Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 3994 – 4008

doi:10.3906/elk-1804-221

A distributed load balancing algorithm for deduplicated storage

Prabavathy BALASUNDARAM[1,*], Chitra BABU[1], Pradeep RENGASWAMY[2]
[1]Department of Computer Science and Engineering, Anna University, Chennai, Tamil Nadu, India
[2]Department of Computer Science and Engineering, IIT Kharagpur, Kharagpur, West Bengal, India

Received: 30.04.2018 • **Accepted/Published Online:** 08.04.2019 • **Final Version:** 18.09.2019

Abstract: While deduplication brings the advantage of significant space savings in storage, it nevertheless incurs the overhead of maintaining huge metadata. Updating such huge metadata during the data migration that arises due to load balancing activity results in significant overhead. In order to reduce this metadata update overhead, this paper proposes a suitable alternate index that tracks the data blocks even when they migrate across the nodes without explicitly storing the location information. In addition, a virtual server-based load balancing (VSLB) algorithm has been proposed in order to reduce the migration overhead. The experimental results indicate that the proposed index reduces the metadata update overhead by 74% when compared to the existing index. Furthermore, VSLB reduces the migration overhead by 33% when compared to the existing ID reassignment-based load balancing approach.

Key words: Deduplicated storage, data deduplication, load balancing, chord protocol, gossip-based aggregation protocol

1. Introduction

In the past decade, there has been a tremendous explosion in data storage and retention needs of various business sectors. This necessitates the usage of a sophisticated space-saving technique, namely deduplication [1], for efficient utilization of the storage. This technique divides every incoming file into a set of blocks of either fixed or variable size. Subsequently, a cryptographic algorithm, namely secure hash algorithm 1, is utilized to find the hash values (also called fingerprints) corresponding to the blocks. These fingerprints are maintained in a fingerprint index [2]. Each fingerprint entry holds the fingerprint of a specific block, the location of that block, and a count indicating the number of files that share that block. Since every file is stored as a set of blocks that are distributed across the cluster, in deduplicated storage, it is essential to maintain a file recipe to reconstruct the file during the read operation.

In the present research work, deduplicated storage has been developed using commodity hardware. The workload for the proposed deduplicated storage is non-backup and consists of individual files with no significant locality among them. The deduplicated storage maintains the metadata and data blocks in separate clusters of machines. Whenever the load of the deduplicated storage has to be balanced, data blocks need to be migrated from heavily loaded nodes to suitable lightly loaded nodes. This would necessitate updating the location information in the fingerprint entries corresponding to those blocks. This is referred to as metadata update overhead (MUO) henceforth. Devising a load balancing solution with minimal MUO for non-backup workloads in deduplicated storage is an important issue. Existing research solutions such as those in [3] and [4] have utilized content similarity among the files for placing the data in a suitable node to balance the load in the

*Correspondence: prabavathyb@ssn.edu.in

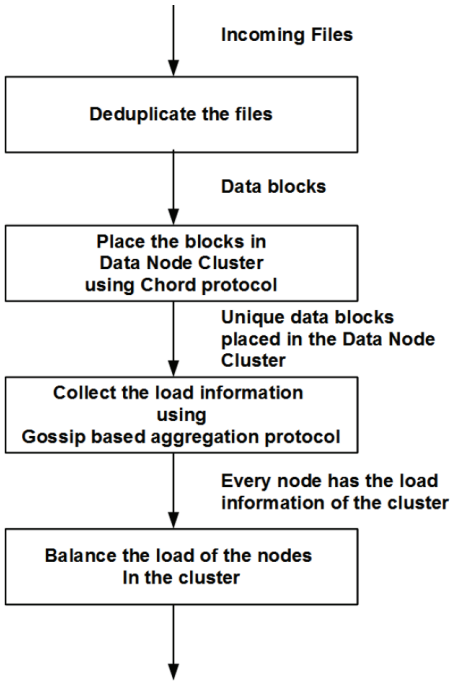


Figure 1. Working principle of distributed load balancing approach.

cluster. These techniques improve the performance of the cluster only when the workload comprises a large number of files with highly similar content among them.

Another existing research work proposed an identifier (ID) reassignment-based distributed load balancing solution [5] by utilizing a distributed hash table (DHT) for a data node cluster in the Hadoop distributed file system (HDFS). In this solution, initially, the lightly loaded node sheds its load to its successor. Thereafter, it rejoins as the predecessor for the heavily loaded node, with a new ID to receive load from it. However, the shedding of data causes an additional overhead besides the overhead due to the migration of data blocks during load balancing. This approach thus incurs a higher migration overhead.

Therefore, it is desirable to devise a load balancing strategy that will minimize both metadata updates as well as migration overhead for the non-backup workload. To this end, this paper proposes the following strategies:

- Location-independent fingerprint index and enriched file recipe to reduce metadata update overhead;
- A virtual server-based load balancing algorithm to reduce migration overhead, where the virtual server is the unit of load balancing.

Figure 1 explains the sequence of steps involved in the proposed distributed load balancing strategy. In this context, every incoming file is divided into a set of fixed or variable sized blocks. They are deduplicated and forwarded to the data node cluster. The data node cluster consists of commodity machines organized in a DHT network where every node implements the chord protocol [6,7]. The deduplicated blocks are placed in the data node cluster as per the chord protocol. Once the cluster is in use over a period of time, there may possibly be some level of load imbalance in the cluster. Hence, it is necessary to balance the cluster periodically to improve its performance. As a part of load balancing, initially, the load information is collected by utilizing

the gossip-based aggregation protocol [8]. Subsequently, every node possesses the load information of the entire cluster. Based on the load status of the individual nodes, data are migrated from the heavily loaded to the lightly loaded nodes. The rest of the paper is organized as follows. Section 2 discusses research works related to deduplication and load balancing. Sections 3 and 4 describe the proposed location-independent fingerprint index and the virtual server-based load balancing algorithm, respectively. Section 5 discusses the implementation in detail and substantiates the proposed strategies with detailed performance analysis. Section 6 concludes the paper.

2. Related work

Two existing strategies, namely content-aware load balancing and probabilistic deduplication, place the blocks of every incoming stream into a suitable node that hosts a large number of similar blocks. These approaches maximize the deduplication efficiency and minimize the network bandwidth utilization while retaining the balance in the cluster. However, these approaches are suitable only for workloads that comprise a large number of similar files.

Xu et al. [9] proposed a scalable hybrid hash cluster to maintain the fingerprint index using a DHT. This reduces the latency of the hash lookup process during duplicate detection. Scalable dedupe [10] partitions the incoming data stream based on the k-least significant bits of their corresponding fingerprints. These partitioned blocks and the fingerprints are mapped to the respective nodes using a DHT. This enables the deduplicated storage to identify the duplicate blocks by searching the fingerprints in parallel. However, these solutions have utilized the DHT either to improve the deduplication process by searching in parallel or to reduce the latency of the hash lookup process. More importantly, DHT was not utilized in this work for load balancing.

Bhagwat et al. [11] utilized a hierarchical index to maintain the fingerprint entries for the blocks corresponding to similar files. This approach facilitates faster detection of duplicates. Fu et al. [12] applied different chunking mechanisms for different types of files in the workload. Furthermore, they suggested maintaining separate indices for different types of files. In these approaches, the fingerprint index and the data blocks are maintained separately. In addition, every fingerprint entry in the fingerprint index involves location information corresponding to a block. Hence, adopting these solutions in deduplicated storage will incur substantial MUO during load balancing. Hsiao et al. proposed an ID reassignment-based load rebalancing approach in a distributed environment that was implemented over HDFS. In this approach, all the data nodes are placed in a peer-to-peer (P2P) fashion, with each of them implementing the chord protocol. This work exploits the self-configurable and self-healing characteristic of the chord protocol to implement the distributed load balancing algorithm. However, this approach did not employ the deduplication technique. Xu et al. [13] proposed a mechanism to build a reliable deduplicated storage. They proposed an even data placement (EDP) algorithm for the fair distribution of deduplicated blocks. Furthermore, they utilized an erasure coding mechanism to improve the reliability of the storage.

The works in [14] and [15] proposed a data routing algorithm that aims to route the new data blocks to the data server, which already hosts more similar data blocks. In this approach, every data server maintains a set of representative fingerprints corresponding to the segments of the files. Whenever a new segment has to be written, its representative fingerprint will be first checked against the set that is present in every data server to calculate its similarity value. In order to balance the load, the new segment will be moved to the data server for which it has a higher similarity value.

It becomes clear from the above discussion that the existing deduplicated storage, which caters to non-

backup workloads, maintains the fingerprint index and the data blocks separately. Two shortcomings of these approaches are that the fingerprint index is location-dependent and that ID reassignment-based load balancing requires considerable migration overhead. Hence, a location-independent fingerprint index and a virtual server-based load balancing algorithm have been proposed for deduplicated storage to minimize both the metadata update and the data migration overheads.

3. Proposed location-independent fingerprint index and enriched file recipe

The proposed deduplicated storage consists of a dedupe engine, a metadata node, and a data node cluster. The dedupe engine divides every incoming file into a set of fixed or variably sized blocks and finds their fingerprints. These fingerprints are compared against the entries in the index available in the metadata node to detect duplicates. The unique blocks corresponding to that file alone are sent to the data node cluster for storing them. This process is illustrated in Figure 2.

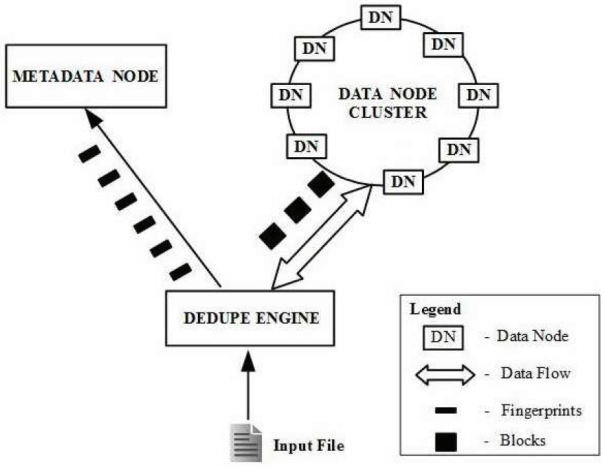
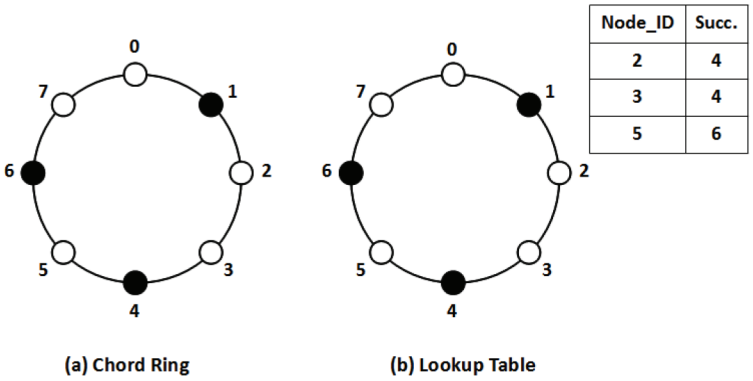


Figure 2. Design of deduplicated storage.

The nodes in the data node cluster are organized as a DHT network, in which each of these nodes implements a DHT protocol, namely Chord. In the data node cluster, the IP address of each physical node is hashed to obtain the node identifier. This identifier is mapped onto an address in the logical N-address space chord ring. Similarly, the hash value of the file path of any incoming file is found out and is labeled as the file identifier. The unique blocks of every incoming file are appropriately placed by mapping the file identifier to a corresponding node identifier using the lookup table. This lookup table keeps track of the locations of the blocks by periodical updates.

Figure 3a shows an 8-node chord ring where three nodes are mapped onto the logical ring with the addresses 1, 4, and 6. For every node, there is a successor and a predecessor. For example, nodes 4 and 6 are the successor and predecessor, respectively, for node 1. Nodes 1, 4, and 6 are responsible for the address spaces (7, 0, 1), (2, 3, 4), and (5, 6), respectively. Any incoming file is placed in a relevant node based on its file identifier. In order to locate the data in such a cluster, every node maintains a lookup table.

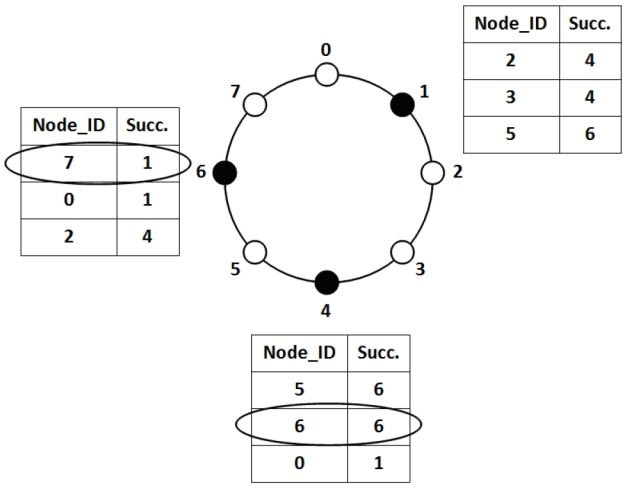
In a 2^m logical ring, a lookup table contains information about m entries where an entry corresponds to a node identifier, $(n + 2^k)$ mod 2^m, $0 \leq k < m$, and its successor. For example, the lookup table of node 1 has 3 entries, namely [((1 + 2^0) mod 2^3), 4], [((1 + 2^1) mod 2^3), 4], and [((1 + 2^2) mod 2^3), 6], as shown in Figure 3b. Let us assume that the user requests a file with file identifier '7' from node 4. Node 4 first searches



Node_ID	Succ.
2	4
3	4
5	6

Figure 3. Chord example.

for the file within itself. As it is not available, it consults the resident lookup table for an entry closer to the file identifier (circled entry) to get its successor node. As the file is not available in node 6 also, the node to be searched (i.e. node 1) is obtained through its lookup table. At this point, the file with file identifier '7' is found in node 1. This is depicted in Figure 4.



Node_ID	Succ.
2	4
3	4
5	6

Node_ID	Succ.
7	1
0	1
2	4

Node_ID	Succ.
5	6
6	6
0	1

Figure 4. Location of data in chord ring.

If a node 'j' joins the chord ring, it is mapped to an address. Let us assume that, after node 'j' joins the chord ring, it is responsible for holding the files corresponding to the address space 'a'. The files that belong to address space 'a' would have been placed in the successor of node 'j' previously. Once node 'j' joins, the successor of 'j' moves these files to node 'j'. Similarly, if node 'k' leaves the ring, it sheds the files that it is currently responsible for to its successor.

Such frequent arrival and departure of the nodes in a P2P network is referred to as churn. This kind of churn is more pronounced in general P2P networks because these networks are formed using voluntary nodes. As the set of voluntary nodes keeps changing dynamically in these networks, there is a significant overhead associated with the movement of data. However, the nodes utilized for building the deduplicated storage are dedicated for this purpose. Hence, this churn will be minimal in the context of deduplicated storage.

In the chord protocol, nodes contact each other periodically to update the status of the cluster dynam-

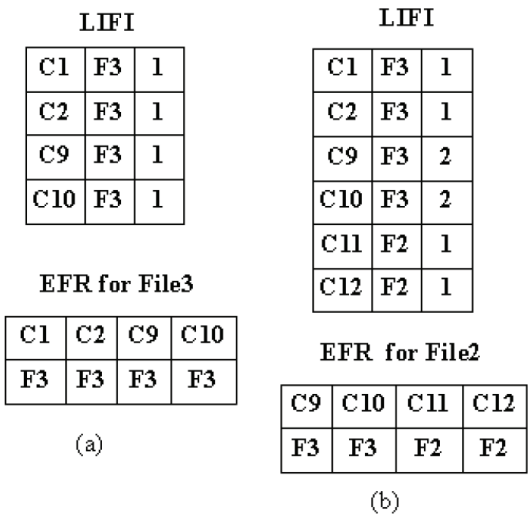


Figure 5. Location-independent fingerprint index and enriched file recipe.

ically. Hence, the lookup table in each node is always up-to-date. To cater to the deduplicated storage, this protocol is suitably modified to place only the unique blocks of any incoming file.

A traditional deduplication system maintains metadata, namely the fingerprint index and file recipe. Fingerprint information in the fingerprint index is utilized to detect duplicates. Similarly, both the location information corresponding to the data blocks in the fingerprint index and the order of the data blocks that is available in the file recipe are required to reconstruct any specific file in the deduplicated storage. However, as the data node cluster of the deduplicated storage built in this paper utilizes the DHT, the file identifier, which is maintained in the lookup table, is responsible for placing and retrieving the unique blocks corresponding to a file. Furthermore, even when the blocks migrate from one node to another due to an addition or a deletion of a node, the lookup table is suitably updated to reflect the changes periodically. Hence, it can be deduced that it is no longer necessary to maintain location information in the fingerprint index to keep track of the location of data blocks.

Based on this inference, a location-independent fingerprint index (LIFI) that exploits the availability of the lookup table in the Chord protocol is proposed in the present work. Every block has an entry in the LIFI that consists of a fingerprint, its file identifier, and the reference count, which indicates the number of files sharing this block. Since the blocks are placed in the data node cluster using file identifiers, they are required to retrieve a file. Though the information pertaining to the file identifiers can be retrieved from LIFI, the computational overhead is quite high owing to the large size of the LIFI. Hence, an enriched file recipe (EFR) has been designed to maintain the fingerprints and file identifiers corresponding to each file for reconstructing the file. These index structures are maintained in the metadata node of the deduplicated storage.

The following example illustrates the proposed indices LIFI and EFR. When a File3, which consists of blocks C1, C2, C9, and C10, needs to be written into the data node cluster of the deduplicated storage, the file identifier (i.e. F3) for File3 is found. Since these blocks are unique to File3, new entries are created in the LIFI and EFR with their corresponding file identifiers as shown in Figure 5a. After the updating of the LIFI and EFR, these unique blocks are written into the data node cluster using file identifier F3. Similarly, when another file, File2, that consists of the blocks C9, C10, C11, and C12 arrives, the updates in the LIFI and EFR are as shown in Figure 5b.

Whenever there is a read request for File2, the file identifiers are first obtained from the EFR of that file. Subsequently, these file identifiers are used to retrieve the blocks from the data node cluster to reconstruct the file. Let us assume that the blocks of File2 are placed in two nodes, namely nodes 4 and 6. For example, C9 and C10 are placed in node 4 using file identifier F3 and C11 and C12 are placed in node 6 using file identifier F2. A read request for File2 first gets the file identifiers (F3, F2) from its corresponding EFR. These file identifiers are used to retrieve the blocks from the data node cluster using the lookup table. These blocks are assembled later based on the order indicated in the EFR to reconstruct File2.

4. Proposed virtual server-based load balancing algorithm

In general, a cluster will be made up of several nodes. The load of a particular node in the cluster is referred to as the amount of current usage of physical storage that belongs to that node. The overall capacity (λ) and overall load (μ) refer to the aggregation of the maximum physical storage and the amount of current usage of storage across all the nodes in the cluster respectively. Over a period of utilization, load imbalance might probably arise among the nodes that belong to the cluster. Hence, it is essential to find the ideal load that a node can handle with respect to the cluster to which it belongs. The ideal load of a node can be found using the following equation:

$$\Upsilon = \frac{\mu}{\lambda} * C, \tag{1}$$

where λ is the overall capacity, μ is the overall load of a cluster, κ is the current load, and C is the maximum physical storage capacity of a node.

In a load balancing process, it is essential for each node to know whether it is lightly or heavily loaded at any specific time in order to maintain balance in the cluster. This can be found using the following formula:

$$\theta = \begin{cases} \text{Lightly loaded}, & \text{if } \kappa < 0.8 * \Upsilon \\ \text{Moderately loaded}, & \text{if } \kappa \geq 0.8 * \Upsilon \text{ and } \kappa < 1.2 * \Upsilon \\ \text{Heavily loaded}, & \text{if } \kappa \geq 1.2 * \Upsilon \end{cases} \tag{2}$$

As there is no centralized process in the data node cluster to collect the load information about every node, a distributed gossip-based aggregation protocol has been used. It has several aggregate functions, namely summation, average, mean, and median. The data node cluster utilizes the sum aggregate function of this protocol for finding the overall load (μ) of the cluster as illustrated in Figure 6.

Consider an 8-node cluster as shown in Figure 6. Here, a square represents a node, letters represent the names of nodes, and integer values inside the squares represent the current load pertaining to each node. First, a node randomly selects another node. For example, node 'A' selects node 'G' randomly to exchange current load information between them. Hence, nodes 'A' and 'G' are represented by the same color. Similarly, the same colored nodes (B, H), (C, F), and (D, E) communicate with each other for exchanging their load information as shown in Figure 6a. At the end of the first cycle, the same colored nodes are updated with the aggregated load information and this is shown in Figure 6b. Figure 6c depicts that nodes with the same color are not allowed to communicate with each other in the next cycle.

In the second cycle, a node once again chooses another node that it did not contact earlier. For example, node A can contact any other node other than G. Let us assume that node A chooses node F, and they both

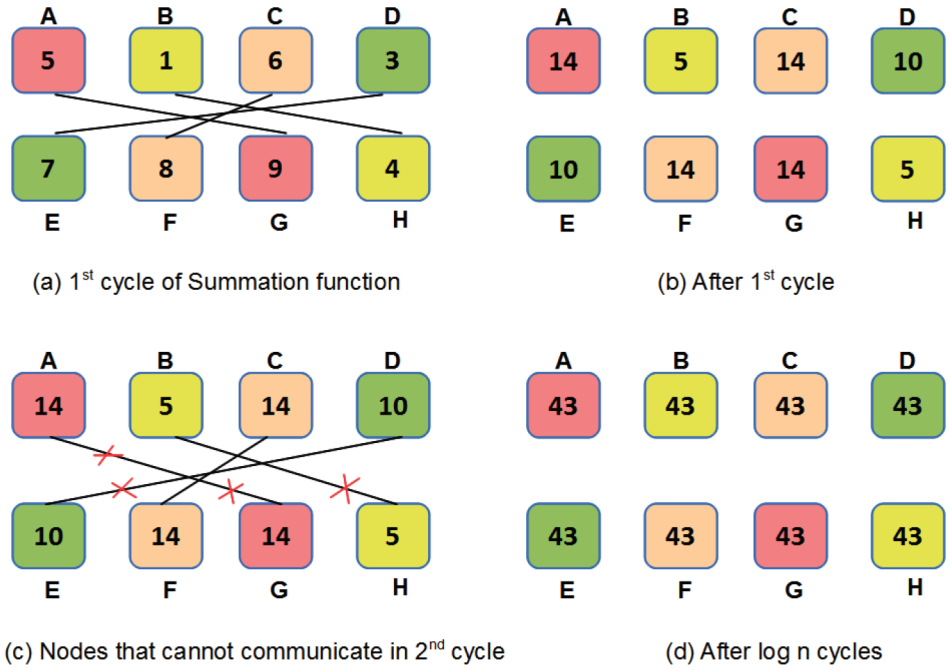


Figure 6. Gossip-based aggregation.

exchange their current load information. As nodes G and C were contacted by A and F in the previous cycle, respectively, the nodes (A, C, F, G) are updated with the new sum 28. Similarly, nodes B, D, E, and H are updated with the sum 15. In the next cycle, all the nodes are updated with the value of 43 as shown in Figure 6d. The overall capacity (λ) of the cluster also has been determined in a similar fashion. While nodes exchange their current load, other information such as the node capacity and available network bandwidth are also exchanged.

In DHT-based load balancing, a load is balanced based on two methods. In one approach, the lightly loaded node initiates the transfer of data. In this case, first, the lightly loaded node sheds its data to its successor and rejoins the chord ring as a predecessor of the heavily loaded node. This is equivalent to the node *join* operation in the chord. Hence, the predecessor gets the load of the successor, which is a heavily loaded node. Similarly, a change in the location of the data is automatically updated in the lookup table during periodic exchange of information in the chord. This approach is referred to as ID reassignment-based load balancing. However, as it involves data shedding, it results in considerable migration overhead.

The second approach, namely virtual server-based load balancing, uses the idea of maintaining multiple partitions of a DHT's address space in a single node. Within the chord system, one virtual server is responsible for maintaining data with respect to a specific interval of address space. In this context, each virtual server is viewed as an independent node. Furthermore, the displacement of virtual servers among the arbitrary nodes is similar to the standard *join* or *leave* operations in a chord ring. Hence, there is a periodical exchange of information between the virtual servers residing in the physical nodes that constitute the chord ring to reflect the current updates. During the load balancing process, a suitable virtual server from a heavily loaded node is migrated to a lightly loaded node along with its data. However, if the number of virtual servers increases in a node, it might adversely affect the performance of that node due to the increase in bandwidth utilization

Algorithm: Data migration algorithm initiated by heavily loaded node.

Input:
Aggregated load information (*info*) about all the
nodes by Gossip Protocol
ToMigrateLoad corresponds to the load of a virtual server

Output: Suitable lightly loaded node is chosen

```
w1 = 0.2      // Weight corresponding to load
w2 = 0.3      // Weight corresponding to capacity
w3 = 0.5      // Weight corresponding to
                 network bandwidth
// Calculation of weights for each lightly loaded node
for every node do
    Retrieve load, capacity, n/w bandwidth from Info
    Assign the load status to every node as light, moderate, or heavy
    Calculate weight as w1*load+w2*capacity+w3*n/w bandwidth
Sort both the lightly and moderately loaded nodes in descending order
for every lightly loaded node do
    if (currentLoad+ToMigrateLoad) <idealLoad then
        currentLoad = currentLoad + ToMigrateLoad
        Transfer data to lightly loaded node
if there are no lightly loaded nodes then
    Find the average weight (AV) of moderately loaded nodes
    while ToMigrateLoad ≥ 0 do
        for every moderately loaded node do
            if (currentLoad+(AV*ToMigrateLoad)) <idealLoad then
                currentLoad = currentLoad + AV*ToMigrateLoad
                Transfer data to moderately loaded node
        ToMigrateLoad = ToMigrateLoad − (AV ∗ ToMigrateLoad)
```

arising from the periodic exchange of information. Hence, the choice of a lightly loaded node must be made with respect to network utilization also in addition to its current load and capacity.

In the context of deduplicated storage, each node has the information regarding the current load, capacity, and network utilization of every other node since the gossip-based aggregation protocol has been utilized. Hence, it is possible for every node to find the load status (lightly, moderately, or heavily loaded) of every other node in the system. Based on this, a data migration algorithm, illustrated in Algorithm, is proposed in the present work. In this algorithm, each node finds the ideal load of all the other nodes by using the expression in Eq. (1). Lightly and moderately loaded nodes are identified by using Eq. (2) and their weights are then determined. Once this is completed, both the lightly and moderately loaded nodes are arranged in descending order according to their weights. A heavily loaded node paired with a suitable lightly loaded node is used to transfer a virtual server without making that light node heavy. In case no lightly loaded nodes are available, suitable virtual servers can be moved from heavily loaded nodes to one or more moderately loaded nodes.

5. Implementation

The distributed load balancing technique has been implemented in the deduplicated storage that has been built using commodity machines (Intel Core i7 2.93GHZ, 8GB RAM DDR3, 500 GB HDD). One of these machines acts as both dedupe engine and metadata server. The proposed LIFI has been implemented using an on-disk hash table in the metadata server. Furthermore, an open source chord GUI implementation has been utilized to implement the data node cluster of the deduplicated storage. The existing chord implementation has been suitably modified to receive only the unique blocks with respect to any incoming file. Furthermore, a gossip-based aggregation protocol and Algorithm 1 have been incorporated to ensure load balance in the cluster.

A workload of size 50 GB consisting of different types of files (.pdf, .mp3, .avi, .doc, and .ppt, to name a few) with sizes varying from several KBs to a few MBs has been kept in the input folder. The files from this folder are deduplicated and the blocks are written into the nodes organized in a P2P fashion in the deduplicated storage for a certain predetermined time, and relevant experiments have been conducted.

5.1. Evaluation

This section describes the performance of the proposed LIFI and the virtual server-based load balancing algorithm. It also provides a comparison with the existing fingerprint index and ID reassignment-based load balancing approach using a set of common evaluation metrics.

5.2. Metrics for evaluation

Three performance metrics, namely metadata update overhead, migration overhead, and load status have been utilized to analyze the behavior of the proposed index and the load balancing technique.

The number of location information updates in the fingerprint index is the metadata update overhead. Since this parameter impacts the performance of the deduplicated storage, the proposed location-independent fingerprint index and the existing fingerprint index have been evaluated using this performance metric.

Migration overhead is the amount of data transferred from a heavily loaded to a lightly loaded node. Furthermore, the load status for every node in the cluster has been observed. Since these parameters affect the performance of any storage system, the virtual server-based and ID reassignment-based load balancing approaches have been assessed using these parameters.

5.3. Results and discussion

Multiple experiments have been conducted to provide a thorough comparison of the various load balancing approaches with respect to the chosen evaluation metrics.

5.3.1. Impact of the proposed LIFI on metadata update overhead

In order to study the impact of metadata update overhead, the cluster has been populated with the files from the dataset for a certain duration. Furthermore, an ideal load threshold has been suitably set to simulate the load imbalance in the cluster. In the existing fingerprint index, the metadata will be updated for every block that is migrated. A variable count has been incremented for every block that is migrated to measure the metadata update overhead.

In the proposed work, the fingerprint index is made location-independent due to the presence of lookup table in the chord protocol. Hence, when there are migrations of files between the nodes, the lookup table will be updated as it maintains file identifiers with respect to files. The *fixFingers()* method of chord protocol is

responsible for updating the lookup table. A variable in this method is used to keep track of the metadata update. The metadata updates with respect to the existing fingerprint index and location-independent fingerprint index were measured and plotted as shown in Figure 7. Since the metadata maintained for the files are less when compared to that of blocks of files, the metadata update overhead incurred by the LIFI is comparatively less than that of the existing fingerprint index.

The metadata of this proposed work include the LIFI and EFR. These were maintained in only one machine in our testbed environment. Hence, there is no duplication of metadata. Since there are no duplications, there will not be any possibility of inconsistency in the metadata updates. Hence, whenever the metadata server is contacted, it will always give the correct updated metadata, which will be used for the file access.

5.3.2. Computational load of chord protocol

The proposed LIFI of deduplicated storage capitalizes on the lookup table of the chord protocol to eliminate the location information from the fingerprint index. Hence, it involves time for sending management messages across the nodes. These messages are responsible for the detection of liveness of nodes and also the update of the lookup table due to the addition or deletion of nodes in the topology. It sends *log n* heartbeat messages periodically across the nodes for managing the network. The *notify()* method of the chord protocol is responsible for sending the heartbeat messages. In the present experiments, the periodicity has been set to 1 min.

In order to study the computational load, the cluster has been populated with the files from the dataset for a certain duration. Furthermore, an ideal load threshold has been suitably set to simulate the load imbalance in the cluster. In order to bring load balance in the cluster, a few nodes may leave and join. The time taken for sending the liveness and update messages is measured in the *notify()* method. Whenever this method is called, this measured time is aggregated and plotted for every hour as shown in Figure 8. These same management messages are utilized also for sending gossip messages. Hence, there is no separate computational overhead that is incurred for the execution of gossip protocol.

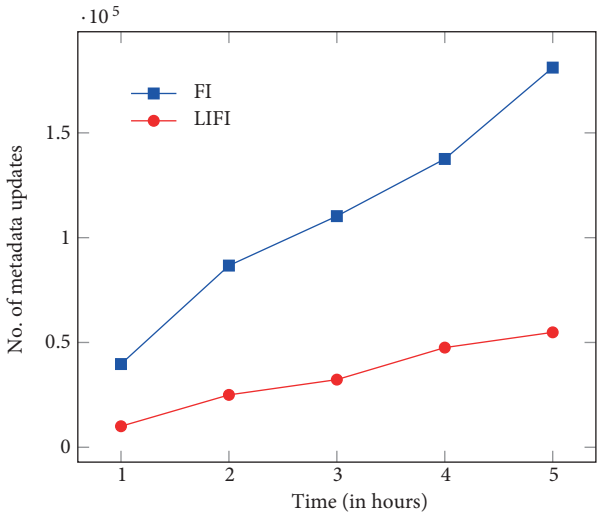


Figure 7. Metadata update overhead

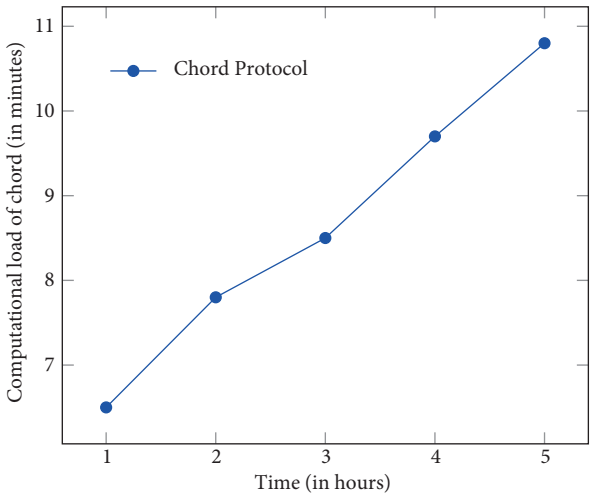


Figure 8. Computational load for chord protocol.

5.3.3. Scalability of gossip-based aggregation protocol

In order to study the scalability of the gossip-based protocol, P2P networks with varying numbers of nodes have been created. In each network, files from the dataset are populated and an ideal load threshold is suitably set to simulate the load imbalance scenario. The gossip-based protocol consumes *log n* cycles to retrieve the aggregated load information of all the *'n'* nodes. The time taken to propagate the load information is measured in the *notify()* method and plotted in Figure 9. It can be seen that the propagation time is less than 1 h even when the number of nodes is as high as 100.

5.3.4. Impact of the virtual server-based load balancing algorithm on migration overhead

In order to measure the migration overhead, both the virtual server-based and ID reassignment-based load balancing approaches have been implemented. Both these approaches have utilized the gossip-based aggregation protocol to collect the information about every node. This incurs no additional overhead as it utilizes only the periodic exchange of chord management messages.

The cluster has been populated with a random load at a specific time. These algorithms have been executed 5 times and the average number of data blocks migrated between the heavily and the lightly loaded nodes is plotted in Figure 10. In the ID reassignment-based load balancing algorithm, a lightly loaded node initiates the transfer of data migration. In this case, the lightly loaded node sheds its data blocks to its successor and rejoins as the predecessor of the heavily loaded node to receive the load from it. Subsequently, the actual data transfer happens to maintain the balance. However, in the virtual server-based load balancing algorithm, each node knows the status of every other peer in the cluster as it utilizes the gossip-based aggregation protocol. It selects the best suited lightly loaded node to transfer the virtual server in order to lighten the heavy node without any additional shedding. Hence, it involves less migration overhead.

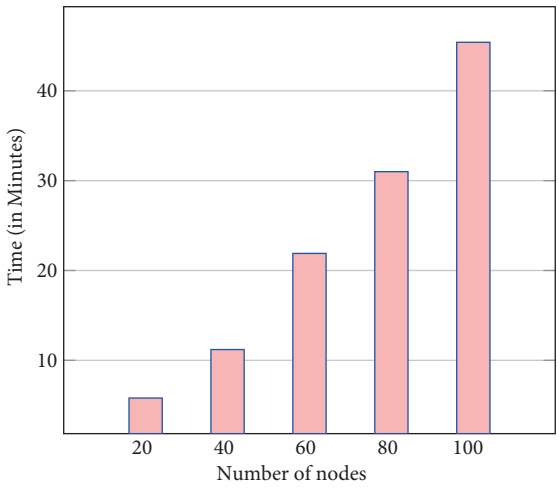


Figure 9. Information propagation time.

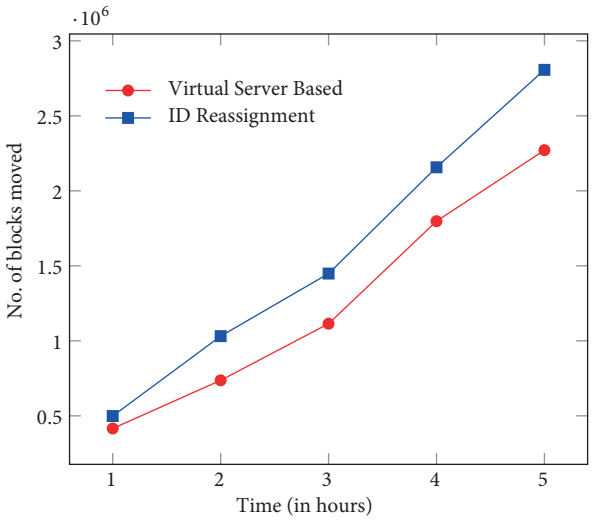


Figure 10. Migration overhead.

5.3.5. Impact of virtual server-based load balancing algorithm on load status

In order to study the impact of the existing and the proposed load balancing algorithms on load status, a cluster has been set up with 25 nodes. Each node has been assumed to have the capacity of 2 GB. The deduplicated

blocks are sent to the cluster from the workload. Further, the above-mentioned algorithms have been executed and the load status for every node is recorded. The current load in the cluster is found to be 12 GB. The ideal load for every node in the cluster is calculated as 500 MB using Eq. (1). If the load of a node is greater than 600 MB, then that node is considered as a heavily loaded node. Similarly, if the load of a node is less than 400 MB, then such a node is considered as a lightly loaded node. If the load of a node is between 400 MB and 600 MB, it is viewed as a moderately loaded node. The lightly, moderately, and heavily loaded nodes are shown in blue, black, and red, respectively. The load status of the cluster before and after the load balancing process for the two algorithms are plotted in Figures 11 and 12, respectively.

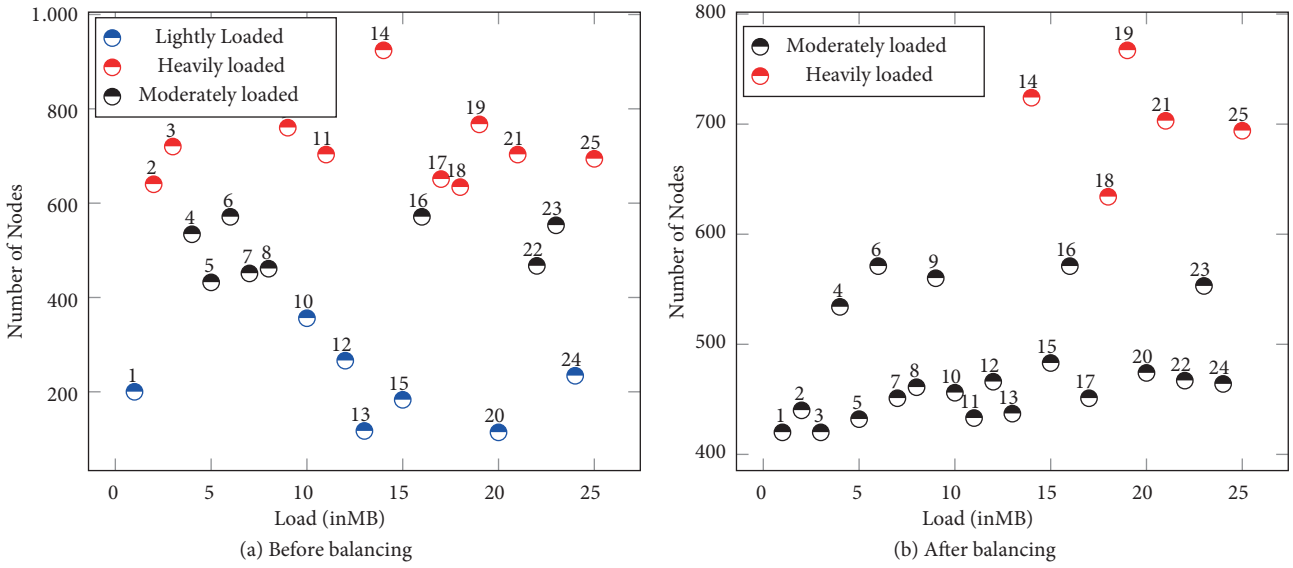


Figure 11. Load status of the cluster using ID reassignment-based load balancing approach.

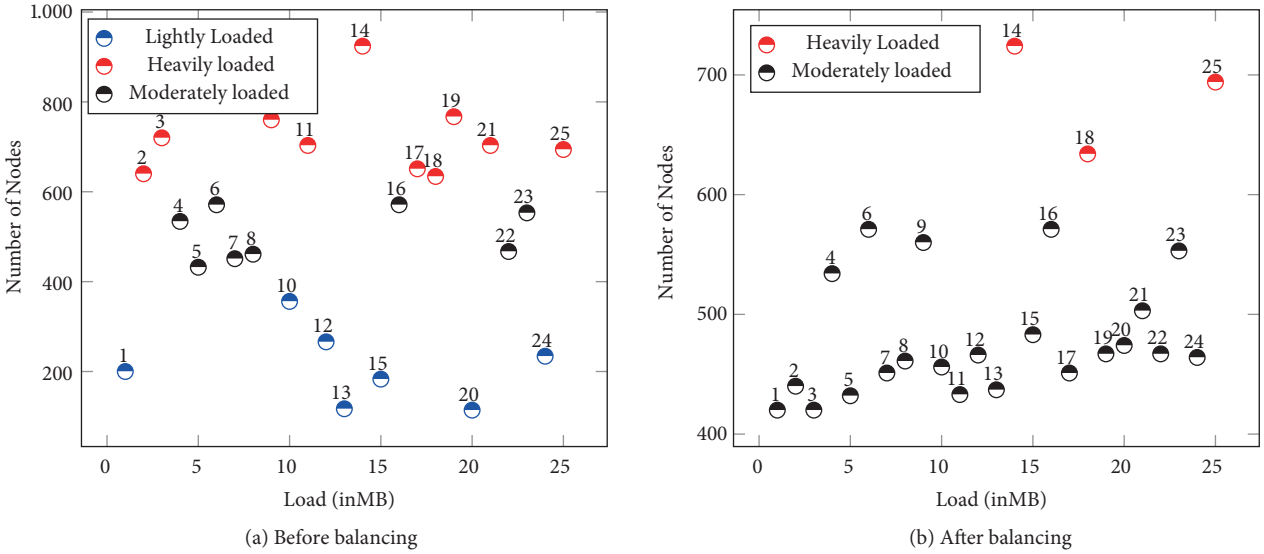


Figure 12. Load status of the cluster using virtual server-based load balancing approach.

Since the proposed algorithm takes care of migrating the load to moderately loaded nodes in the absence of lightly loaded nodes, the number of heavily loaded nodes is reduced in Figure 12 when compared to Figure 11. Hence, the load status of the cluster for the virtual server-based load balancing approach is comparatively better than that for the ID reassignment-based load balancing approach.

6. Conclusions

A deduplicated cloud storage has been built using commodity machines. A dedupe engine and the proposed LIFI have been implemented in one of the commodity machines. Furthermore, an open source implementation of a GUI-based chord protocol has been utilized to build the data node cluster. The existing chord protocol has been suitably modified to store only the unique blocks of every incoming file. The gossip-based aggregation protocol and the virtual server-based load balancing algorithm have been implemented over the chord protocol.

The data node cluster is populated with a random number of documents, which are of random sizes at specific time intervals. The proposed LIFI and the existing fingerprint index have been evaluated with respect to the metadata update overhead. Virtual server-based and ID reassignment-based load balancing algorithms have been evaluated with respect to the migration overhead. The results clearly demonstrate that the proposed LIFI reduces the metadata update overhead by 74% as opposed to the existing fingerprint index. Furthermore, the virtual server-based load balancing algorithm reduces the migration overhead by 33% when compared to the ID reassignment-based load balancing algorithm.

References

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

[2] Wu J, Hua Y, Zuo P, Sun Y. Improving restore performance in deduplication systems via a cost-efficient rewriting scheme. IEEE Transactions on Parallel and Distributed Systems 2019; 30 (1): 119-132. doi: 10.1109/TPDS.2018.2852642

[3] Douglis F, Bhardwaj D, Qian H, Shilane P. Content-aware load balancing for distributed backup. In: Proceedings of 25th Conference on Large Installation System Administration; Boston, MA, USA; 2011. pp. 1–18.

[4] Frey D, Kermarrec AM, Kloudas K. Probabilistic deduplication for cluster-based storage systems. In: Proceedings of ACM Cloud Computing Symposium; San Jose, CA, USA; 2012. pp. 1–17.

[5] Hsiao HC, Liao H, Chen ST, Huang KC. Load rebalancing for distributed file systems in clouds. IEEE Transactions on Parallel and Distributed Systems 2013; 24 (5): 951–962. doi: 10.1109/TPDS.2012.196

[6] Stoica I, Morris R, Liben-Nowell D, Karger DR, Khaashoek MF et al. Chord: A scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Transactions on Networking 2003; 11 (1): 17–32. doi: 10.1109/TNET.2002.808407

[7] Brunel J, Chemouil D, Tawa J. Analyzing the fundamental liveness property of the chord protocol. In: Proceedings of IEEE International Conference on Formal Methods in Computer Aided Design; Austin, TX, USA; 2018. pp. 1-9.

[8] Jelasity M, Montresor A, Babaoglu O. Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems 2005; 23 (3): 219–252. doi: 10.1145/1082469.1082470

[9] Xu L, Hu J, Mkandawire S, Jiang H. SHHC: A scalable hybrid hash cluster for cloud backup services in data centers. In: Proceedings of 33rd IEEE International Conference on Distributed Computing System Workshops; Philadelphia, PA, USA; 2013. pp. 61–65.

[10] Santos W, Teixeira T, Machado C, Meira W Jr, Ferreira R et al. A scalable parallel deduplication algorithm. In: Proceedings of 19th IEEE International Symposium on Computer Architecture and High Performance Computing; Rio Grande do Sul, Brazil; 2007. pp. 79–86.

[11] Bhagwat D, Eshghi K, Long DDE, Lillibridge M. Extreme binning: scalable, parallel deduplication for chunk-based file backup. In: Proceedings of IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems; London, UK; 2009. pp. 1–9.

[12] Fu Y, Jiang H, Xiao N, Tian L, Liu F. Aa-dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In: Proceedings of IEEE International Conference on Cluster Computing; Austin, TX, USA; 2015. pp. 112–120.

[13] Xu M, Zhu Y, Lee PP, Xu Y. Even data placement for load balance in reliable distributed deduplication storage systems. In: Proceedings of 23rd IEEE International Symposium on Quality of Service; Portland, OR, USA; 2015. pp. 349-358.

[14] Huang Z, Li H, Li X, He W. SS-dedup: A high throughput stateful data routing algorithm for cluster deduplication system. In: Proceedings of IEEE International Conference on Big Data; Washington, DC, USA; 2016. pp. 2991-2995.

[15] Luo S, Zhang G, Wu C, Khan S, Li K, Boafft: Distributed deduplication for big data storage in the cloud. IEEE Transactions on Cloud Computing 2018; 61 (11): 1-13. doi: 10.1109/TCC.2015.2511752