ISBN 978-93-84422-22-6
Proceedings of 2015 International Conference on Image Processing, Production and Computer Science (ICIPCS'2015)
Istanbul (Turkey), June 3-4, 2015 pp. 52-58

DLMCC - Distributed Lock Management in Cloud Computing

Artur Koçi[1], Betim Çiço[2]
[1,2] Computer Engineering, Epoka University, Tirana, Albania.
[1]akoci@epoka.edu.al, [2]bcico@epoka.edu.al

***Abstract*:** *Distributed storage systems with replication are well known for storing large amount of data. A large number of replication is done in order to provide reliability. This makes the system expensive. Various methods have been proposed over time to reduce the degree of replication and yet provide same level of reliability. One recently suggested scheme is of Regenerating codes, where a file is divided into parts which are then processed by a coding mechanism and network coding to provide large number of parts. These are stored at various nodes with more than one part at each node. These codes can generate whole file and can repair a failed node by contacting some out of total existing nodes. This property ensures reliability in case of node failure and uses clever replication. This also optimizes bandwidth usage. In a practical scenario, the original file will be read and updated many times. With every update, we will have to update the data stored at many nodes. Handling multiple requests at the same time will bring a lot of complexity. Reading and writing or multiple writing on the same data at the same time should also be prevented. In this paper, we propose an algorithm that manages and executes all the requests from the users which reduces the update complexity in cloud computing .*

Keywords: *cloud data storage, cloud computing, distributed systems, lock manager, voting algorithm, erasure codes etc.*

1. Introduction

Cloud data storage is a challenging field posing many new problems for researchers since big companies such as Facebook and Google are storing several terabytes of data of the users and still provide the smooth user experience of services. Many companies such as Amazon etc. are providing cloud data storage services and in doing so, it is needed to solve multi-dimensional optimization problem. For such a cloud storage, in order to make the data reliable one can use the classical technique of replication. This technique provides that the data on the nodes is replicated many times. This technique of storing data is very easy but may consume a lot of space. The technique it is used in popular distributed systems [2], where data is divided into parts, and stored at various nodes. One advantage of distributed storage is reliability. Even if one of the node gets corrupted or fails, then the other nodes still remain alive. Most modern day distributed systems use replication [2], [1], [3]. While using distributed systems, there is a high chance for the nodes to get damaged or to leave after some amount of time. That new node will be directly copied from its replication as per the current storing scenario. The amount of space consumed in storing the data in replicated manner is huge, for example, Gmail uses around 21 times replication [3]. Many researches are done in this field for deploying new technique for reducing the storage consumption. One way is to use the already existing nodes to make the new node instead of the replicated nodes. In such systems a file of size M is divided into n parts with each part of size M/n bytes and stored at different nodes in such a way that any node can be reconstructed using a subset of total n nodes. These codes are called Erasure codes [4]. Maximum distance separable(MDS) erasure codes described in [5], [4] are optimal erasure codes and can generate a node using k nodes where $k < n$. The main concern using MDS codes is the bandwidth consumption as compared to replication. We need to download data from k nodes in MDS codes instead of just one in replication. In a seminal paper Dimakis et al. [4] has shown that repair bandwidth can be reduced if we apply network coding [6] on parts of the file and store such linearly combined parts on every node. In order to repair a failed node, we now download data only from d nodes instead of k, where $d < n$, and from each node

instead of downloading the whole data, we download only β packets out of α, where $\beta < \alpha$. These codes are called Regenerating codes [4]. The repair bandwidth for the code is dβ. There is well known tradeoff between storage and repair bandwidth [4]. These codes can decrease the data that needs to be downloaded from the nodes to repair or construct the new node. Each file is divided, encoded, and stored at various nodes. Considering a practical scenario, there will be constant requests to read or write the file. Some nodes might fail in between, so there will be repair procedures going on. Any small change in the file will change its associated parts at various nodes. There has to be a proper method to do this, so that the multiple read's, repair's and write's on the parts of the file are controlled in a proper way and do not interfere with each other creating wrong results. Also this method should not decrease the availability of the nodes and should be able to perform many requests in parallel. Another approach offered by Author 1is develop an algorithm by which the read, write and repair requests are scheduled in a way such that we not encounter any inconsistency in the results and get the fastest results possible by processing as many requests as possible in parallel. Author 1 use the basic concept of simple voting mechanism with coding as in [7] where each node is given one vote to describe its state.

When we store data in a distributed system, we have to coordinate the access to the data on various nodes in a way that the operations are fast and their results are free from error. One has to optimize two main parameters. update complexity and repair bandwidth [5]. Network coding [6] has shown a way to reduce the repair bandwidth and provide better reliability [5]. In network coding, one performs mathematical operations on packets instead of just forwarding them. On the other hand quorum sensing has been used by researchers [7] to give efficient algorithms that reduces update complexity in such a scenario. One kind of voting mechanism, suggested by Gifford [8], assigns votes to the various nodes and they communicate with each other using these votes. The read and write quorums are defined to provide a read-write and a write-write exclusion. The read quorum r and write quorum w are that $r + w > N$ and $2 * w > N$, where N is the total number of votes assigned to all nodes. An approach called simple voting with coding (SVWC), as described in [7], defines read and write quorums for a replication scheme using coding to store files. The bounds on read and write quorum values were studied in this paper. These bounds were calculated in a way that you need at least votes equal to the lower bound to complete the read/write quorum and if equal to upper bound will always complete the read/write quorum. Mit Sheth et al [9] in his algorithm offers the solution for multiple chunk node with relative update. Instead of one vote per node they give α vote to each node. Every vote is distinct, even on the same node and each Node i will have votes from α i1,α i2 ..... α iα for its α number of packets, where $1 \leq i \leq N$. All the requests are directed to a super user . Whenever the super user asks the nodes for acceptance of a request , the vote bit and lock bit of every packet of every node is check. One a the drawback of this algorithm is that all requests are directed to a super user that is a single server. Another drawback is that algorithm provides a centralized management and it can be turned in a bottle neck. All the traffic is carried out from one point and this will cause traffic overhead. We use the basic concept from this algorithms on how votes can be used in managing a big network of nodes storages and offer distributed lock management algorithm for servers communication and synchronization in a proper order to avoid the read/write inconsistency. Throughout the whole discussion we assume that no votes are lost in the process. Server can fail only if it is stopped voluntarily. We also assume that all the votes of one request move at same speed in the network.

2. Preliminaries

Contributions In this section, we consider a code with MDS property and discuss the basic file operations in a practical scenario. We define various terms as we go on describing the algorithm. In the algorithm we design conditions that will prevent both read-write and write-write from executing at the same node at the same time. Consider a Maximum Distance Separable (MDS) code for a file F. The file is initially divided into k parts called as native chunks. Now, these k chunks are encoded by linear combination to form n code chunks, with $n > k$, and stored at N nodes. Considering the MDS property, any k nodes can be contacted to reconstruct the whole file again. The maximum number of failures which can be tolerated by the system is n - k. Every node in the cloud stores α chunks making nα chunks in total. We can contact any d nodes and download β out of α packets from each node in order to construct a new node which can replace the corrupted or failed node. For any given regenerating code, let us say any change in file F updates q chunks in total.

There are three operations that can be executed on the file.

1) Read/download/reconstruct the whole file- There will be a lot of users who would perform these requests. In this case, we will have to generate the complete file, and to do that, we will have to contact any k nodes out of total N nodes.
2) Write/update- Any change in a file has to change all the corresponding data in all nodes. Again there will be a lot of users who would try to perform simultaneous write requests, which will update the information stored in the nodes.
3) Node Repair- If a node gets corrupted or fails; it is needed to generate a new similar node. For this, we need to connect to d different nodes and download β packets from each of them.

A request for a shared file can be made from each of the server that has connection to the nodes as in fig 1

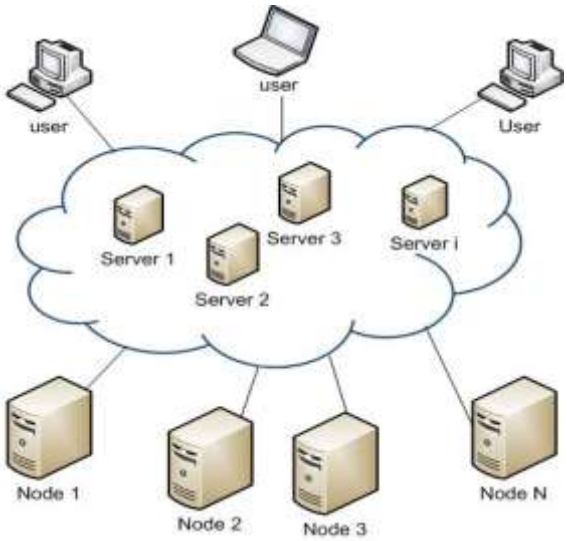

Fig 1: Cloud Computing Architecture

When any of the servers participation in the private cloud, receives any request from end users immediately initiates the procedure to acquire votes for accessing the file

The user requests can be categorized into above types. Everything can happen in a smoothly and requests comes from different servers do not interfere with each other. In a practical scenario, there will be constantly two or more requests from the same server and from different servers which might be requested at the same time. Such simultaneous requests might result in some problem. There would be some inconsistency when a certain request is reading from a certain node and suddenly a write operation tries to write on it. One way to operate and avoid inconsistency is to keep the incoming operations waiting and that write operation will keep trying till the read is completed. Following this approach, if another read request comes after the write request, it will also be put in the waiting queue behind write. There would be a lot of polling which is a waste of resources because two reads can occur at a same time. There will be many such requests trying to read or write a large number of nodes. If these requests are not properly controlled and managed, the whole network might get congested and may take too much time to process user requests.

3. The Proposed Algorithm

As mention in the preview section, centralized management scheme has some drawbacks. In this section we present the implementation of the proposed algorithm on Distributed lock management in cloud computing.

Definition 1: The read, write and repair quorums are defined as the minimum number of votes required to initiate the execution of that request. We denote the read, write and repair quorum by r , w and rep.

Our main concern is to define an approach which will try to reduce as many inconsistencies as possible and give the result of the request in the least possible time. There are six possible simultaneous request cases. Despite the total number of the simultaneous requests there will be made up of any such two pairs of requests only.

read-read, read-write , read-repair , write-write , write-repair , repair-repair

In order to describe the algorithm, let us first define which requests can execute simultaneously in a chunk.

read-read - Yes. Two simultaneous reads will never be a problem. They do not create any inconsistency.

read-write- No. As every write updates every node currently, there will be nodes on which both operations would be executed simultaneously. This will bring an inconsistency in read operation.

read-repair- Yes, because repair is just another kind of a read request contacting lesser nodes.

write-write- No. Simultaneous write's will definitely create inconsistency in as both of them will try to update every node at the same time giving incorrect results in the end. write-repair- No. It is another kind of read-write repair. Even though the number of nodes that we are contacting for repair operation is less, the write operation is updating all the nodes and so there will definitely be some common nodes and so the inconsistency.

repair-repair- Yes. It is an another kind of read-read case and so it is possible.

We define a unique vote-bit and lock-bit of one bit on every chunk. When the vote bit is 1, the node can send votes and when it is 0, they cannot. Lock-bit for a node is a one bit value describing what lock is on it, read lock or write lock. If the lock bit is 0 then there is no lock on it. We call the servers a super-users from which the read, write and repair requests will be executed and end nodes as end users from where the requests are coming. This is shown in Figure 1. Whenever the super users gets any kind of request, it will check if the nodes are free to execute that request on them. We will discuss later the pattern in which the super-users will send requests. We now define two kinds of locks and Lock Manager.

1) Read lock- Whenever any node is free to be read, we apply a read lock on it keeping its vote bit unchanged i.e. equal to 1 and lock bit to 1. When the read lock is set, it cannot surpass write requests but can allow multiple read and repair requests. Once read lock is set, the server starts reading from that node and when the read operation is done, the server node frees it from the read lock and the lock bit is changed to 0.
2) Write lock- Whenever any node is free to get updated, we try to form a write lock on it after changing its vote bit to 0 and lock bit to 2. When the write lock is set, it cannot allow any kind of further requests and the super user starts updating that node. When the write operation is done, the super user node frees it from the write lock and changes its vote bit to 1 again and lock bit to 0.

4. DLMCC - Distributed Lock Management in Cloud Computing.

4.1. DLMCC Architecture

In DLMCC, a lock manager is responsible for preserving data consistency during concurrent file accesses from servers to the nodes. A lock manager on a server cooperates with lock managers on other servers to control concurrent access on the shared files by using inter-process communication among them and various lock statuses kept in their internal lock control tables. The lock manager runs as a process on every server and works as an external lock management module for applications and the file system in the operating system on the server. The managers function only to coordinate concurrent file accesses when clients issue lock requests for files saved in the end nodes storages.

Our aim is to control read and write lock consistency in the shared files. All locks are advisory. Only one lock manager is responsible for a file at a time across the system.

Data structures in DLMCC

In the internal structure of the lock manager, there are six key data structures for performing concurrency control and self-management of shared file between server in the cloud. Server Node Table (SNT).File Directory (FD), Migrate-out Table (M-outT), Migrate-in Table (M-inT), Requesting Lock Table (RLT), Locked File List (LFL).

Those data structures are essential for lock managers to keep such entities as consistent across the cloud. Each lock manager has one set of the data structures in its own memory in a fully distributed manner without sharing any of them at a centralized site. SNT works for memorizing configurations of all server nodes in the cloud with the information such as servers ID, Servers status and switchover server in the case of server failures. The table is fully replicated on all servers. FD is the directory to determine which server is responsible for locking specific entities. W e assume that FD is a data structure with pairs of file names and server number and

is fully replicated in a cloud. However, it does not necessarily appear in the form of a data structure, but can be done in calculation with a hash function of an file name or a file block address to be locked. M-inT and M-outT maintain information to redirect lock management for an file to a lock manager on the appropriate server when request migration happens. RLT keeps track of all locks a server has requested. The attributes in RLT consist of a requester ID, and file name to request, a lock mode and a timestamp. The table is local and works to manage shared locks locally in the server requesting locks and decrease overall locking overhead in the cloud. LFL includes all necessary information about locks for which a lock manager is managing and responsible. LFL has a set of attributes of a requester ID, a requester server, an file name, a lock mode and a timestamp, and two lists for queuing lock requests: one for granted and the other for blocked locks.

4.2. Lock Manager Algorithm

Lock managers are processes running on the cloud with one manager per server. In the cloud will be many client requests that will ask for granting access to a specific file at the same time. Those requests perform concurrent accesses on the (files) and request their server lock managers the necessary access controls with parameters of a lock requester ID, an file name and a lock mode. The lock managers in the cloud cooperate for the lock management necessary to perform lock request operations. The lock management algorithm will follow the sequence of four steps to archive the lock to an file and respond to the client as described in Fig. 2.

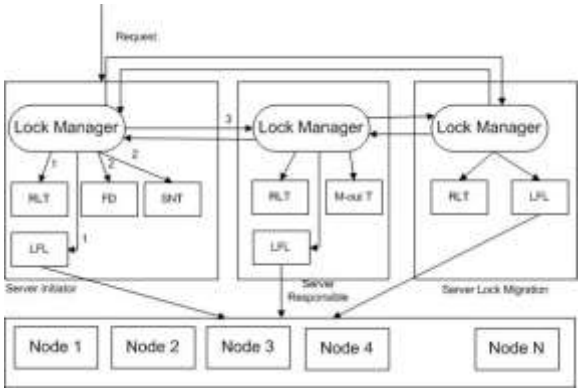

Fig 2: DLMA Architecture

Step I: Self-management of shared locks in servers

The end users issues a request to a server for accessing entities in the storage nodes. The server receives the request and asks the lock manager to looks up if the file name in the request already exists in the table of its RLT. If the file name does the manager adds the request in the table of RLT with the requester ID and the related attributes for the requested file, than the request is added to the LFL table and proceeded. LFL receives the request starts sending vote to the nodes for acquiring quorum vote to read or write the file requested.

When LFL receives a request from the end users, tries to know the availability of the chunk saved in the nodes for that request by sending the request to the nodes. Every request from the server to the nodes has a unique request Id. Nodes will be available if there is no lock on them, and if there is a lock, then the availability will depend on the type of lock on them. If they are available for a certain request, they will send a vote to the server in form of a two bit (node number, request id,) number only if their vote bit is 1. When vote reaches server, it checks in the lock table if that node can be granted that lock. If it can be granted, then it updates its lock table with the corresponding lock on the end user request id, and then updates the vote bit and lock bit on the node. It would not be necessary to update the vote bit and lock bit if the node on which the request was granted was already locked, because these requests will be read-read, read-repair or repair-repair and all these have same vote bit. The node structure can be seen in Fig 3

Lock Bit
Vote Bit
Chunk Data

Fig 3: Node Structure

The request is managed solely by the lock manager in the node. Processing the lock request is done at this moment without interacting with other lock managers at remote servers. If the file name does not exist in RLT, it is necessary to acquire a new lock on the file by cooperating with other lock managers as described in the steps below.

Step 2: Finding a lock manager

The first procedure is to look up FD and find the node of the lock manager responsible for the file. The next one is to check SNT to find out if the server is alive and still in the cloud. If the Server exists alive in the Cloud, the lock manager on that server is responsible for the file. If not, the switchover Server in the table of SNT is taken as the alternative node for the lock management. The lock manager at the lock initiator server sends a lock request message to the lock manager on the node found in the above procedures.

Step 3: Checking a request migration

The lock manager which receives a lock request message from the lock request initiator server is responsible for managing the lock on the file in the request. The manager checks in the table of RLT and on M-out T if any lock is applied to the file or the lock management is migrated to a lock manager on another Server. If the name of the file is not found in RLT, or in M-out T, the responsible server puts the file in M-out T and the lock management is migrated to the server initiator. Server initiator grants lock management for the file.

If the file name on the request is found in RLT of the server responsible it means that the server has locked the file . The responsible server informs the server initiator and the request is migrate for the server initiator to that server and is proceeded from there.

If the file name is found on M-out T the request is migrated to that server and proceeded from that specific server

Step 4: Lock acquisition

The lock manager receives the request and updates it RTL and updates also the LFL Table with the file name and its attributes. Than inform the server initiator that the request is added in the list for the execution. The initiator server removes the request from its RLT table. After migration of the request from the server initiator, the server starts sending votes to nodes for r/w/rep the file in the request

There are two lists in LFL, which are the timeout time t_0 and a request queue. The request queue distributes the incoming requests in to slots, where all the requests of one slot are handled together and every request has its own priority.

5. Conclusion

We proposed an algorithm that can be used to manage the read, write and repair requests on data which is stored using regenerating codes. The algorithm proposed a solution to manage a large number of requests in a distributed system in a way such that they can be executed in least possible time. The algorithm also proposed a way to prevent inconsistencies that can happen due to read and write performed on the same node at the same time from different servers that participate in the cloud. the proposed algorithm can be implemented in any cloud that uses regenerating codes. The shared files are kept in a proper way that are free from the errors and fault tolerant from the server failures. Each file has the server responsible and a switchover server in case of server failure.

6. References

[1] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin,, "Erasure coding in windows azure," in USENIX conference on Annual Technical Conference, ser. USENIX ATC'12., Berkeley, CA,, 2012.

[2] K. Nguyen, T. Nguyen, Y. Kovchegov, and V. Le,, "Distributed data replenishment," IEEE Transactions on Parallel and Distributed Systems, vol. 24, no. 2, pp. 275-287, 2013.

http://dx.doi.org/10.1109/TPDS.2012.115

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung,, "The google file system," Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 29-43, 2003.

http://dx.doi.org/10.1145/945445.945450

[4] A. Dimakis, P. Godfrey, M. Wainwright, and K. Ramchandran,, "Network coding for distributed storage systems,," in INFOCOM 2007. 26-th IEEE International Conference on Computer Communications, IEEE, 2007.

http://dx.doi.org/10.1109/infcom.2007.232

[5] A. Dimakis, K. Ramchandran, Y. Wu, and C. Suh,, "A survey on network codes for distributed storage,," Proceedings of the IEEE, vol. 99, no. 3, pp. 476-486, 2011.

http://dx.doi.org/10.1109/JPROC.2010.2096170

[6] R. Ahlswede, N. Cai, S.-Y. Li, and R. Yeung,, "Network information flow," Information Theory, IEEE Transactions on, vol. 46, no. 4, pp. 1204-1216, 2000.

http://dx.doi.org/10.1109/18.850663

[7] A. Agrawal and P. Jalote,, "Coding-based replication schemes for distributed systems," Parallel and Distributed Systems, IEEE Transactions on, vol. 6, no. 3, pp. 240-251, 1995.

http://dx.doi.org/10.1109/71.372774

[8] D. K. Gifford, "Weighted voting for replicated data," Proceedings of the seventh ACM symposium on Operating systems principles, pp. 150-162, 1979.

http://dx.doi.org/10.1145/800215.806583

[9] Mit Sheth, Krishna Gopal Benerjee, Manish K. Gupta, "Quorum Sensing for Regenerating Codes in Distributed Storages," in 2014 Sixth International Conference on Communication Systems and Networks (COMSNETS), 2014.

http://dx.doi.org/10.1109/COMSNETS.2014.6734929