same prefix with pattern P. The node will be deleted if we find a leaf node, otherwise the pattern do not exist in the Trie tree. More, the father node may merge the remaining children after deleting the failed node. It's shown as Fig.3.2 that is the shape after deleting node 12.

The node in cloud computing environment could not leave initiatively unless the node failed, and the node fail is small probability event. Comparing with other P2P scenarios, the probability for data nodes to quit or join the system is very low, so it's not often to delete or add nodes.

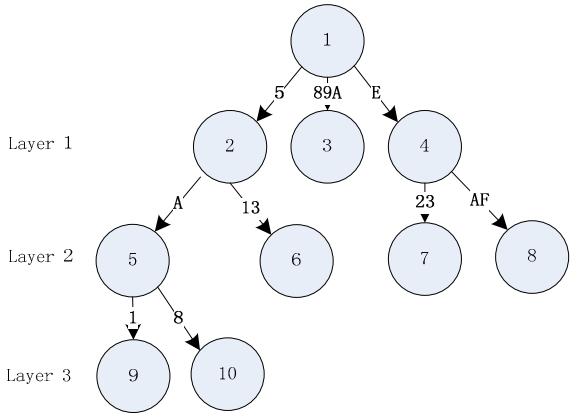


Figure 3.2 Trie tree's shape after deleting node 9

B. Space complexity for pre-fetching

Let l represents the height of the Trie tree, and let L_i represents the leaf number of layer-i and let M_i represents the non-leaf number of layer-i. Due to $|T|=m$, so we can get formula (1).

$$\left\lceil \frac{L_i}{m} \right\rceil \le M_{i-1} \le \left\lceil \frac{L_i}{2} \right\rceil \tag{1}$$

Let M represents the total number of all non-leaf nodes in the Trie tree, so we can get formula (4) according to formula (2) and formula (3), at last we can easily calculate that $M \in [n/m-l, n/2+l]$, so the space complexity is $O(n)$.

$$M = \sum_{i=0}^{l-1} M_i , \tag{2}$$

$$\sum_{i=1}^{l} L_i = n , \tag{3}$$

$$\sum_{i=0}^{l-1} \left\lceil \frac{L_i}{m} \right\rceil \le M \le \sum_{i=0}^{l-1} \left\lceil \frac{L_i}{2} \right\rceil , \tag{4}$$

C. Scalability of Central Server

The query time complexity based on Trie tree is $O(m\ l)$, which meets the requirements of online query. But the memory consumption of Trie tree exceeds the

physical limit of the central node when the system extends, for instance, the number of small files is 1×108 and each leaf takes up 30 Bytes, thus the total memory taken by Trie tree is up to 3GB. So the center node will become the bottleneck when system extends. In this paper we propose a new scalable distributed architecture of the center server, which is shown as Fig.3.3. The server on the top in the Fig. 3.3 is portal to clients, and the server A, B and C respectively construct the Trie tree using the substrings except the characters A, B and C. By this way, the space of Trie tree is distributed to each server, and just one hop routing time is added.

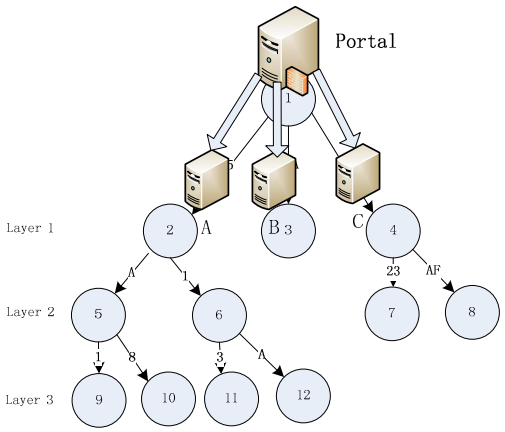


Figure 3.3 Topology after the system extended

D. Fault tolerance of central node

The system uses the double hot backup method, which could ensure system sustainable work when the center node failed. The backup machine will take over the failed one immediately.

E. Cache mechanism of client's routing information

Clients often read or write the same data many times in a period of time according to the Locality Principle, so clients can cache the routing information after obtaining the routing information from the central node.

IV *EXPERIMENTS*

In experiments, there are 10 data nodes and a center node, which is shown as Fig. 4.1. The Switch is D-Link's 10/100/1000 adaptive switch. The data node runs on Windows XP with Intel's dual-core CPU 2.93 and 2GB memory, and the IDE is .NET Framework 3.5 and Visual Studio 2005. The software of node side is written in C# language, and the software of central node side that runs on Ubuntu with kernel 2.6.35 is written in C with compiler gcc 4.4.5.

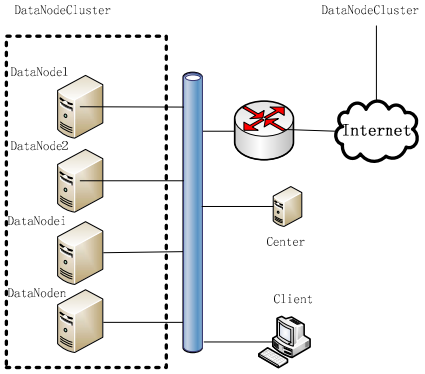


Figure 4.1 Network System topology diagram

A. Comparison of query time

It's shown as Tab. 4.1 that the query time of array structure proportionally increases with the number of small files, but the query time of Trie structure increases little when the number of small files increase from one hundred thousand to one hundred million, the little increased time is mainly because more intermediate nodes are needed to compare. But the worst case is m-times comparisons to each layer, and the time complexity is $O(m\ l)$, which is independent on n and is a constant.

The query time above doesn't include round-trip time in the network, but it's only the time of searching Trie tree.

Table.4.1 Query Time of both algorithms

The number of small files	**Query time of Trie（Ms）**	**Query time of array（Ms）**
100000	0.014	1.05
1000000	0.016	10.3
10000000	0.018	111.2
100000000	0.022	807.1

B. Comparison of memory consumption

It's shown as Tab.4.2 that the memory consumption of two data structures increases with the number of small files, and the memory consumption of Trie structure is 1.6 times more than Array structures. The main reason is that the Trie tree needs additional intermediate nodes and additional pointers to maintain the tree.

Table.4.2 Memory Consumption of both algorithms

The number of small files	Memory consumption of Trie（Bytes）	Memory consumption of Array（Bytes）
100000	3837408	2400000
1000000	40077184	24000000
10000000	398140736	240000000
100000000	17692305640	2400000000

C. Comparison with Hadoop HDFS

Hadoop is the most widely used open source distributed file system, which is similar with Google's GFS. We compared the performance between ours system and Hadoop HDFS. The relationship chart between the file size and time consumption is shown as Fig.4.2 and Fig.4.3, and the time overhead of reading small files with Hadoop is far more than our system. It's because that Hadoop is designed for large-size file in parallel, so Hadoop doesn't take its advantage when reading and writing small-size file, instead, more system overhead is added for parallel design.

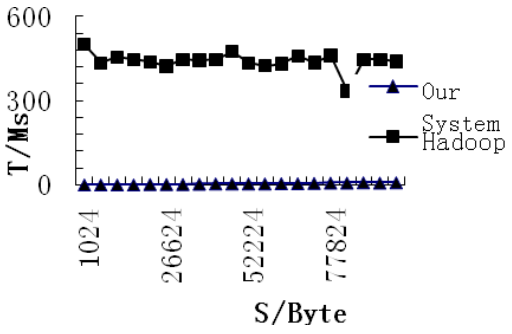
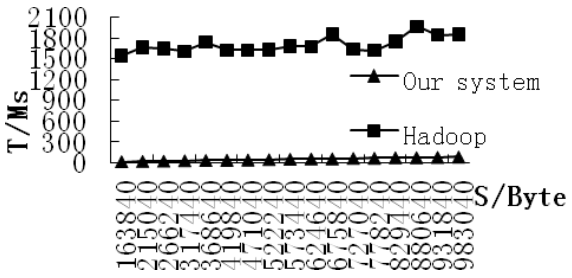


Figure. 4.2 Time cost for reading files of 1K to 100 K comparing with Hadoop



Figuer 4.3 Time cost for reading files of 10 K to 1000 K comparing with Hadoop

V. CONCLUSION

With the rapid development of computational science, all kinds of data including growing proportion small files are generated, and it's too costly for storing the small files on the GFS or HDFS. In this paper we proposes a new distributed storage system for small files based on P2P after analyzing the distributed storage system with Master/Slaver structure. We introduce a central routing node to improve the efficiency of resource discovery, so clients can find data using only one message compared with Chord's log(N). The central routing node stored the status and routing information of all the nodes, which is indexed by the Trie, and query time meets the requirements of online query. The clients can pre-fetch the routing information when the data is not very large, and the clients can cache the information when the number of small files is very large. The experiments show that the performance of our system is significantly improved compared with Hadoop HDFS when reading and writing small files.

The system implemented the basic function of storage system for small files based on DHT, and the system reaches the requirement of online query. But the central node is the bottleneck when the system extends, so our next-stage works include the following contents:

(a)The node in cloud computing environment could not leave until the node failed or the node was damaged by natural disasters, which is small probability event. Our next-stage work is to study the relationship between the system performance and the probability the node failed.

(b)Implement the scalability based on the Trie structure.

REFERENCE

[1] Patascale Data Storage Institute, “NERSC file system statistics.” [2007]. http://pdsi.nersc.gov/filesystem.htm.

[2] Felix. E., “Environmental Molecular Sciences Laboratory: Static Survey of File System Statistics.” [2007]. http://www.pdsi-scidac.org/fsstats/index.html.

[3] Chervenak A., Schopf J. M., Pearlman L., et al. “Monitoring the Earth System Grid with MDS4.” e-Science and Grid Computing 2006. New York: IEEE, 2006:69-69.

[4] Neilsen E. H., “The Sloan Digital Sky Survey Data Archive Server.” Computing in Science Engineering, 2008, 10(1):13-17.

[5] Bonfield J. K., Staden R., “ZTR: a new format for DNA sequence trace data”. Bioinformatics, 2002, 18(1):3-10.

[6] Faraz Shaikh, Mikhil Chainani, “A case for small file packing in parallel virtual file system (pvfs2),” In Advanced and Distributed Operating Systems Fall 07, 2007.

[7] Carns P., Lang S., Ross R., Vilayannur M., et al. “Small-file access in parallel file systems” International Parallel and Distributed Processing Symposium, New York: IEEE Computer Society, 2009:1-11.

[8] Hendricks J., Sambasivan R. R., Sinnamohideen S.,et al. “Improving small file performance in object-based storage.” [2006]. http://www.pdl.cmu.edu/PDL-FTP/Storage/CMU-PDL-06-104.pdf.

[9] Kuhn M., Kunkel J. M., Ludwig T., “Dynamic file system semantics to enable metadata optimizations in PVF.” Concurrency and Computation: Practice and Experience, 2009, 21(14): 1775–1788.

[10] Mackey G., Sehrish S., Jun W., “Improving metadata management for small files in HDFS.” Cluster Computing and Workshops 2009. New York: IEEE Computer Society, 2009:1-4.

[11] Ghemawat S., Gobioff H., Leung S., “The Google file system.” Symposium on Operating Systems Principles 2003. New York: ACM.2003:29-43.

[12] Amazon. “Amazon Simple Storage Service” [2011]. http://www.amazon.com/s3.

[13] Shvachko K., Hairong Kuang, Radia, S., Chansler R., "The Hadoop Distributed File System," Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on , vol., no., pp.1-10, 3-7 May 2010.

[14] Decandia G., Hastorun D., Jampani M., et al. “Dynamo: amazon's highly available key-value store.” Special Interest Group on Operating System 2007. New York: ACM. 2007. 41: 205-220.

[15] Li xiu-qiao, Dong bin, Xiao Li-min, et al. “Small Files Problem in Parallel File System.” Network Computing and Information Security. New York: IEEE Computer Society. 2011: 227-232.

[16] Karger D., Lehman E., Leighton T., et al. “Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web.” Symposium on Theory of Computing 1997, New York: ACM, 1997: 654-663.

[17] Stoica I., et al. “Chord: A scalable peer-to-peer lookup service for internet applications.” in SIGCOMM '01. 2001. New York, NY, USA: ACM.

[18] Maymounkov P., D. Mazières, Kademlia: “A Peer-to-Peer Information System Based on the XOR Metric,” in Lecture Notes in Computer Science, P. Druschel, F. Kaashoek and A. Rowstron, P. Druschel, F. Kaashoek and A. Rowstron^Editors. 2002, Springer Berlin/Heidelberg. p. 53-65.

[19] Shvachko, K., et al. “The Hadoop Distributed File System.” in Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. 2010.