A Load Balancing Algorithm For Private Cloud Storage

Prabavathy.B
Dept. of Computer Science
SSN College of Engineering
Kalavakkam, Tamilnadu, India
prabavathyb@ssn.edu.in

Priya.K
Dept. of Computer Science
SSN College of Engineering
Kalavakkam, Tamilnadu, India
preeyah77@gmail.com

Chitra Babu
Dept. of Computer Science
SSN College of Engineering
Kalavakkam, Tamilnadu, India
chitra@ssn.edu.in

***Abstract*—Cloud computing enables on-demand network access to a shared pool of configurable computing resources such as servers, storage and applications. These shared resources can be rapidly provisioned to the consumers on the basis of paying only for whatever they use. Cloud storage refers to the delivery of storage resources to the consumers over the Internet. Private cloud storage is restricted to a particular organization and data security risks are less compared to the public cloud storage. Hence, private cloud storage is built by exploiting the commodity machines within the organization and the important data is stored in it. When the utilization of such private cloud storage increases, there will be an increase in the storage demand. It leads to the expansion of the cloud storage with additional storage nodes. During such expansion, storage nodes in the cloud storage need to be balanced in terms of load. In order to maintain the load across several storage nodes, the data need to be migrated across the storage nodes. This data migration consumes more network bandwidth. The key idea behind this paper is to develop a dynamic load balancing algorithm to balance the load across the storage nodes during the expansion of private cloud storage.**

***Index Terms*—Cloud computing, private cloud storage, data placement, load balancing, data migration.**

I. Introduction

Cloud computing provides a shared pool of configurable computing resources to the consumers. This technology effectively utilizes the resources such as computation power, storage, memory and bandwidth. Cloud storage is a repository in which the data is maintained, managed, and is made available to the consumers over the Internet. There are four ways in which the cloud storage can be deployed. They are private, public, community and hybrid cloud.

Private cloud is provisioned for the users of a single organization exclusively. It can be accessed from anywhere and at any point of time within the organization. Public cloud is provisioned for open use by general public. Community cloud is provisioned for exclusive use by a specific community of consumers from several organizations. Hybrid cloud is a composition of two or more distinct clouds such as private, public or community clouds.

Even though the public cloud can be shared and accessed from anywhere at any time, it has certain significant security related risks due to data remnants, unencrypted data and shared multi-tenant environments. Hence, private cloud storage is used by organizations which need more customization and control over data. It is built by exploiting the commodity machines within the organization where several users store their data. There may be increasing storage demands from the existing consumers or the number of consumers itself may increase. These situations necessitate the dynamic expansion of cloud storage with additional storage nodes.

During such expansion of the storage cluster or due to the addition and deletion of files, load imbalance may arise in the cluster. In order to maintain the balance in the cluster, a load balancing algorithm has been proposed in this paper. This algorithm attempts to balance the load during the data placement as well as in any later situations that lead to load imbalance.

The rest of this paper is organized as follows. Section II discusses the related works. Section III describes the architecture of the proposed work. Section IV discusses the proposed algorithms. Section V describes the simulation. Section VI describes the implementation and results. Section VII concludes and provides future directions.

II. Related works

Myint et al. [1] proposed an approach that deals with data placement algorithm in cloud storage system. In this work, the commodity machines are used to build the private cloud storage. This makes use of storage capacity and the failure probability to find a weight for each machine. Weights of the storage nodes are used to construct a binary tree. This tree is searched to find the node with the maximum weight to assign the workload. However, network utilization of the link to which the commodity machines are connected has not been considered to find the weight of the storage nodes.

Zeng et al. [2] proposed an approach that deals with load rebalancing in large scale Distributed File Systems (DFS). In this work, rebalancing ensures that before the actual migration, heavily loaded node should check for the existence of a replica in the lightly loaded node.

In Myint et al. proposed approach the data placement algorithm does not use the network utilization for finding the weight for the storage node. In Zeng et al. proposed approach the load rebalancing in DFS does not clearly specify how much data has to be migrated. However, the proposed private cloud

storage need to take care of the network utilization to find the weight as commodity machines are used to construct the storage cluster. Hence, this paper proposes suitable algorithms for data placement, rebalancing and data migration to achieve load balancing in private cloud storage.

III. Architecture of Private Cloud Storage

A private cloud storage is built using the commodity machines. These are considered as a set of storage nodes $S = \{S_i/i\varepsilon(1...n)\}$ with heterogeneous configurations. One machine acts as a centralized coordinator (CC). It acts as an interface between the commodity machines and the client. It is used to coordinate the storage resources of the commodity machines. Once the storage resources of the commodity machines are consolidated, it is provided as a service via Internet. Walrus controller is the storage controller installed in CC to serve the storage request from the client which is forwarded to the deduplication engine for finding the unique chunks of the input file and in turn to load balancer to balance the load across the storage nodes. This is illustrated in Figure 1.

Commodity machines are made to coordinate with each other via Transmission Control Protocol (TCP). File Transfer Protocol (FTP) is used to transfer files across the commodity machines. During the start-up of the storage cluster, storage nodes register their details such as free space, network utilization, utilized space and IP address with the CC. In addition, these details are sent by the storage nodes to CC periodically. As the storage space within the private cloud storage is limited, it has to be optimally utilized. Hence, deduplication approach [3] is used for efficient usage of the storage space. This approach attempts to find the duplicate content across the files.

In order to enable this, a file is split into several chunks and these chunks are distributed across the storage nodes. $F = \{F_i/i\varepsilon(1...n)\}$ represents the workload for the storage system, where F is set of files of that belong to different types and $f = \{f_i/i\varepsilon(1...n)\}$ represents the size of each File F_i. $F_i = \{F_iC_1, F_iC_2.....F_iC_m\}$ represents the chunks of a particular file.

Whenever a write request for a file is received by the CC, it is forwarded to the walrus controller such that the file is split into several chunks with the help of deduplication engine. It forwards only the unique chunks to the load balancer for distributing them across the various storage nodes in the cluster. When the chunks are placed initially across the storage nodes, it is essential to place them in such a way to maintain the load balance across the storage nodes. Subsequent usage of the storage cluster leads to load imbalance in the cluster. This load imbalance may occur due to the following reasons.

- Inclusion of a new storage node - Newly added storage node may have lesser or no data compared to the other nodes.
- Addition or deletion of files - It leads to imbalance in the cluster.
- Heavily loaded node - When the load exceeds a predefined threshold, the storage node becomes heavily loaded.

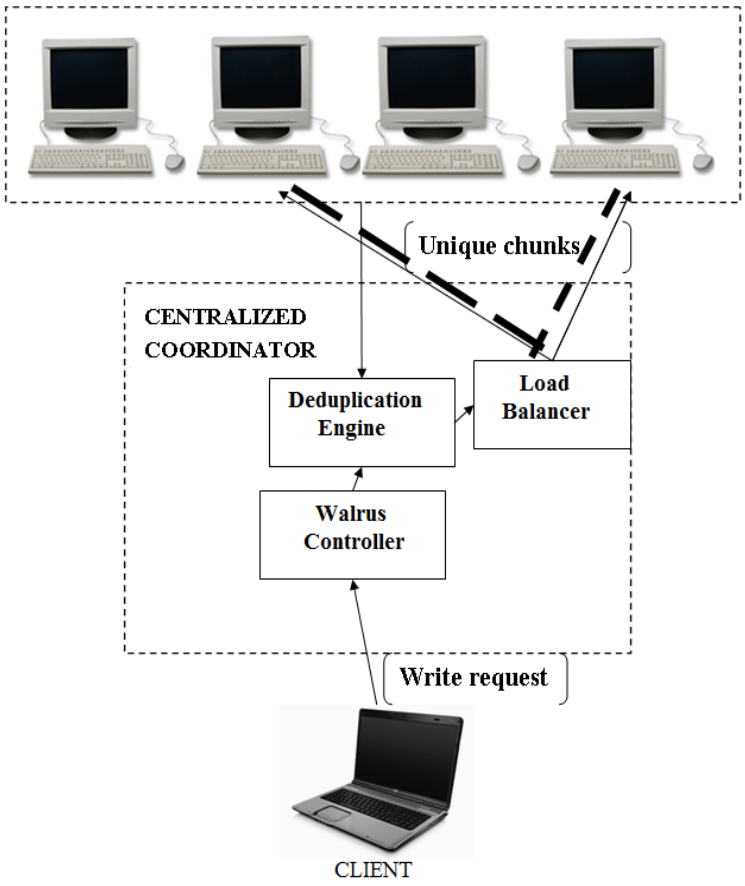

Fig. 1. Architecture of the private cloud storage

The above mentioned imbalance situations trigger the data to be migrated from one storage node to another. However, this migration should also ensure that the placement of replicas of any given chunk happens in different storage nodes. The load balancing activities are performed by the load balancer module of the CC. In order to coordinate these activities, suitable algorithms are proposed which are explained in the following section.

IV. Proposed Algorithm

Whenever the CC receives the write request for a file, load balancer module in the CC is used to distribute the chunks of that file in such a way to balance the load across the storage cluster. Load balancer consists of coordinator, data placement and load rebalancer sub modules.

A. Coordinator sub module

The coordinator sub module receives the status of the storage nodes, that are registered in the cluster. It is represented as a vector. This vector consists of capacity, utilized space, residual space, network utilization, weight, idealLoad, IP address and loadStatus of each storage node. Then, it receives the unique chunks of the file from the deduplication engine. These unique chunks have to be distributed across the storage

nodes for balancing the load in the cluster. This is illustrated in Algorithm 1.

Algorithm 1: COORDINATOR ALGORITHM

Input:
struct storage
float capacity, utilizedSpace, residualSpace, networkUtilization, weight, idealLoad,
int size
string IPaddress, loadStatus
storage *sptr, *lptr, *hptr
storage light, heavy
File *fptr[]//Array of pointers to unique chunks of the File
int loadCluster← 0 //workload of the cluster
Output: pointer to storage nodes

1 Receive the status of each storage node in a vector

2 Receive the unique chunks of the file to be sent to storage nodes.

3 Compute the size of the received file

4 Invoke the data placement algorithm, dataplacement(sptr,fptr) and receive the IP addresses of two suitable storage nodes

5 Send the unique chunks of the file and their replicas to two storage nodes

6 loadCluster ← loadCluster + (2 * size)

7 It invokes load rebalance algorithm, loadrebalance(sptr, lptr, hptr, loadCluster) to return the sorted list of light weighted and heavy weighted nodes in the storage cluster.

8 Identify the light and heavy weighted node pairs using lptr and hptr

9 Notify the heavy weighted node regarding the corresponding pair to perform data migration

B. Data placement sub module

This sub module receives the status of the storage cluster and the chunks of the file to be distributed from the coordinator sub module. From the status of the storage nodes, weight is calculated for each storage node using the formula $weight = (\omega 1 * networkUtilization) + (\omega 2 * residualSpace/totalCapacity)$ where $\omega 1$ is the weight factor for network utilization and $\omega 2$ is the weight factor for storage utilization and they are assumed to be 0.25 and 0.75 respectively. Storage nodes are sorted based on their weight in descending order. Each storage node is traversed sequentially and the first replica is placed in a node where it is able to fit in. Similarly second replica is also placed. This is described in Algorithm 2.

Algorithm 2: DATA PLACEMENT ALGORITHM

1 dataplacement(storage *sptr, File *fptr)
Input: *storage* *sptr, temp
File *fptr
float $\omega 1$ // Weight factor for network utilization
float $\omega 2$ // Weight factor for storage utilization
int: i, j, k
int n // Number of storage nodes registered in the cluster
Output: ptr to String Array B[]
2 $\omega 1 \leftarrow 0.25$
3 $\omega 2 \leftarrow 0.75$
4 // Calculation of weights for each storage node
5 **for** $i \leftarrow 0$ ***to*** $n-1$ **do**
6 $sptr[i]->weight \leftarrow (\omega 1 * sptr[i]->$
7 $networkUtilization)+$
8 $(\omega 2 * (sptr[i]->$
9 $residualSpace/sptr[i]->capacity))$
10 // Sorting storage nodes based on the weight
11 **for** $i \leftarrow 0$ ***to*** $n-1$ **do**
12 **for** $j \leftarrow i+1$ ***to*** n **do**
13 **if** $(sptr[i]->weight) < (sptr[j]->weight)$ **then**
14 $copy(temp, sptr[i])$
15 $copy(sptr[i], sptr[j])$
16 $copy(str[j], temp)$
17 // Finding the suitable nodes to place the data
18 $i \leftarrow 0$, $k \leftarrow 0$
19 **while** $i < n$ **do**
20 **if** $size < (sptr[i]->residualCapacity)] \bigwedge c <= 2$ **then** $strcpy(B[k++], sptr[i]->IPaddress)$
21 $c \leftarrow c+1$
22 Return ptr of Array B

C. Load rebalance sub module

This sub module is responsible for periodically checking whether the storage nodes are lightly or heavily loaded. In order to detect this, first the ideal load for each storage node has to be found. Ideal load can be found as follows. Ideal load for a given node = (Overall load of the cluster/overall capacity of the cluster) * Capacity of the given node. From the ideal load, lower and higher thresholds for each storage node are found. This is illustrated in Algorithm 3. The utilized space of each storage node is compared with the lower and higher threshold values. If the utilized space of a storage node is less than the lower threshold, it is considered as a light weighted storage node and its status is set as "low". Similarly, if the utilized space of a storage node is greater than the higher threshold, it is considered as a heavy weighted storage node and its status is set as "high". Then, the light weighted nodes are sorted in ascending order and heavy weighted nodes are sorted in descending order. Once this ordering is finished, each light weighted node is paired with a corresponding heavy

Algorithm 3: LOAD REBALANCE ALGORITHM

loadrebalance(sptr, lptr, hptr, loadCluster)
Input: *storage* *sptr, *lptr, *hptr
float loadCluster // Workload for the cluster
float overallCapacity // Overall storage capacity of the cluster
float lowThreshold // Low threshold to detect light weighted node
float highThreshold // High threshold to detect heavy weighted node
int n // Number of storage nodes registered in the cluster
float delta // System parameter
int j, k
Output: segregated light weighted and heavy weighted storage nodes
$overallCapacity \leftarrow 0$
$delta \leftarrow 0.2$
$j \leftarrow 0$
$k \leftarrow 0$
// Computation of overall capacity of the cluster
for $i \leftarrow 0$ ***to*** $n-1$ **do**
  $overallCapacity \leftarrow overallCapacity + sptr[i]->capacity$
// Computation of ideal load for each storage node
for $i \leftarrow 0$ ***to*** $n-1$ **do**
  $sptr[i]->idealLoad \leftarrow ((loadCluster/overallCapacity) * sptr[i]->capacity)$
// Detection of light and heavy weighted storage nodes
for $i \leftarrow 0$ ***to*** $n-1$ **do**
  $lowThreshold \leftarrow (1-delta) * sptr[i]->idealLoad$
  $highThreshold \leftarrow (1+delta) * sptr[i]->idealLoad$
  if $sptr[i]->utilizedSpace < lowThreshold$ **then**
    $strcpy(sptr[i]->loadStatus, "low")$
  if $sptr[i]->utilizedSpace > highThreshold$ **then**
    $strcpy(sptr[i]->loadStatus, "high")$
// Segregating the light weighted storage nodes
for $i \leftarrow 1$ ***to*** n **do**
  if $sptr[i]->loadStatus == "low"]$ **then**
    $copy(lptr[j++], sptr[i])$
sort hptr in ascending order based on utilizedspace
// Segregating the heavy weighted storage nodes
for $i \leftarrow 1$ ***to*** n **do**
  if $sptr[i]->loadStatus == "high"]$ **then**
    $copy(hptr[k++], sptr[i])$
sort hptr in descending order based on utilizedspace

weighted node for load rebalancing. The pairing stops if the heavy weighted node cannot find a suitable light weighted node.

D. Data migration algorithm

In order to balance the load across the storage cluster, the load from heavy weighted nodes has to be migrated to the light weighted nodes. This is called data migration. It is carried out by data migration algorithm in each storage node. In order to facilitate the data migration, CC maintains a file index and each storage node maintains a migration index. File index is maintained for each file. It consists of file name and its hash value of the file as shown in Figure 2.

FILE NAME	HASH VALUE
file1.txt	34b20bb545.....
file2.txt	aq54sw65de.....
file3.txt	jknsf54jn87j.....
=	=
file n.txt	58hgf65bjh98.....

Fig. 2. File Index

Migration index consists of chunkID, chunk name and its size as shown in Figure 3.

HASH VALUE OF THE PATH OF THE FILE	PATH OF THE FILE	SIZE OF THE FILE
34b20bb545bg7k5d0.....	file1.txt	19 KB
98y43c2j5054jhb54j.....	file1.part0.txt	58 bytes
54j54f34jhb89f87v87....	file1.part1.txt	187 bytes
=	=	=
6f9135422b9be9a85...	file1.partn.txt	576 bytes
34b20bb545njiu545.....	file2.txt	20 KB
f478888e10d08bf44.....	file2.part0.txt	
=	=	=
4678e9908fb2f41dj.....	filen.partn.txt	123 bytes

Fig. 3. Migration Index

While migrating the data from the heavily loaded node to a lightly loaded node, two things have to be considered. First is that the lightly loaded node should not become heavily loaded during migration. Hence, LoadToBeMoved and LoadToBeAccepted for each storage node are calculated to find how much data can be migrated to a lightly loaded node. It is calculated using the formula $LoadToBeMoved = heavy-> utilized - heavy-> idealload$ for heavily loaded nodes and $LoadToBeAccepted = light-> utilized - light-> idealload$ for lightly loaded nodes.

Once these values are calculated, if LoadToBeAccepted is greater than or equal to LoadToBeMoved, data can be migrated from heavily loaded nodes to lightly loaded nodes. Second is that the replica of a file that is going to be migrated from heavily loaded node should not exist in the lightly loaded node. For this, the hash value of the file in file index is checked with the migration index for its existence. If it does not exist, the chunks of the corresponding file are transferred to the storage node and the migration index is updated.

Algorithm 4: DATA MIGRATION ALGORITHM

```
1  datamigration(light,heavy)
   Input: storage light, heavy
   Output: Data migrated from heavy weighted nodes to
           light weighted nodes
2  //find the sizeload for the heavy loaded node
3  for i ← 0 to n − 1 do
4      LoadToBeMoved = heavy− >
       utilized − heavy− > idealload
5  //find the sizeload for the lightly loaded node
6  for i ← 0 to n − 1 do
7      LoadToBeAccepted = light− >
       utilized − light− > idealload
8  for i ← 0 to n − 1 do
9      find the light weighted nodes whose
       LoadToBeAccepted >= LoadToBeMoved
10 while count >= sizeload do
11     for each file in file index
12     Get hash value of the file for its existence
       in the lightly loaded node.
13     if it does not exist, move the chunks of that
       file by referring to the chunks in migration
       index.
14     count = count + size of the file
15     Notify the change in the machine where
       the index is maintained.
```

V. SIMULATION AND RESULTS

In the university use case scenario, in order to efficiently utilize the storage space of commodity machines, deduplication technique is employed. It splits the file into several chunks. The size of these chunks are in KBs or MBs and the capacities of the commodity machines are in GB.

Due to this reason, it is complex to check the load balance across the storage nodes in real time as it takes more time. In order to check the effectiveness of the algorithms, the same real environment is simulated. It is carried out with one machine. Details of each storage node are maintained with some initial values.

All the modules such as coordinator, data placement, load rebalance and data migration are implemented. In order to check whether the load has been distributed across the storage nodes uniformly, load at each node is calculated by the definition as follows. $loadNode = utilizedSpace/capacity$.

Load is tabulated for each storage node to understand the load distribution. It is inferred from Figure 4 that the distribution of load across the storage cluster is balanced.

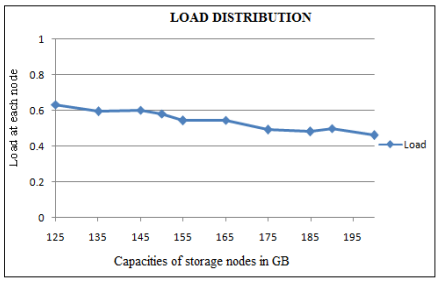

Fig. 4. Load distribution in storage cluster

Even though the load seems to be balanced, after a particular point of time, the storage nodes in the cluster may become lightly loaded and heavily loaded nodes. In order to make the storage cluster balanced again, load rebalancing is necessary. Heavy weighted nodes are paired with the light weighted nodes in the storage cluster. Subsequently, data migration is carried out. Thus, the load becomes rebalanced.

VI. IMPLEMENTATION

A test bed is created using four machines. Each machine has the configuration of 8GB hard disk , 4GB RAM capacity, and Intel core i5, i7 processor. One of the machines is designated as the centralized coordinator (CC). It acts as a TCP server and FTP client. The remaining three machines are the storage nodes that act as TCP clients and FTP servers. This is illustrated in Figure 5.

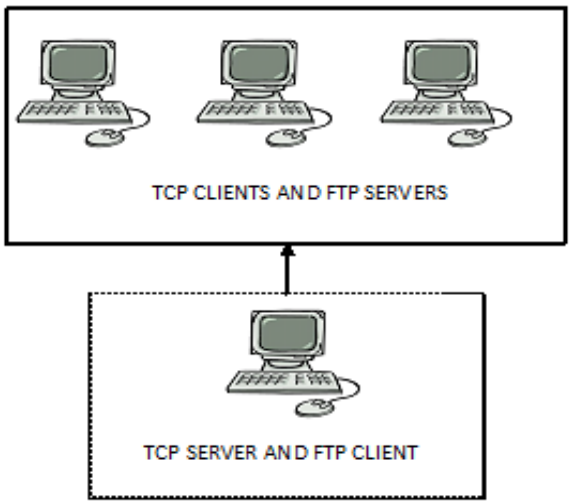

Fig. 5. Test bed creation

Private cloud storage is set up using Linux-based open source Eucalyptus framework version 3.2 [4]. Walrus controller of the Eucalyptus component is modified suitably to perform the deduplication in order to optimally use the storage space. The load balancer module is implemented in the centralized coordinator (CC), which consists of coordinator, data placement and load rebalancer sub modules. In the storage nodes, data migration algorithm is implemented. These are shown below.

In Figure 6, the TCP clients (storage nodes) find their IP address, free space, total space and utilized space to the TCP server (CC).

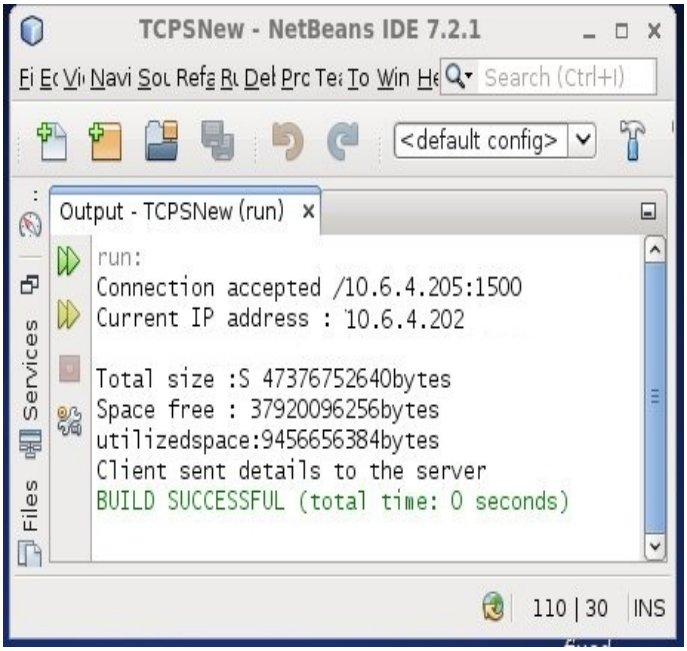

Fig. 6. Storage node 1

CC sends the unique chunks of the file and their replicas to the storage nodes based upon the weight. Figure 7 shows that the file gets stored efficiently in the suitable storage nodes (FTP servers).

VII. Conclusion

A private cloud storage is built using commodity machines. As the storage space within the private cloud storage is limited, it has to be effectively used. Deduplication technique attempts to store unique instances of the data in the storage. These unique instances of the files have to be placed uniformly across the storage nodes in order to balance the load across them. Though the data placement balances the load in the cluster, there may be imbalance in the cluster due to the expansion of the cluster. Hence, centralized load balancing technique is proposed to balance the cluster. This technique proposes data placement, load rebalancing and data migration algorithms.

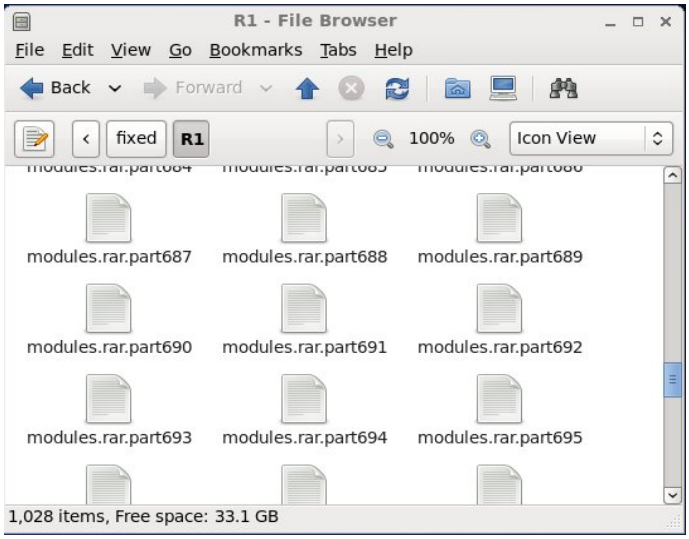

Fig. 7. Chunks of the file gets stored in storage nodes

Data placement algorithm is implemented in the simulation environment and it is evident from the result that it balances the load across the storage nodes in the cluster. Proposed load rebalancing and data migration algorithms are yet to be implemented and the migration cost has to be analyzed.

References

[1] J. Myint, T. Naing. "A data placement algorithm with binary weighted tree on PC cluster-based cloud storage system". *IEEE International Conference on Cloud and Service Computing (CSC), 2011*, pp. 315–320.

[2] W. Zeng, Y. Li, J. Wu, Q. Zhong and Q. Zhang. "Load Rebalancing in Large-Scale Distributed File System."*IEEE 1st International Conference on Information Science and Engineering (ICISE), 2009, pp. 265-269.*

[3] W. Zeng, Y. Zhao, K. Ou and W. Song. Research on cloud storage architecture and key technologies. In Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human, pp. 1044-1048. ACM, 2009.

[4] http://www.eucalyptus.com/eucalyptus-cloud/tools/s3curl