Load Rebalancing for Distributed File Systems in Clouds

Hung-Chang Hsiao, *Member*, *IEEE Computer Society*, Hsueh-Yi Chung, Haiying Shen, *Member*, *IEEE*, and Yu-Chang Chao

Abstract—Distributed file systems are key building blocks for cloud computing applications based on the MapReduce programming paradigm. In such file systems, nodes simultaneously serve computing and storage functions; a file is partitioned into a number of chunks allocated in distinct nodes so that MapReduce tasks can be performed in parallel over the nodes. However, in a cloud computing environment, failure is the norm, and nodes may be upgraded, replaced, and added in the system. Files can also be dynamically created, deleted, and appended. This results in load imbalance in a distributed file system; that is, the file chunks are not distributed as uniformly as possible among the nodes. Emerging distributed file systems in production systems strongly depend on a central node for chunk reallocation. This dependence is clearly inadequate in a large-scale, failure-prone environment because the central load balancer is put under considerable workload that is linearly scaled with the system size, and may thus become the performance bottleneck and the single point of failure. In this paper, a fully distributed load rebalancing algorithm is presented to cope with the load imbalance problem. Our algorithm is compared against a centralized approach in a production system and a competing distributed solution presented in the literature. The simulation results indicate that our proposal is comparable with the existing centralized approach and considerably outperforms the prior distributed algorithm in terms of load imbalance factor, movement cost, and algorithmic overhead. The performance of our proposal implemented in the Hadoop distributed file system is further investigated in a cluster environment.

Index Terms—Load balance, distributed file systems, clouds

1 Introduction

Cloud Computing (or *cloud* for short) is a compelling technology. In clouds, clients can dynamically allocate their resources on-demand without sophisticated deployment and management of resources. Key enabling technologies for clouds include the MapReduce programming paradigm [1], distributed file systems (e.g., [2], [3]), virtualization (e.g., [4], [5]), and so forth. These techniques emphasize scalability, so clouds (e.g., [6]) can be large in scale, and comprising entities can arbitrarily fail and join while maintaining system reliability.

Distributed file systems are key building blocks for cloud computing applications based on the MapReduce programming paradigm. In such file systems, nodes simultaneously serve computing and storage functions; a file is partitioned into a number of chunks allocated in distinct nodes so that MapReduce tasks can be performed in parallel over the nodes. For example, consider a *wordcount* application that counts the number of distinct words and the frequency of each unique word in a large file. In such an application, a cloud partitions the file into a large number of disjointed and fixed-size pieces (or *file chunks*) and assigns them to different cloud storage nodes (i.e., chunkservers). Each storage node (or *node* for short) then calculates the frequency of each unique word by scanning and parsing its local file chunks.

In such a distributed file system, the load of a node is typically proportional to the number of file chunks the node possesses [3]. Because the files in a cloud can be arbitrarily created, deleted, and appended, and nodes can be upgraded, replaced and added in the file system [7], the file chunks are not distributed as uniformly as possible among the nodes. Load balance among storage nodes is a critical function in clouds. In a load-balanced cloud, the resources can be well utilized and provisioned, maximizing the performance of MapReduce-based applications.

State-of-the-art distributed file systems (e.g., *Google GFS* [2] and *Hadoop HDFS* [3]) in clouds rely on central nodes to manage the metadata information of the file systems and to balance the loads of storage nodes based on that metadata. The centralized approach simplifies the design and implementation of a distributed file system. However, recent experience (e.g., [8]) concludes that when the number of storage nodes, the number of files and the number of accesses to files increase linearly, the central nodes (e.g., the master in Google GFS) become a performance bottleneck, as they are unable to accommodate a large number of file accesses due to clients and MapReduce applications. Thus,

- *H.-C. Hsiao is with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 70101, Taiwan. E-mail: hchsiao@csie.ncku.edu.tw.*
- *H.-Y. Chung is with the Department of Computer Science and Information Engineering, Distributed Computing Research Laboratory, National Cheng Kung University, Tainan 70101, Taiwan. E-mail: p7697138@mail.ncku.edu.tw.*
- *H. Shen is with the Holcombe Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634. E-mail: shenh@clemson.edu.*
- *Y.-C. Chao is with the Cloud Platform Technology Department, Cloud Service Application Center, Industrial Technology Research Institute South Campus, Tainan 709, Taiwan. E-mail: ycchao@itri.org.tw.*

Manuscript received 18 Jan. 2012; revised 27 May 2012; accepted 7 June 2012; published online 22 June 2012.
Recommended for acceptance by H. Jiang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-01-0037.
Digital Object Identifier no. 10.1109/TPDS.2012.196.

depending on the central nodes to tackle the load imbalance problem exacerbate their heavy loads. Even with the latest development in distributed file systems, the central nodes may still be overloaded. For example, *HDFS federation* [9] suggests an architecture with multiple *namenodes* (i.e., the nodes managing the metadata information). Its file system namespace is statically and manually partitioned to a number of namenodes. However, as the workload experienced by the namenodes may change over time and no adaptive workload consolidation and/or migration scheme is offered to balance the loads among the namenodes, any of the namenodes may become the performance bottleneck.

In this paper, we are interested in studying the *load rebalancing problem* in distributed file systems specialized for large-scale, dynamic and data-intensive clouds. (The terms "rebalance" and "balance" are interchangeable in this paper.) Such a large-scale cloud has hundreds or thousands of nodes (and may reach tens of thousands in the future). Our objective is to allocate the chunks of files as uniformly as possible among the nodes such that no node manages an excessive number of chunks. Additionally, we aim to reduce network traffic (or *movement cost*) caused by rebalancing the loads of nodes as much as possible to maximize the network bandwidth available to normal applications. Moreover, as failure is the norm, nodes are newly added to sustain the overall system performance [2], [3], resulting in the heterogeneity of nodes. Exploiting capable nodes to improve the system performance is, thus, demanded.

Specifically, in this study, we suggest offloading the load rebalancing task to storage nodes by having the storage nodes balance their loads spontaneously. This eliminates the dependence on central nodes. The storage nodes are structured as a network based on *distributed hash tables* (*DHTs*), e.g., [10], [11], [12]; discovering a file chunk can simply refer to rapid key lookup in DHTs, given that a unique handle (or *identifier*) is assigned to each file chunk. DHTs enable nodes to self-organize and -repair while constantly offering lookup functionality in node dynamism, simplifying the system provision and management.

In summary, our contributions are threefold as follows:

- By leveraging DHTs, we present a load rebalancing algorithm for distributing file chunks as uniformly as possible and minimizing the movement cost as much as possible. Particularly, our proposed algorithm operates in a distributed manner in which nodes perform their load-balancing tasks independently without synchronization or global knowledge regarding the system.
- Load-balancing algorithms based on DHTs have been extensively studied (e.g., [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]). However, most existing solutions are designed without considering both movement cost and node heterogeneity and may introduce significant maintenance network traffic to the DHTs. In contrast, our proposal not only takes advantage of physical network locality in the reallocation of file chunks to reduce the movement cost but also exploits capable nodes to improve the overall system performance. Additionally, our algorithm reduces algorithmic overhead introduced to the DHTs as much as possible.
- Our proposal is assessed through computer simulations. The simulation results indicate that although each node performs our load rebalancing algorithm independently without acquiring global knowledge, our proposal is comparable with the centralized approach in Hadoop HDFS [3] and remarkably outperforms the competing distributed algorithm in [14] in terms of load imbalance factor, movement cost, and algorithmic overhead. Additionally, our load-balancing algorithm exhibits a fast convergence rate. We derive analytical models to validate the efficiency and effectiveness of our design. Moreover, we have implemented our load-balancing algorithm in HDFS and investigated its performance in a cluster environment.

The remainder of the paper is organized as follows: the load rebalancing problem is formally specified in Section 2. Our load-balancing algorithm is presented in Section 3. We evaluate our proposal through computer simulations and discuss the simulation results in Section 4. In Section 5, the performance of our proposal is further investigated in a cluster environment. Our study is summarized in Section 6. Due to space limitation, we defer the extensive discussion of related works in the appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.196.

2 Load Rebalancing Problem

We consider a large-scale distributed file system consisting of a set of *chunkservers* V in a cloud, where the cardinality of V is $|V| = n$. Typically, n can be 1,000, 10,000, or more. In the system, a number of files are stored in the n chunkservers. First, let us denote the set of files as F. Each file $f \in F$ is partitioned into a number of disjointed, fixed-size chunks denoted by C_f. For example, each chunk has the same size, 64 Mbytes, in Hadoop HDFS [3]. Second, the load of a chunkserver is proportional to the number of chunks hosted by the server [3]. Third, node failure is the norm in such a distributed system, and the chunkservers may be upgraded, replaced and added in the system. Finally, the files in F may be arbitrarily created, deleted, and appended. The net effect results in file chunks not being uniformly distributed to the chunkservers. Fig. 1 illustrates an example of the load rebalancing problem with the assumption that the chunkservers are homogeneous and have the same capacity.

Our objective in the current study is to design a load rebalancing algorithm to reallocate file chunks such that the chunks can be distributed to the system as uniformly as possible while reducing the movement cost as much as possible. Here, the *movement cost* is defined as the number of chunks migrated to balance the loads of the chunkservers. Let $\mathcal{A}$ be the ideal number of chunks that any chunkserver $i \in V$ is required to manage in a system-wide load-balanced state, that is,

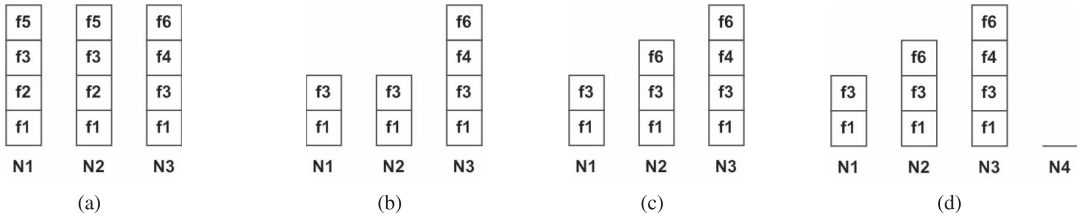


Fig. 1. An example illustrates the load rebalancing problem, where (a) an initial distribution of chunks of six files f_1, f_2, f_3, f_4, f_5, and f_6 in three nodes N_1, N_2, and N_3, (b) files f_2 and f_5 are deleted, (c) f_6 is appended, and (d) node N_4 joins. The nodes in (b), (c), and (d) are in a load-imbalanced state.

$$\mathcal{A} = \frac{\sum_{f \in F} |C_f|}{n}. \tag{1}$$

Then, our load rebalancing algorithm aims to minimize the *load imbalance factor* in each chunkserver i as follows:

$$\|L_i - \mathcal{A}\|, \tag{2}$$

where L_i denotes the load of node i (i.e., the number of file chunks hosted by i) and $\|\cdot\|$ represents the absolute value function. Note that "chunkservers" and "nodes" are interchangeable in this paper.

Theorem 1. *The load rebalancing problem is $\mathcal{NP}$-hard.*

Proof. By *restriction*, an instance of the decision version of the load rebalancing problem is the *knapsack* problem [23]. That is, consider any node $i \in V$. i seeks to store a subset of the file chunks in F such that the number of chunks hosted by i is not more than $\mathcal{A}$, and the "value" of the chunks hosted is at least ϕ, which is defined as the inverse of the sum of the movement cost caused by the migrated chunks. □

To simplify the discussion, we first assume a homogeneous environment, where migrating a file chunk between *any* two nodes takes a unit movement cost and each chunkserver has the identical storage capacity. However, we will later deal with the practical considerations of node capacity heterogeneity and movement cost based on chunk migration in physical network locality.

3 Our Proposal

Table 1 in Appendix B, which is available in the online supplemental material, summarizes the notations frequently used in the following discussions for ease of reference.

3.1 Architecture

The chunkservers in our proposal are organized as a DHT network; that is, each chunkserver implements a DHT protocol such as Chord [10] or Pastry [11]. A file in the system is partitioned into a number of fixed-size chunks, and "each" chunk has a unique *chunk handle* (or chunk identifier) named with a globally known hash function such as *SHA1* [24]. The hash function returns a unique identifier for a given file's pathname string and a chunk index. For example, the identifiers of the first and third chunks of file "`/user/tom/tmp/a.log`" are, respectively, `SHA1(/user/tom/tmp/a.log, 0)` and `SHA1(/user/tom/tmp/a.log, 2)`. Each chunkserver also has a unique ID. We represent the IDs of the chunkservers in V by $\frac{1}{n}, \frac{2}{n}, \frac{3}{n}, \ldots, \frac{n}{n}$; for short, denote the n chunkservers as $1, 2, 3, \ldots, n$. Unless otherwise clearly indicated, we denote the *successor* of chunkserver i as chunkserver $i+1$ and the successor of chunkserver n as chunkserver 1. In a typical DHT, a chunkserver i hosts the file chunks whose handles are within $(\frac{i-1}{n}, \frac{i}{n}]$, except for chunkserver n, which manages the chunks whose handles are in $(\frac{n}{n}, \frac{1}{n}]$.

To discover a file chunk, the DHT lookup operation is performed. In most DHTs, the average number of nodes visited for a lookup is $O(\log n)$ [10], [11] if each chunkserver i maintains $\log_2 n$ *neighbors*, that is, nodes $i + 2^k \bmod n$ for $k = 0, 1, 2, \ldots, \log_2 n - 1$. Among the $\log_2 n$ neighbors, the one $i + 2^0$ is the successor of i. To look up a file with l chunks, l lookups are issued.

DHTs are used in our proposal for the following reasons:

- The chunkservers self-configure and self-heal in our proposal because of their arrivals, departures, and failures, simplifying the system provisioning and management. Specifically, typical DHTs guarantee that if a node leaves, then its locally hosted chunks are reliably migrated to its successor; if a node joins, then it allocates the chunks whose IDs immediately precede the joining node from its successor to manage. Our proposal heavily depends on the node arrival and departure operations to migrate file chunks among nodes. Interested readers are referred to [10], [11] for the details of the self-management technique in DHTs.
- While lookups take a modest delay by visiting $O(\log n)$ nodes in a typical DHT, the lookup latency can be reduced because discovering the l chunks of a file can be performed in parallel. On the other hand, our proposal is independent of the DHT protocols. To further reduce the lookup latency, we can adopt state-of-the-art DHTs such as Amazon's Dynamo in [12] that offer one-hop lookup delay.
- The DHT network is transparent to the metadata management in our proposal. While the DHT network specifies the locations of chunks, our proposal can be integrated with existing large-scale distributed file systems, e.g., Google GFS [2] and Hadoop HDFS [3], in which a centralized master node manages the namespace of the file system and the mapping of file chunks to storage nodes. Specifically, to incorporate our proposal with the master node in GFS, each chunkserver periodically piggybacks its locally hosted chunks' information to the master in a heartbeat message [2] so that the master can gather the locations of chunks in the system.

- In DHTs, if nodes and file chunks are designated with uniform IDs, the maximum load of a node is guaranteed to be $O(\log n)$ times the average in a probability of $1 - O(\frac{1}{n})$ [14], [16], [25], thus balancing the loads of nodes to a certain extent. However, our proposal presented in Section 3.2 performs well for both uniform and nonuniform distributions of IDs of nodes and file chunks due to arbitrary file creation/deletion and node arrival/departure.

As discussed, the load rebalancing problem defined in Section 2 is $\mathcal{NP}$-hard (Theorem 1), which is technically challenging and thus demands an in-depth study. Orthogonal issues such as metadata management, file consistency models, and replication strategies are out of the scope of our study, and independent studies are required.

3.2 Load Rebalancing Algorithm

3.2.1 Overview

A large-scale distributed file system is in a *load-balanced state* if each chunkserver hosts no more than $\mathcal{A}$ chunks. In our proposed algorithm, each chunkserver node i first estimates whether it is underloaded (light) or overloaded (heavy) without global knowledge. A node is *light* if the number of chunks it hosts is smaller than the threshold of $(1 - \Delta_L)\mathcal{A}$ (where $0 \leq \Delta_L < 1$). In contrast, a *heavy* node manages the number of chunks greater than $(1 + \Delta_U)\mathcal{A}$, where $0 \leq \Delta_U < 1$. Δ_L and Δ_U are system parameters. In the following discussion, if a node i departs and rejoins as a successor of another node j, then we represent node i as node $j + 1$, node j's original successor as node $j + 2$, the successor of node j's original successor as node $j + 3$, and so on. For each node $i \in V$, if node i is light, then it seeks a heavy node and takes over at most $\mathcal{A}$ chunks from the heavy node.

We first present a load-balancing algorithm, in which each node has global knowledge regarding the system, that leads to low movement cost and fast convergence. We then extend this algorithm for the situation that the global knowledge is not available to each node without degrading its performance. Based on the global knowledge, if node i finds it is the least-loaded node in the system, i leaves the system by migrating its locally hosted chunks to its successor $i + 1$ and then rejoins instantly as the successor of the heaviest node (say, node j). To immediately relieve node j's load, node i requests $\min\{L_j - \mathcal{A}, \mathcal{A}\}$ chunks from j. That is, node i requests $\mathcal{A}$ chunks from the heaviest node j if j's load exceeds $2\mathcal{A}$; otherwise, i requests a load of $L_j - \mathcal{A}$ from j to relieve j's load.

Node j may still remain as the heaviest node in the system after it has migrated its load to node i. In this case, the current least-loaded node, say node i', departs and then rejoins the system as j's successor. That is, i' becomes node $j + 1$, and j's original successor i thus becomes node $j + 2$. Such a process repeats iteratively until j is no longer the heaviest. Then, the same process is executed to release the extra load on the next heaviest node in the system. This process repeats until all the heavy nodes in the system become light nodes. Such a load-balancing algorithm by mapping the least-loaded and most-loaded nodes in the system has properties as follows:

- **Low movement cost.** As node i is the lightest node among all chunkservers, the number of chunks migrated because of i's departure is small with the goal of reducing the movement cost.
- **Fast convergence rate.** The least-loaded node i in the system seeks to relieve the load of the heaviest node j, leading to quick system convergence towards the load-balanced state.

The mapping between the lightest and heaviest nodes at each time in a sequence can be further improved to reach the global load-balanced system state. The time complexity of the above algorithm can be reduced if each light node can know which heavy node it needs to request chunks beforehand, and then all light nodes can balance their loads in parallel. Thus, we extend the algorithm by pairing the top-k_1 underloaded nodes with the top-k_2 overloaded nodes. We use $\mathcal{U}$ to denote the set of top-k_1 underloaded nodes in the sorted list of underloaded nodes, and use $\mathcal{O}$ to denote the set of top-k_2 overloaded nodes in the sorted list of overloaded nodes. Based on the above-introduced load-balancing algorithm, the light node that should request chunks from the k_2'th ($k_2' \leq k_2$) most loaded node in $\mathcal{O}$ is the k_1'th ($k_1' \leq k_1$) least loaded node in $\mathcal{U}$, and

$$k_1' = \left\lceil \frac{\sum_{i=1}^{k_2'} i\text{th most loaded node} \in \mathcal{O} \, (L_i - \mathcal{A})}{\mathcal{A}} \right\rceil, \qquad (3)$$

where $\sum_{i=1}^{k_2'} i\text{th most loaded node} \in \mathcal{O}(L_i - \mathcal{A})$ denotes the sum of the excess loads in the top-k_2' heavy nodes. It means that the top-k_1' light nodes should leave and rejoin as successors of the top-k_2' overloaded nodes. Thus, according to (3), based on its position k_1' in $\mathcal{U}$, each light node can compute k_2' to identify the heavy node to request chunks. Light nodes concurrently request chunks from heavy nodes, and this significantly reduces the latency of the sequential algorithm in achieving the global system load-balanced state.

We have introduced our algorithm when each node has global knowledge of the loads of all nodes in the system. However, it is a formidable challenge for each node to have such global knowledge in a large-scale and dynamic computing environment. We then introduce our basic algorithms that perform the above idea in a distributed manner without global knowledge in Sections 3.2.2. Section 3.2.3 improves our proposal by taking advantage of physical network locality to reduce network traffic caused by the migration of file chunks. Recall that we first assume that the nodes have identical capacities in order to simplify the discussion. We then discuss the exploitation of node capacity heterogeneity in Section 3.2.4. Finally, high file availability is usually demanded from large-scale and dynamic distributed storage systems that are prone to failures. To deal with this issue, Section 3.2.5 discusses the maintenance of replicas for each file chunk.

3.2.2 Basic Algorithms

Algorithms 1 and 2 (see Appendix C, which is available in the online supplemental material) detail our proposal;

Algorithm 1 specifies the operation that a light node i seeks an overloaded node j, and Algorithm 2 shows that i requests some file chunks from j. Without global knowledge, pairing the top-k_1 light nodes with the top-k_2 heavy nodes is clearly challenging. We tackle this challenge by enabling a node to execute the load-balancing algorithm introduced in Section 3.2.1 based on a sample of nodes. In the basic algorithm, each node implements the *gossip-based aggregation protocol* in [26] and [27] to collect the load statuses of a sample of randomly selected nodes. Specifically, each node contacts a number of randomly selected nodes in the system and builds a vector denoted by $\mathcal{V}$. A vector consists of entries, and each entry contains the ID, network address and load status of a randomly selected node. Using the gossip-based protocol, each node i exchanges its locally maintained vector with its neighbors until its vector has s entries. It then calculates the average load of the s nodes denoted by $\widetilde{\mathcal{A}}_i$ and regards it as an estimation of $\mathcal{A}$ (Line 1 in Algorithm 1).

If node i finds itself is a light node (Line 2 in Algorithm 1), it seeks a heavy node to request chunks. Node i sorts the nodes in its vector including itself based on the load status and finds its position k_1' in the sorted list, i.e., it is the top-k_1 underloaded node in the list (Lines 3-5 in Algorithm 1). Node i finds the top-k_2' overloaded nodes in the list such that the sum of these nodes' excess loads is the least greater than or equal to $k_1'\widetilde{\mathcal{A}}_i$ (Line 6 in Algorithm 1). Formula (ii) in the algorithm is derived from (3). The complexity of the step in Line 6 is $O(|\mathcal{V}|)$. Then, the k_2'th overloaded node is the heavy node that node i needs to request chunks (Line 7 in Algorithm 1). Considering the step in Line 4, the overall complexity of Algorithm 1 is then $O(|\mathcal{V}|\log|\mathcal{V}|)$.

We note the following:

- Our proposal is distributed in the sense that each node in the system performs Algorithms 1 and 2 simultaneously without synchronization. It is possible that a number of distinct nodes intend to share the load of node j (Line 1 of Algorithm 2). Thus, j offloads parts of its load to a randomly selected node among the requesters. Similarly, a number of heavy nodes may select an identical light node to share their loads. If so, the light node randomly picks one of the heavy nodes in the reallocation.
- The nodes perform our load rebalancing algorithm periodically, and they balance their loads and minimize the movement cost in a best effort fashion.

Example: Fig. 2 depicts a working example of our proposed algorithm. There are $n = 10$ chunkservers in the system; the initial loads of the nodes are shown in Fig. 2a. Assume $\Delta_L = \Delta_U = 0$ in the example. Then, nodes $N1$, $N2$, $N3$, $N4$, and $N5$ are light, and nodes $N6$, $N7$, $N8$, $N9$, and $N10$ are heavy. Each node performs the load-balancing algorithm independently, and we choose $N1$ as an example to explain the load-balancing algorithm. $N1$ first queries the loads of $N3$, $N6$, $N7$, and $N9$ selected randomly from the system (Fig. 2b). Based on the samples, $N1$ estimates the ideal load $\mathcal{A}$ (i.e., $\widetilde{\mathcal{A}}_{N1} = \frac{L_{N1}+L_{N3}+L_{N6}+L_{N7}+L_{N9}}{5}$). It notices that it is a light node. It then finds the heavy node it needs to request chunks. The heavy node is the most loaded node (i.e., $N9$) as $N1$ is the lightest among N_1 and its sampled nodes $\{N3, N6, N7, N9\}$ (Line 6 in Algorithm 1). $N1$ then sheds its load to its successor $N2$, departs from the system, and rejoins the system as the successor of $N9$. $N1$ allocates $\min\{L_{N9} - \widetilde{\mathcal{A}}_{N1}, \widetilde{\mathcal{A}}_{N1}\} = \widetilde{\mathcal{A}}_{N1}$ chunks from $N9$ (Lines 5 and 6 in Algorithm 2).

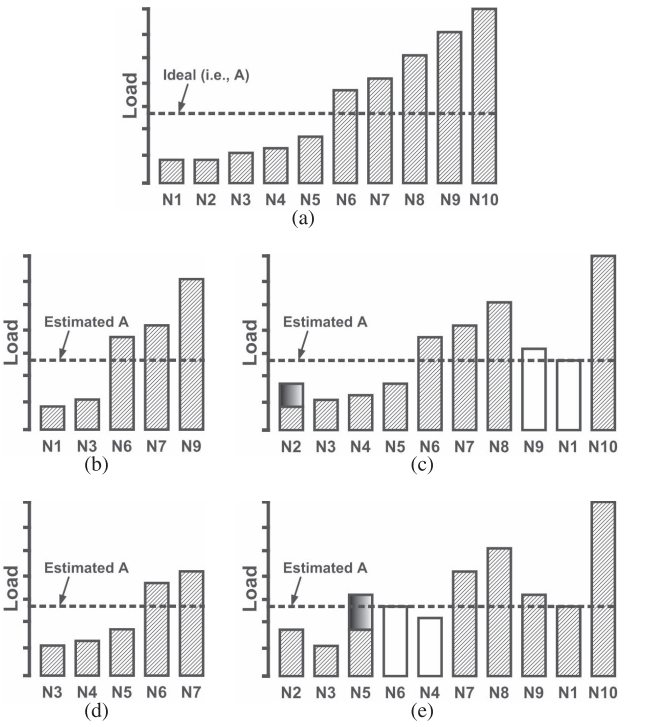


Fig. 2. An example illustrating our algorithm, where (a) the initial loads of chunkservers $N1, N2, \ldots, N10$, (b) $N1$ samples the loads of $N1$, $N3$, $N6$, $N7$, and $N9$ in order to perform the load rebalancing algorithm, (c) $N1$ leaves and sheds its loads to its successor $N2$, and then rejoins as $N9$'s successor by allocating $\widetilde{\mathcal{A}}_{N1}$ chunks (the ideal number of chunks $N1$ estimates to manage) from $N9$, (d) $N4$ collects its sample set $\{N3, N4, N5, N6, N7\}$, and (e) $N4$ departs and shifts its load to $N5$, and it then rejoins as the successor of $N6$ by allocating $L_6 - \widetilde{\mathcal{A}}_{N4}$ chunks from $N6$.

In the example, $N4$ also performs the load rebalancing algorithm by first sampling $\{N3, N4, N5, N6, N7\}$ (Fig. 2d). Similarly, $N4$ determines to rejoin as the successor of $N6$. $N4$ then migrates its load to $N5$ and rejoins as the successor of $N6$ (Fig. 2e). $N4$ requests $\min\{L_{N6} - \widetilde{\mathcal{A}}_{N4}, \widetilde{\mathcal{A}}_{N4}\} = L_6 - \widetilde{\mathcal{A}}_{N4}$ chunks from $N6$.

Our load-balancing algorithm offers an analytical performance guarantee and exhibits a fast convergence rate in terms of algorithmic rounds. Let the initial number of heavy nodes be k (where $k \leq |V| = n$). Then, we have the major analytical result as follows:

Theorem 2. *Algorithms 1 and 2 take $O(\log\log k)$ algorithmic rounds in expectation such that the system contains no light nodes.*

For the detailed proof of Theorem 2, interested readers are referred to Appendix D, which is available in the online supplemental material.

3.2.3 Exploiting Physical Network Locality

A DHT network is an overlay on the application level. The logical proximity abstraction derived from the DHT does not necessarily match the physical proximity information in reality. That means a message traveling between two

neighbors in a DHT overlay may travel a long physical distance through several physical network links. In the load-balancing algorithm, a light node i may rejoin as a successor of a remote heavy node j. Then, the requested chunks migrated from j to i need to traverse several physical network links, thus generating considerable network traffic and consuming significant network resources (i.e., the buffers in the switches on a communication path for transmitting a file chunk from a source node to a destination node).

We improve our proposal by exploiting physical network locality. Basically, instead of collecting a single vector (i.e., $\mathcal{V}$ in Algorithm 1) per algorithmic round, each light node i gathers $n_{\mathcal{V}}$ vectors. Each vector is built using the method introduced previously. From the $n_{\mathcal{V}}$ vectors, the light node i seeks $n_{\mathcal{V}}$ heavy nodes by invoking Algorithm 1 (i.e., SEEK) for each vector and then selects the physically closest heavy node based on the message round-trip delay.

In Algorithm 3 (see Appendix C, which is available in the online supplemental material), Lines 2 and 3 take $O(n_{\mathcal{V}}|\mathcal{V}|\log|\mathcal{V}|)$. We will offer a rigorous performance analysis for the effect of varying $n_{\mathcal{V}}$ in Appendix E, which is available in the online supplemental material. Specifically, we discuss the tradeoff between the value of $n_{\mathcal{V}}$ and the movement cost. A larger $n_{\mathcal{V}}$ introduces more overhead for message exchanges, but results in a smaller movement cost.

To demonstrate Algorithm 3, consider the example shown in Fig. 2. Let $n_{\mathcal{V}} = 2$. In addition to the sample set $\mathcal{V}_1 = \{N1, N3, N6, N7, N9\}$ (Fig. 2b), $N1$ gathers another sample set, say, $\mathcal{V}_2 = \{N1, N4, N5, N6, N8\}$. $N1$ identifies the heavy node $N9$ in $\mathcal{V}_1$ and $N8$ in $\mathcal{V}_2$. Suppose $N9$ is physically closer to $N1$ than $N8$. Thus, $N1$ rejoins as a successor of $N9$ and then receives chunks from $N9$. Node i also offloads its original load to its successor. For example, in Figs. 2a and 2b, node $N1$ migrates its original load to its successor node $N2$ before $N1$ rejoins as node $N9$'s successor. To minimize the network traffic overhead in shifting the load of the light node i to node $i+1$, we suggest *initializing* the DHT network such that every two nodes with adjacent IDs (i.e., nodes i and $i+1$) are geometrically close. As such, given the potential IP addresses of the participating nodes (a 4D lattice) in a storage network, we depend on the *space-filling curve* technique (e.g., *Hilbert curve* in [28]) to assign IDs to the nodes, making physically close nodes have adjacent IDs. More specifically, given a 4D lattice representing all IP addresses of storage nodes, the space-filling curve attempts to visit each IP address and assign a unique ID to each address such that geometrically close IP addresses are assigned with numerically close IDs. By invoking the space filling curve function with the input of an IP address, a unique numerical ID is returned.

Algorithm 3 has the performance guarantee for the data center networks with α-*power law latency expansion*. By latency we mean the number of physical links traversed by a message between two storage nodes in a network. For example, *BCube* [29] and *CamCube* [30], respectively, organize the data center network as a hypercube interconnect and a 2D torus network, thus exhibiting 3- and 2-power law latency expansion. For further discussion, interested readers are referred to Appendix E, which is available in the online supplemental material.

3.2.4 Taking Advantage of Node Heterogeneity

Nodes participating in the file system are possibly heterogeneous in terms of the numbers of file chunks that the nodes can accommodate. We assume that there is one bottleneck resource for optimization although a node's capacity in practice should be a function of computational power, network bandwidth, and storage space [20], [31]. Given the capacities of nodes (denoted by $\{\beta_1, \beta_2, \ldots, \beta_n\}$), we enhance the basic algorithm in Section 3.2.2 as follows: each node i approximates the ideal number of file chunks that it needs to host in a load balanced state as follows:

$$\widetilde{\mathcal{A}}_i = \gamma\beta_i, \tag{4}$$

where γ is the *load per unit capacity* a node should manage in the load balanced state and

$$\gamma = \frac{m}{\sum_{k=1}^{n}\beta_k}, \tag{5}$$

where m is the number of file chunks stored in the system.

As mentioned previously, in the distributed file system for MapReduce-based applications, the load of a node is typically proportional to the number of file chunks the node possesses [3]. Thus, the rationale of this design is to ensure that the number of file chunks managed by node i is proportional to its capacity. To estimate the aggregate γ, our proposal again relies on the gossip-based aggregation protocol in [26] and [27] in computing the value.

Algorithm 4 in Appendix C, which is available in the online supplemental material, presents the enhancement for Algorithm 1 to exploit node heterogeneity, which is similar to Algorithm 1 and is self-explanatory. If a node i estimates that it is light (i.e., $L_i < (1-\Delta_L)\widetilde{\mathcal{A}}_i$), i then rejoins as a successor of a heavy node j. i seeks j based on its sampled node set $\mathcal{V}$. i sorts the set in accordance with $\frac{L_t}{\beta_t}$, the load per capacity unit a node currently receives, for all $t \in \mathcal{V}$. When node i notices that it is the kth least-loaded node (Line 6 in Algorithm 4), it then identifies node j and rejoins as a successor of node j. Node j is the least-loaded node in the set of nodes $\mathcal{P} \subset \mathcal{V}$ having the minimum cardinality, where 1) the nodes in $\mathcal{P}$ are heavy, and 2) the total excess load of nodes in $\mathcal{P}$ is not less than $\sum_{j\text{th light node in } \mathcal{V}}^{k}\widetilde{\mathcal{A}}_j$ (Line 7 in Algorithm 4). Here, $\sum_{j\text{th light node in } \mathcal{V}}^{k}\widetilde{\mathcal{A}}_j$ indicates the sum of loads that the top-k light nodes in $\mathcal{V}$ will manage in a load-balanced system state.

3.2.5 Managing Replicas

In distributed file systems (e.g., Google GFS [2] and Hadoop HDFS [3]), a constant number of replicas for each file chunk are maintained in distinct nodes to improve file availability with respect to node failures and departures. Our current load-balancing algorithm does not treat replicas distinctly. It is unlikely that two or more replicas are placed in an identical node because of the random nature of our load rebalancing algorithm. More specifically, each underloaded node samples a number of nodes, each selected with a probability of $\frac{1}{n}$, to share their loads (where n is the total number of storage nodes). Given k replicas for each file chunk (where k is typically a small constant, and $k=3$ in GFS), the probability that k' replicas ($k' \le k$) are placed in an

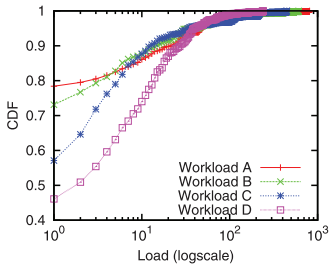


Fig. 3. The workload distribution.

identical node due to migration of our load-balancing algorithm is $(\frac{1}{n})^{k'}$ independent of their initial locations. For example, in a file system with $n = 1{,}000$ storage nodes and $k = 3$, then the probabilities are only $\frac{1}{10^6}$ and $\frac{1}{10^9}$ for two and three replicas stored in the same node, respectively. Consequently, the probability of more than one replica appearing in a node due to our proposal is approximately (as $k \ll n$)

$$\sum_{i=2}^{k} \left(\frac{1}{n}\right)^{i}. \tag{6}$$

Replica management in distributed systems has been extensively discussed in the literature. Given any file chunk, our proposal implements the *directory-based* scheme in [32] to trace the locations of k replicas for the file chunk. Precisely, the file chunk is associated with $k-1$ pointers that keep track of $k-1$ randomly selected nodes storing the replicas.

We have investigated the percentage of nodes storing redundant replicas due to our proposal. In our experiments, the number of file chunks and the number of nodes in the system are $m = 10{,}000$ and $n = 1{,}000$, respectively. (Details of the experimental settings are discussed in Section 4.) Among the $m = 10{,}000$ file chunks, we investigate in the experiment $k = 2, 4, 8$ replicas for each file chunk; that is, there are 5,000, 2,500 and 1,250 unique chunks in the system, respectively. The experimental results indicate that the number of nodes managing more than one redundant chunk due to our proposal is very small. Specifically, each node maintains no redundant replicas for $k = 2, 4$, and only 2 percent of nodes store ≥ 2 redundant replicas for $k = 8$.

4 Simulations

4.1 Simulation Setup and Workloads

The performance of our algorithm is evaluated through computer simulations. Our simulator is implemented with Pthreads. In the simulations, we carry out our proposal based on the Chord DHT protocol [10] and the gossip-based aggregation protocol in [26] and [27]. In the default setting, the number of nodes in the system is $n = 1{,}000$, and the number of file chunks is $m = 10{,}000$. To the best of our knowledge, there are no representative realistic workloads available. Thus, the number of file chunks initially hosted by a node in our simulations follows the geometric distribution, enabling stress tests as suggested in [15] for various load rebalancing algorithms. Fig. 3 shows the cumulative distribution functions (CDF) of the file chunks in the simulations, where workloads A, B, C, and D represent four distinct geometric distributions. Specifically, these distributions indicate that a small number of nodes initially possess a large number of chunks. The four workloads exhibit different variations of the geometric distribution.

We have compared our algorithm with the competing algorithms called *centralized matching* in [3] and *distributed matching* in [14], respectively. In Hadoop HDFS [3], a standalone load-balancing server (i.e., balancer) is employed to rebalance the loads of storage nodes. The server acquires global information on the file chunks distributed in the system from the namenode that manages the metadata of the entire file system. Based on this global knowledge, it partitions the node set into two subsets, where one (denoted by $\mathcal{O}$) contains overloaded nodes, and the other (denoted by $\mathcal{U}$) includes the underloaded nodes. Conceptually, the balancer randomly selects one heavy node $i \in \mathcal{O}$ and one light node $j \in \mathcal{U}$ to reallocate their loads. The reallocation terminates if the balancer cannot find a pair of heavy and light nodes to reallocate their loads. Notably, to exploit physical network locality and thus reduce network traffic, the balancer first pairs i and j if i and j appear in the same rack. If a node in a rack remains unbalanced, and if it cannot find any other node in the same rack to pair, then the node will be matched with another node, in a foreign rack. The balancer in HDFS does not differentiate different locations of foreign racks when performing the matches. In our simulations, each rack has 32 nodes in default.

On the contrary, a storage node i in the decentralized random matching algorithm in [14] independently and randomly selects another node j to share its load if the ratio of i's load to j's is smaller (or larger) than a predefined threshold δ (or $\frac{1}{\delta}$). As suggested by [14], δ is greater than 0 and not more than $\frac{1}{4}$. In our simulations, $\delta = \frac{1}{4}$. To be comparable, we also implement this algorithm on the Chord DHT. Thus, when node i attempts to share the load of node j, node i needs to leave and rejoin as node j's successor.

In our algorithm, we set $\Delta_L = \Delta_U = 0.2$ in default. Each node maintains n_V vectors, each consisting of $s = 100$ random samples of nodes (entries in a vector may be duplicated), for estimating $\mathcal{A}$. $n_V = 1$ in default.

The cloud network topology interconnecting the storage nodes simulated is a 2D torus direct network, as suggested by the recent studies in [30] and [33]. (In Appendix E, which is available in the online supplemental material, we also investigate the performance effects on the hypercube topology in [29].) Finally, unless otherwise noted, each node has an identical capacity in the simulations.

Due to space limitation, we report the major performance results in Section 4.2. Extensive performance results can be found in the appendix, which is available in the online supplemental material, including the effect of varying the number of file chunks (Appendix G, which is available in the online supplemental material), the effect of different numbers of samples (Appendix H, which is available in the online supplemental material), the effect of different algorithmic rounds (Appendix I, which is available in the online supplemental material) and the effect of system dynamics (Appendix J, which is available in the online supplemental material).

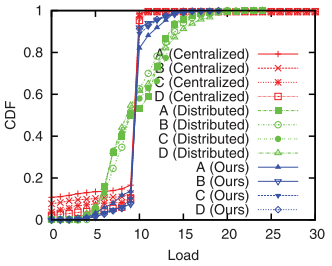


Fig. 4. The load distribution.

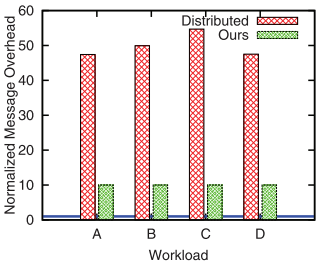


Fig. 6. The message overhead.

4.2 Simulation Results

Fig. 4 presents the simulation results of the load distribution after performing the investigated load-balancing algorithms. Here, the nodes simulated have identical capacity. The simulation results show that `centralized matching` performs very well as the load balancer gathers the global information from the namenode managing the entire file system. Since $\mathcal{A} = 10$ is the ideal number of file chunks a node should manage in a load-balanced state, in `centralized matching`, most nodes have 10 chunks. In contrast, `distributed matching` performs worse than `centralized matching` and our proposal. This is because each node randomly probes other nodes without global knowledge about the system. Although our proposal is distributed and need not require each node to obtain global system knowledge, it is comparable with `centralized matching` and remarkably outperforms `distributed matching` in terms of load imbalance factor.

Fig. 5 shows the movement costs of `centralized matching`, `distributed matching`, and our algorithm, where the movement costs have been normalized to that of `centralized matching` (indicated by the horizontal line in the figure). Clearly, the movement cost of our proposal is only 0.37 times the cost of `distributed matching`. Our algorithm matches the top least-loaded light nodes with the top most-loaded heavy nodes, leading to a fewer number of file chunks migrated. In contrast, in `distributed matching`, a heavy node i may be requested to relieve another node j with a relatively heavier load, resulting in the migration of a large number of chunks originally hosted by i to i's successor. We also observe that our proposal may incur slightly more movement cost than that of `centralized matching`. This is because in our proposal, a light node needs to shed its load to its successor.

Fig. 6 shows the total number of messages generated by a load rebalancing algorithm, where the message overheads in `distributed matching` and our proposal are normalized to that of `centralized matching`. The simulation results indicate that `centralized matching` introduces much less message overhead than `distributed matching` and our proposal, as each node in `centralized matching` simply informs the centralized load balancer of its load and capacity. On the contrary, in `distributed matching` and our proposal, each node probes a number of existing nodes in the system, and may then reallocate its load from/to the probed nodes, introducing more messages. We also see that our proposal clearly produces less message overhead than `distributed computing`. Specifically, any node i in our proposal gathers partial system knowledge from its neighbors [26], [27], whereas node i in `distributed matching` takes $O(\log n)$ messages to probe a randomly selected node in the network.

Both `distributed matching` [14] and our proposal depend on the Chord DHT network in the simulations. However, nodes may leave and rejoin the DHT network for load rebalancing, thus increasing the overhead required to maintain the DHT structure. Thus, we further investigate the number of rejoining operations. Note that `centralized matching` introduces no rejoining overhead because nodes in `centralized matching` does not need to self-organize and self-heal for rejoining operations. Fig. 7 illustrates the simulation results, where the number of rejoining operations caused by our algorithm is normalized to that of `distributed matching` (indicated by the horizontal line). We see that the number of rejoining operations in `distributed matching` can be up to two times greater than that of our algorithm. This is because a heavy node in `distributed matching` may leave and rejoin the network to reduce the load of another heavy node. On the contrary, in our proposal, only light nodes rejoin the system as successors of heavy nodes. Our algorithm attempts to pair light and heavy nodes precisely, thus reducing the number of rejoining operations.

In Section 3.2.3, we improve our basic load rebalancing algorithm by exploiting physical network locality. The network traffic introduced by `centralized matching`,

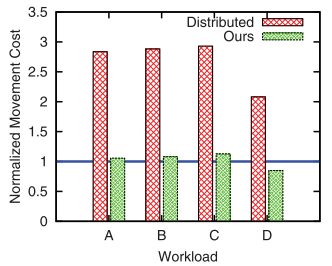


Fig. 5. The movement cost.

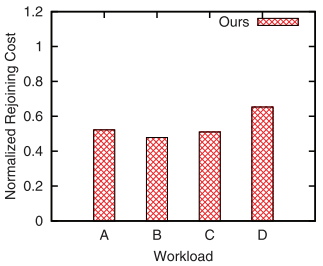


Fig. 7. The rejoining cost.

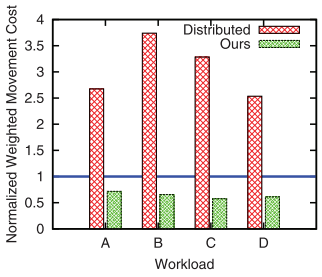


Fig. 8. The WMC.

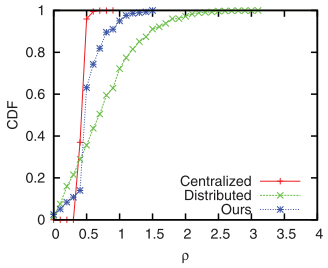


Fig. 10. The effect of heterogeneity.

distributed matching, and our proposal is thus investigated. Specifically, we define the *weighted movement cost* (WMC) as follows:

$$\sum_{\text{chunk } i \in \mathcal{M}} \text{size}_i \times \text{link}_i, \tag{7}$$

where $\mathcal{M}$ denotes the set of file chunks selected for reallocation by a load rebalancing algorithm, size_i is the size of file chunk i, and link_i represents the number of physical links chunk i traverses. In the simulations, the size of each file chunk is identical. We assume that $\text{size}_i = 1$ for all $i \in \mathcal{M}$ without loss of generality. Hence, based on (7), the greater the WMC, the more physical network links used for load reallocation.

Fig. 8 shows the WMC's caused by centralized matching, distributed matching, and our proposal (where the costs of distributed matching and our proposal are normalized to that of centralized matching, and Fig. 9 presents the corresponding CDF for the numbers of physical links traversed by file chunks for workload C. (The other workloads are not presented due to space limitation.) Notably, as suggested by Raicu et al. [30], [33], a 2D torus direct network is simulated in the experiment. The simulation results indicate that our proposal clearly outperforms centralized matching and distributed matching in terms of the WMC. In contrast to distributed matching, which does not exploit the physical network locality for pairing light and heavy nodes, centralized matching initially pairs nodes present in the same rack and then matches light nodes with heavy ones in different racks. (Here, each rack contains 32 nodes.) However, centralized matching does not differentiate the locations of nodes in different racks for matching. Unlike centralized matching and distributed matching, each light node in our proposal first finds eight matched heavy nodes from its eight vectors (i.e., $n_{\mathcal{V}} = 8$ in this experiment), and then chooses the physically closest node to pair with, leading a shorter physical distance for migrating a chunk. This operation effectively differentiates nodes in different network locations, and considerably reduces the WMC.

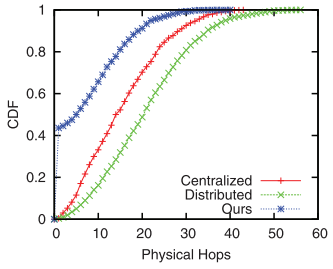


Fig. 9. The breakdown of WMC.

As previously mentioned in Section 3.2.3, our proposal organizes nodes in the Chord ring such that adjacent nodes in the ring are physically close. Before rejoining a node, the node departs and migrates its locally hosted file chunks to its physically close successor. The simulation results illustrate that $\approx 45\%$ of file chunks in our proposal are moved to the physically closest nodes, which is due to our design having a locality-aware Chord ring (see Fig. 9). Interested readers may refer to Appendix E, which is available in the online supplemental material, for the analytical model that details the performance of the locality-oblivious and locality-aware approaches discussed in this section. Moreover, in Appendix E, which is available in the online supplemental material, the effects of the different numbers of racks in centralized matching and the different numbers of node vectors $n_{\mathcal{V}}$ maintained by a node in our proposal are investigated.

We then investigate the effect of node heterogeneity for centralized matching, distributed matching, and our proposal. In this experiment, the capacities of nodes follow the power-law distribution, namely, the Zipf distribution [19], [20], [22]. Here, the ideal number of file chunks per unit capacity a node should host is approximately equal to $\gamma = 0.5$. The maximum and minimum capacities are 110 and 2, respectively, and the mean is ≈ 11. Fig. 10 shows the simulation results for workload C. In Fig. 10, the ratio of the number of file chunks hosted by each node $i \in V$ to i's capacity, denoted by ρ, is measured. Node i attempts to minimize $\|\rho - \gamma\|$ in order to approach its load-balanced state. The simulation results indicate that centralized matching performs better than distributed matching and our proposal. This is because capable nodes in distributed matching and our proposal may need to offload their loads to their successors that are incapable of managing large numbers of file chunks. We also see that our proposal manages to perform reasonably well, clearly outperforming distributed matching. In our proposal, although a light node may shed its load to its successor j, which is incapable and accordingly overloaded, another light node can quickly discover the heavy node j to share j's load. In particular, our proposal seeks the top-k light nodes in the reallocation and thus reduces the movement cost caused by rejoining these light nodes as compared to distributed matching.

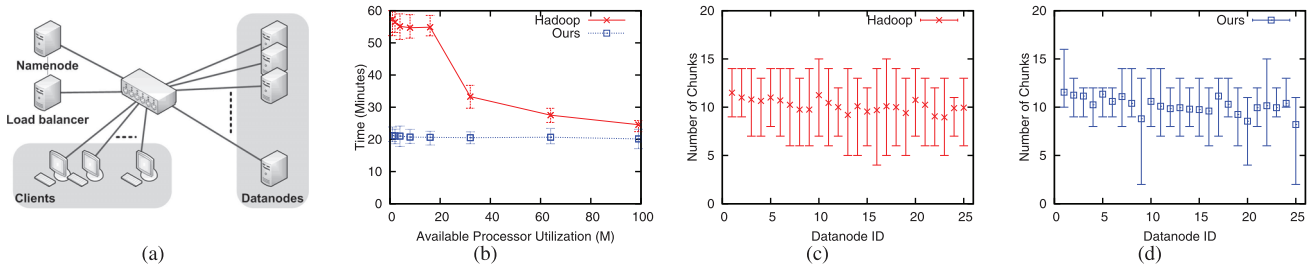


Fig. 11. The experimental environment and performance results, where (a) shows the setup of the experimental environment, (b) indicates the time elapsed of performing the HDFS load balancer and our proposal, and (c) and (d) show the distributions of file chunks for the HDFS load balancer and our proposal, respectively

5 Implementation and Measurement

5.1 Experimental Environment Setup

We have implemented our proposal in Hadoop HDFS 0.21.0, and assessed our implementation against the load balancer in HDFS. Our implementation is demonstrated through a small-scale cluster environment (Fig. 11a) consisting of a single, dedicated namenode and 25 datanodes, each with Ubuntu 10.10 [34]. Specifically, the namenode is equipped with Intel Core 2 Duo E7400 processor and 3 Gbytes RAM. As the number of file chunks in our experimental environment is small, the RAM size of the namenode is sufficient to cache the entire namenode process and the metadata information, including the directories and the locations of file chunks.

In the experimental environment, a number of clients are established to issue requests to the namenode. The requests include commands to create directories with randomly designated names, to remove directories arbitrarily chosen, etc. Due to the scarce resources in our environment, we have deployed 4 clients to generate requests to the namenode. However, this cannot overload the namenode to mimic the situation as reported in [8]. To emulate the load of the namenode in a production system and investigate the effect of the namenode's load on the performance of a load-balancing algorithm, we additionally limit the processor cycles available to the namenode by varying the maximum processor utilization, denoted by $\mathcal{M}$, available to the namenode up to $\mathcal{M} = 1\%, 2\%, 8\%, 16\%, 32\%, 64\%, 99\%$. The lower processor availability to the namenode represents the less CPU cycles that the namenode can allocate to handle the clients' requests and to talk to the load balancer.

As data center networks proposed recently (e.g., [29]) can offer a fully bisection bandwidth, the total number of chunks scattered in the file system in our experiments is limited to 256 such that the network bandwidth in our environment (i.e., all nodes are connected with a 100 Mbps fast Ethernet switch) is not the performance bottleneck. Particularly, the size of a file chunk in the experiments is set to 16 Mbytes. Compared to each experimental run requiring 20-60 minutes, transferring these chunks takes no more than $\frac{16\times256\times8}{100} \approx 328$ seconds ≈ 5.5 minutes in case the network bandwidth is fully utilized. The initial placement of the 256 file chunks follows the geometric distribution as discussed in Section 4.

For each experimental run, we quantity the time elapsed to complete the load-balancing algorithms, including the HDFS load balancer and our proposal. We perform 20 runs for a given $\mathcal{M}$ and average the time required for executing a load-balancing algorithm. Additionally, the 5- and 95-percentiles are reported. For our proposal, we let $\Delta_U = \Delta_L = 0.2$. Each datanode performs 10 random samples.

Note that 1) in the experimental results discussed later, we favor HDFS by dedicating a standalone node to perform the HDFS load-balancing function. By contrast, our proposal excludes the extra, standalone node. 2) The datanodes in our cluster environment are homogeneous, each with Intel Celeron 430 and 3 Gbytes RAM. We, thus, do not study the effect of the node heterogeneity on our proposal. 3) We also do not investigate the effect of network locality on our proposal as the nodes in our environment are only linked with a single switch.

5.2 Experimental Results

We demonstrate in Fig. 11 the experimental results. Fig. 11b shows the time required for performing the HDFS load balancer and our proposal. Our proposal clearly outperforms the HDFS load balancer. When the namenode is heavily loaded (i.e., small $\mathcal{M}$'s), our proposal remarkably performs better than the HDFS load balancer. For example, if $\mathcal{M} = 1\%$, the HDFS load balancer takes approximately 60 minutes to balance the loads of datanodes. By contrast, our proposal takes nearly 20 minutes in the case of $\mathcal{M} = 1\%$. Specifically, unlike the HDFS load balancer, our proposal is independent of the load of the namenode.

In Figs. 11c and 11d, we further show the distributions of chunks after performing the HDFS load balancer and our proposal. As there are 256 file chunks and 25 datanodes, the ideal number of chunks that each datanode needs to host is $\frac{256}{25} \approx 10$. Due to space limitation, we only offer the experimental results for $\mathcal{M} = 1$ and the results for $\mathcal{M} \neq 1$ conclude the similar. Figs. 11c and 11d indicate that our proposal is comparable to the HDFS load balancer, and balances the loads of datanodes, effectively.

6 Summary

A novel load-balancing algorithm to deal with the load rebalancing problem in large-scale, dynamic, and distributed file systems in clouds has been presented in this paper. Our proposal strives to balance the loads of nodes and reduce the demanded movement cost as much as possible, while taking advantage of physical network locality and node heterogeneity. In the absence of representative real workloads (i.e., the distributions of file chunks in a large-scale storage system) in the public domain, we have

investigated the performance of our proposal and compared it against competing algorithms through synthesized probabilistic distributions of file chunks. The synthesis workloads stress test the load-balancing algorithms by creating a few storage nodes that are heavily loaded. The computer simulation results are encouraging, indicating that our proposed algorithm performs very well. Our proposal is comparable to the centralized algorithm in the Hadoop HDFS production system and dramatically outperforms the competing distributed algorithm in [14] in terms of load imbalance factor, movement cost, and algorithmic overhead. Particularly, our load-balancing algorithm exhibits a fast convergence rate. The efficiency and effectiveness of our design are further validated by analytical models and a real implementation with a small-scale cluster environment.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers who have provided us with valuable comments to improve their study. Hung-Chang Hsiao and Chung-Hsueh Yi were partially supported by Taiwan National Science Council under Grants 100-2221-E-006-193 and 101-2221-E-006-097, and by the Ministry of Education, Taiwan, under the NCKU Project of Promoting Academic Excellence & Developing World Class Research Centers. Haiying Shen was supported in part by US National Science Foundation (NSF) grants CNS-1254006, CNS-1249603, OCI-1064230, CNS-1049947, CNS-1156875, CNS-0917056 and CNS-1057530, CNS-1025652, CNS-0938189, CSR-2008826, CSR-2008827, Microsoft Research Faculty Fellowship 8300751, and the US Department of Energy's Oak Ridge National Laboratory including the Extreme Scale Systems Center located at ORNL and DoD 4000111689.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. Sixth Symp. Operating System Design and Implementation (OSDI '04),* pp. 137-150, Dec. 2004.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03),* pp. 29-43, Oct. 2003.

[3] Hadoop Distributed File System, http://hadoop.apache.org/hdfs/, 2012.

[4] VMware, http://www.vmware.com/, 2012.

[5] Xen, http://www.xen.org/, 2012.

[6] Apache Hadoop, http://hadoop.apache.org/, 2012.

[7] Hadoop Distributed File System "Rebalancing Blocks," http://developer.yahoo.com/hadoop/tutorial/module2.html#rebalancing, 2012.

[8] K. McKusick and S. Quinlan, "GFS: Evolution on Fast-Forward," *Comm. ACM,* vol. 53, no. 3, pp. 42-49, Jan. 2010.

[9] HDFS Federation, http://hadoop.apache.org/common/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/Federation.html, 2012.

[10] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications," *IEEE/ACM Trans. Networking,* vol. 11, no. 1, pp. 17-21, Feb. 2003.

[11] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms Heidelberg,* pp. 161-172, Nov. 2001.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," *Proc. 21st ACM Symp. Operating Systems Principles (SOSP '07),* pp. 205-220, Oct. 2007.

[13] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems," *Proc. Second Int'l Workshop Peer-to-Peer Systems (IPTPS '02),* pp. 68-79, Feb. 2003.

[14] D. Karger and M. Ruhl, "Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems," *Proc. 16th ACM Symp. Parallel Algorithms and Architectures (SPAA '04),* pp. 36-43, June 2004.

[15] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems," *Proc. 13th Int'l Conf. Very Large Data Bases (VLDB '04),* pp. 444-455, Sept. 2004.

[16] J.W. Byers, J. Considine, and M. Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables," *Proc. First Int'l Workshop Peer-to-Peer Systems (IPTPS '03),* pp. 80-87, Feb. 2003.

[17] G.S. Manku, "Balanced Binary Trees for ID Management and Load Balance in Distributed Hash Tables," *Proc. 23rd ACM Symp. Principles Distributed Computing (PODC '04),* pp. 197-205, July 2004.

[18] A. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries," *Proc. ACM SIGCOMM '04,* pp. 353-366, Aug. 2004.

[19] Y. Zhu and Y. Hu, "Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 16, no. 4, pp. 349-361, Apr. 2005.

[20] H. Shen and C.-Z. Xu, "Locality-Aware and Churn-Resilient Load Balancing Algorithms in Structured P2P Networks," *IEEE Trans. Parallel and Distributed Systems,* vol. 18, no. 6, pp. 849-862, June 2007.

[21] Q.H. Vu, B.C. Ooi, M. Rinard, and K.-L. Tan, "Histogram-Based Global Load Balancing in Structured Peer-to-Peer Systems," *IEEE Trans. Knowledge Data Eng.,* vol. 21, no. 4, pp. 595-608, Apr. 2009.

[22] H.-C. Hsiao, H. Liao, S.-S. Chen, and K.-C. Huang, "Load Balance with Imperfect Information in Structured Peer-to-Peer Systems," *IEEE Trans. Parallel Distributed Systems,* vol. 22, no. 4, pp. 634-649, Apr. 2011.

[23] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[24] D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," RFC 3174, Sept. 2001.

[25] M. Raab and A. Steger, "Balls into Bins-A Simple and Tight Analysis," *Proc. Second Int'l Workshop Randomization and Approximation Techniques in Computer Science,* pp. 159-170, Oct. 1998.

[26] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-Based Aggregation in Large Dynamic Networks," *ACM Trans. Computer Systems,* vol. 23, no. 3, pp. 219-252, Aug. 2005.

[27] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M.V. Steen, "Gossip-Based Peer Sampling," *ACM Trans. Computer Systems,* vol. 25, no. 3, Aug. 2007.

[28] H. Sagan, *Space-Filling Curves,* first ed. Springer, 1994.

[29] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers," *Proc. ACM SIGCOMM '09,* pp. 63-74, Aug. 2009.

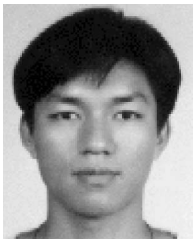
[30] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic Routing in Future Data Centers," *Proc. ACM SIGCOMM '10,* pp. 51-62, Aug. 2010.

[31] S. Surana, B. Godfrey, K. Lakshminarayanan, R. Karp, and I. Stoica, "Load Balancing in Dynamic Structured P2P Systems," *Performance Evaluation,* vol. 63, no. 6, pp. 217-240, Mar. 2006.

[32] S. Iyer, A. Rowstron, and P. Druschel, "Squirrel: A Decentralized Peer-to-Peer Web Cache," *Proc. 21st Ann. Symp. Principles of Distributed Computing (PODC '02),* pp. 213-222, July 2002.

[33] I. Raicu, I.T. Foster, and P. Beckman, "Making a Case for Distributed File Systems at Exascale," *Proc. Third Int'l Workshop Large-Scale System and Application Performance (LSAP '11),* pp. 11-18, June 2011.

[34] Ubuntu, http://www.ubuntu.com/, 2012.



Hung-Chang Hsiao received the PhD degree in computer science from National Tsing Hua University, Taiwan, in 2000. Currently, he is a professor in computer science and information engineering, National Cheng Kung University, Taiwan, since August 2012. He was also a postdoctoral researcher in computer science, National Tsing Hua University, from October 2000 to July 2005. His research interests include distributed computing, and randomized algorithm design and performance analysis. He is a member of the IEEE and the IEEE Computer Society.

Hsueh-Yi Chung received the BS degree in computer science and engineering at Tatung University, Taiwan, in 2009, and the MS degree in computer science and information engineering from National Cheng Kung University, Taiwan, in 2012. His research interests include cloud computing and distributed storage.

Haiying Shen received the BS degree in computer science and engineering from Tongji University, China, in 2000, and the MS and PhD degrees in computer engineering from Wayne State University, in 2004 and 2006, respectively. Currently, she is an assistant professor in the Holcombe Department of Electrical and Computer Engineering at Clemson University. Her research interests include distributed and parallel computer systems and computer networks, with an emphasis on peer-to-peer and content delivery networks, mobile computing, wireless sensor networks, and grid and cloud computing. She was the program cochair for a number of international conferences and member of the Program Committees of many leading conferences. She is a Microsoft Faculty fellow of 2010 and a member of the ACM and the IEEE.

Yu-Chang Chao received the BS degree in computer science and information engineering from Tamkang University, Taipei, Taiwan, and the MS degree in computer science and information engineering from Nation Chen Kung University, Tainan, Taiwan, 2000. He is now a research staff member in Industrial Technology Research Institute (ITRI). His research interests include cloud computing, home networking, and multimedia networking.