LOAD BALANCING FOR DISTRIBUTED FILE SYSTEMS

K. SHANMUGA SUNDARAM

Dr. M.G.R Educational and Research Institute
kshansunder@gmail.com

Abstract— In Distibuted 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 distributed 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.

Keywords- Distributed file systems, Load balancing,Cloud environment.

I. INTRODUCTION

In distributed file system environment, clients can dynamically allocate their resources on-demand without sophisticated deployment and management of resources. Key enabling technologies for clouds include the Map Reduce programming paradigm [1], distributed file systems (e.g., [3], [4]), 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 Map Reduce 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 Map Reduce tasks can be performed in parallel over the nodes. For example, consider a word count 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., chunk servers). 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 [4]. 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 [3] and Hadoop HDFS [4]) 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, 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 [15] suggests an architecture with multiple name nodes (i.e., the nodes managing the metadata information). Its file system namespace is statically and manually partitioned to a number of name nodes. However, as the workload experienced by the name nodes may change over time and no adaptive workload consolidation and/or migration scheme is offered to balance the loads among the name nodes, any of the name nodes may become the performance bottleneck.

In this paper, we are interested in studying the load re-balancing problem in distributed file systems specialized for large-scale, dynamic and data-intensive clouds. (The terms "rebalance" and "balance" are interchangeable in this paper.) dynamism, simplifying the system provision.

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 beenextensively studied (e.g., [27]–[30], [33]–[38]). 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.

II. LOAD BALANCING PROBLEM

We consider a large-scale distributed file system consisting of a set of chunk servers V in a cloud, where the cardinality of V is |V | = n. Typically, n can be one thousand, ten thousand, or more. In the system, a number of files are stored in the n chunk servers. First, let's denote the set of files as F .Each file f ∈ F is partitioned into a number of disjointed, fixed-size chunks denoted by Cf . For example, each chunk has the same size, 64 Mbytes, in Hadoop HDFS [4]. Second, the load of a chunk server is proportional to the number of chunks hosted by the server [4]. Third, node failure is the norm in such a distributed system, and the chunk servers 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 chunk servers. Fig. 1 illustrates an example of the load rebalancing problem with the assumption that the chunk servers 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.

$$V = \frac{\sum_{f \in F} |N|}{n}$$

Let A be the ideal number of chunks that any chunkserver i ∈ V is required to manage in a system-wide load-balanced state, that is,

$$A = \frac{\sum_{f \in F} |C|}{n}$$

Then, our load rebalancing algorithm aims to minimize the load imbalance factor in each chunk server I as follows,

$$L_i - A_i$$

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

III. PROPOSAL

A. Architecture

The chunkservers in our proposal are organized as a DHT network; that is, each chunkserver implements a DHT protocol such as Chord [18] or Pastry [19]. 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 path-name 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 1 2 3 in V by 1/n,2/n,….n/n; for short, denote the n chunkservers n as 1, 2, 3, · · · , 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 i(i−1 , n], except for chunkserver n, which manages the chunks n1 whose handles are in (n , n].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) [18], [19] if each chunkserver I maintains log2 n neighbors, that is,

nodes i + 2k mod n for k = 0, 1, 2, · · · , log2 n − 1. Among the log2 n neighbors, the one i + 20 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 pro-

posal 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; 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 [18], [1] 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 [22] 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 [3] and Hadoop HDFS [4], 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 [3] 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 −

1O(n) [25], [33], [35], thus balancing the loads of nodes to a certain extent. However, our proposal presented in Section III-B performs well for both uniform and non- uniform 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 II is N P-hard, 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.

B. Load balancing Algorithm

1) Overview: A large-scale distributed file system is in a load-balanced state if each chunkserver hosts no more than 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 $(1 - \Delta L)A$ (where $0 \leq \Delta L < 1$). In contrast, a heavy node manages the number of chunks greater than $(1+\Delta U)A$, where $0 \leq \Delta U < 1$ ΔL and ΔU 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 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 $\min\{L_j - A, A\}$ chunks from j. That is, node i requests A chunks from the heaviest node j if j's load exceeds 2A; otherwise, i requests a load of $L_j - A$ 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 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-k1 underloaded nodes with the top-k2 overloaded nodes. We use U to denote the set of top-k1 underloaded nodes in the sorted list of underloaded nodes, and use O to denote the set of top-k2 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 k2 -th ($k2 \leq k2$) most loaded node in O is the k1 –th

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

where the summation denotes the sum of the excess loads in the top-k2 heavy nodes. It means the top-k1 light nodes should leave and rejoin as successors of the top –k2 overloaded nodes.

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 Section III-B2. Section III-B3 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 III-B4. 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 III-B5 discusses the maintenance of replicas for each file chunk..

2) Basic Algorithms: Algorithms 1 and 2 (see Appendix C) detail our proposal; Algorithm 1 species 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-k1 light nodes with the top-k2 heavy nodes is clearly challenging. We tackle this challenge by enabling a node to execute the load balancing algorithm introduced in Section III-B1 based on a sample of nodes. In the basic algorithm, each node implements the gossip-based aggregation protocol in [26], [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 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 Ai and regards it as an estimation of 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 k1 in the sorted list, i.e., it is the top-k1 underloaded node in the list (Lines 3-5 in Algorithm 1). Node i finds the top-k2 overloaded nodes in the list such that the sum of these nodes' excess loads is the least greater than or equal to k1 Ai (Line 6 in Algorithm 1). Formula (ii) in the algorithm is derived from Eq. (3). The complexity of the step in Line 6 is O(|V|).

Then, the k2 -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(|V| log |V|).

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. 2(a). Assume $\Delta L = \Delta U = 0$ in the example. Then, nodes N 1, N 2, N 3, N 4 and N 5 are light, and nodes N 6, N 7, N 8, N 9, and N 10 are heavy. Each node performs the load balancing algorithm independently, and we choose N 1 as an example to explain the load balancing algorithm. N 1 first queries the loads of N 3, N 6, N 7, and N 9 selected randomly from the system (Fig. 2(b)). Based on the samples, N 1 estimates the ideal load A (i.e., AN 1 = LN 1 +LN 3 +LN 6 +LN 7 +LN 9). It notices that it is a light5 node. It then finds the heavy node it needs to request chunks. The heavy node is the most loaded node (i.e., N 9) as N 1 is the lightest among N1 and its sampled nodes {N 3, N 6, N 7, N 9} (Line 6 in Algorithm 1). N 1 then sheds its load to its successor N 2, departs from the system, and rejoins the system as the successor of N 9. N 1

allocates min{LN 9 − AN 1 , AN 1 }= AN 1 chunks from N 9 (Lines 5 and 6 in Algorithm 2). In the example, N 4 also performs the load rebalancing algorithm by first sampling {N 3, N 4, N 5, N 6, N 7} (Fig. 2(d)). Similarly, N 4 determines to rejoin as the successor of N 6. N 4 then migrates its load to N 5 and rejoins as the successor of N 6 (Fig. 2(e)). N 4 requests min{LN 6 − AN 4 , AN 4 }= L6 − AN 4 be k (where k ≤ |V | = n).

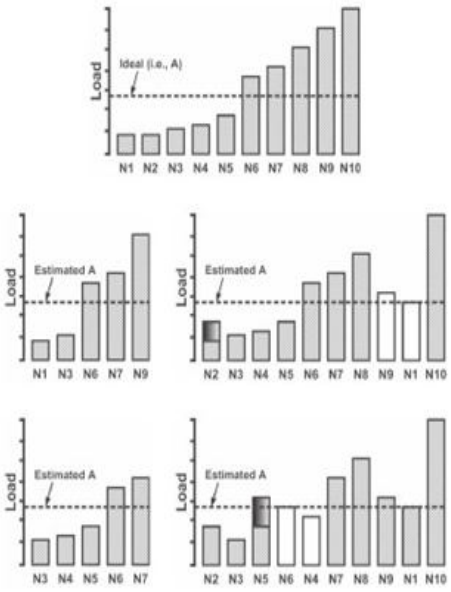

Fig 2

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).

To demonstrate Algorithm 3, consider the example shown in Fig. 2. Let nV = 2. In addition to the sample set V1 = {N 1, N 3, N 6, N 7, N 9} (Fig. 2(b)), N 1 gathers another sample set, say, V2 = {N 1, N 4, N 5, N 6, N 8}. N 1 identifies the heavy node N 9 in V1 and N 8 in V2 . Suppose N 9 is physically closer to N 1 than N 8. Thus, N 1 rejoins as a successor of N 9 and then receives chunks from N 9. Node i also offloads its original load to its successor. For example, in Figs. 2(a) and (b), node N 1 migrates its original load to its successor node N 2 before N 1 rejoins as node N 9'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 four-dimensional 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 four-dimensional 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.

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 ≈ 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 for the analytical model that details the performance of the locality-oblivious and locality-aware approaches discussed in this section. Moreover, in Appendix E, the effects of the different numbers of racks in centralized matching and the different numbers of node vectors n 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 [28]–[30]. 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 distributed matching.

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 [33] 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.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in Proc. 6th Symp. Operating System Design and Implementation (OSDI'04), Dec. 2004, pp. 137–150.

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

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

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

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

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

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

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

[9]HDFSFederation,http://hadoop.apache.org/common/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/Federation.htm

[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. Netw., 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," LNCS 2218, 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," in Proc. 21st ACM Symp. Operating Systems Principles (SOSP'07), Oct. 2007, pp. 205–220.

[13] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems," in Proc. 2nd Int'l Workshop Peer- to-Peer Systems (IPTPS'02), Feb. 2003, pp. 68–79.

[14] D. Karger and M. Ruhl, "Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems," in Proc. 16th ACM Symp. Parallel Algorithms and Architectures (SPAA'04), June 2004, pp. 36–48

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

[16] J. W. Byers, J. Considine, and M. Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables," in Proc. 1st Int'l Workshop Peer-to-Peer Systems (IPTPS'03), Feb. 2003, pp. 80–87.

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

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

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

[20] H. Shen and C.-Z. Xu, "Locality-Aware and Churn-Resilient Load Bal-ancing Algorithms in Structured P2P Networks," IEEE Trans. Parallel Distrib. Syst., 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. Knowl.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 Distrib. Syst., 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,"LNCS 1518, pp. 159–170, Oct. 1998.

[26] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-Based Aggregation in Large Dynamic Networks," ACM Trans. Comput. Syst., 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. Comput. Syst., vol. 25, no. 3, Aug. 2007.

[28] H. Sagan, Space-Filling Curves, 1st 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," in Proc. ACM SIGCOMM'09, Aug. 2009,

★★★