IJCAT - International Journal of Computing and Technology, Volume 3, Issue 5, May 2016
ISSN : 2348 - 6090
www.IJCAT.org

An Efficient Approach for Load Balancing P2P System

[1] Najim Sheikh, [2] Dr. Sachin Choudhari

[1] M.Tech RGPV

[2] Principal, SBITM Betul

Abstract - Most P2P systems that provide a DHT abstraction distribute objects randomly among "peer nodes" in a way that results in some nodes having $\Theta(\log N)$ times as many objects as the average node. Further imbalance may result due to non-uniform distribution of objects in the identifier space and a high degree of heterogeneity in object loads and node capacities. Additionally, a node's load may vary greatly over time since the system can be expected to experience continuous insertions and deletions of objects, skewed object arrival patterns, and continuous arrival and departure of nodes. In this paper, we propose an algorithm for load balancing in such heterogeneous, dynamic P2P systems. Our simulation results show that in the face of rapid arrivals and departures of objects of widely varying load, our algorithm achieves load balancing for system utilizations as high as 90% while moving only about 8% of the load that arrives into the system. Similarly, in a dynamic system where nodes arrive and depart, our algorithm moves less than 60% of the load the underlying DHT moves due to node arrivals and departures. Finally, we show that our distributed algorithm performs only negligibly worse than a similar centralized algorithm, and that node heterogeneity helps, not hurts, the scalability of our algorithm.

Keywords – *Load Balancing, P2P Systems.*

1. Introduction

The last several years have seen the emergence of a class of structured peer-to-peer systems that provide a distributed hash table (DHT) abstraction ([1], [2], [3], [4]). In such structured systems, a unique identifier is associated with each data item and each node in the system. The identifier space is partitioned among the nodes that form the peer-to-peer (P2P) system, and each node is responsible for storing all the items that are mapped to an identifier in its portion of the space. Thus, the system provides an interface consisting of two functions: *put*(*id, item*), which stores an item, associating with it a given identifier *id*; and *get*(*id*) which retrieves the item with the identifier *id*. Consider a system with N nodes. If node and item identifiers are randomly chosen as assumed in [1], [2], [3], [4], there is a $\Theta(\log N)$ imbalance factor in the number of items stored at a node. Furthermore, if applications associate semantics with the item IDs, the imbalance factor can become arbitrarily bad since IDs would no longer be uniformly distributed. For example, a database application may wish to store all tuples (data items) of a relation according to the primary key using the tuple keys as IDs. This would allow the application to efficiently implement range querying (i.e., finding all items with keys in a given interval) and sorting operations, but would assign all the tuples to a small region of the ID space. In addition, the fact that in typical P2P systems, the capabilities of nodes (storage and bandwidth) can differ by multiple orders of magnitude further aggravates the problem of load imbalance.

Several solutions have been proposed to address the load balancing problem [2], [5], [6]. However, these all assume that the system is static and most assume that the IDs of both nodes and items are uniformly distributed. In this paper, we present a solution for a system in which these assumptions do not hold. In particular, we consider a system in which

- data items are continuously inserted and deleted,
- nodes join and depart the system continuously, and
- The distribution of data item IDs and item sizes can be skewed.

Our algorithm uses the concept of *virtual servers* previously proposed in [7]. A virtual server represents a peer in the DHT; that is, the storage of data items and routing happen at the virtual server level rather than at the physical node level. A physical node hosts one or more virtual servers. Load balancing is achieved by moving virtual servers from heavily loaded physical nodes to lightly loaded physical nodes.

In this paper we make the following contributions:

1) We propose an algorithm which to the best of our knowledge is the first to provide dynamic load balancing in heterogeneous, structured P2P systems.

2) We study the proposed algorithm by using extensive simulations over a wide set of system scenarios and algorithm parameters.

Our main results are as follows:

1) Our simulations show that in the face of object arrivals and departures and system utilizations as high as 90%, the algorithm achieves a good load balance while moving only about 8% of the load that arrives into the system. Furthermore, in a dynamic system where nodes arrive and depart, our algorithm moves less than 60% as much load as the underlying DHT moves as a result of node arrivals and departures.
2) Our algorithm produces a 99.9th percentile node utilization less than 3% higher than a similar fully centralized load balancer, showing that the price of decentralization is negligible.
3) Heterogeneity of node capacity allows us to use fewer virtual servers per node than in the equal-capacity case, thus increasing the scalability of the system.
4) The rest of the paper is organized as follows. In Section II, we formulate the load balancing problem more explicitly and discuss what resources we may balance effectively. In Section III, we discuss background material, including our use of virtual

Servers and our previous load balancing schemes given in [5]. In Section IV, we describe our algorithm for load balancing in dynamic P2P systems, and we evaluate its performance through simulation in Section V. We discuss future directions in Section VI, related work in Section VII, and conclude in Section VIII.

2. Problem Formulation and Motivation

2.1 Definitions and Goals

Each object (data item) that enters the system has an associ-ated *load*, which might represent, for example, the number of bits required to store the object, the popularity of the object, or the amount of processor time needed to serve the object. Thus, we do not assume a particular resource, but we assume that there is only one bottleneck resource in the system, leaving multi-resource balancing to future work.

Each object also has a *movement cost*, which we are charged each time we move the object between nodes. We assume this cost is the same regardless of which two nodes are involved in the transfer. One can think of this cost as being proportional to the object's storage size. An object's load may or may not be related to its movement cost.

The *load* $\__i$ of a node i at a particular time is the sum of the loads of the objects stored on that node at that time. Each node i has a fixed *capacity* $c_i > 0$, which might represent, for example, available disk space, processor speed, or bandwidth. A node's *utilization* u_i is the fraction of its capacity that is used: $u_i = \__i / c_i$. The *system utilization* μ is the fraction of the system's total capacity which is used:

$$\mu = \frac{\text{nodes } n \quad \bar{} \quad n}{}.$$

When $u_n > 1$, we say that node n is *overloaded*; otherwise node i is said to be *under loaded*. We use *light* and *heavy* to informally refer to nodes of low or high utilization, respectively.

A load balancing algorithm should strive to achieve the following (often conflicting) goals:

- **Minimize the load imbalance.** To provide the best quality of service, every node would have the same utilization. Furthermore, for resources with a well-defined cliff in the load-response curve, it is of primary importance that no node's load is above the load at which the cliff occurs. We can take this point to be the capacity of the node.
- **Minimize the amount of load moved.** Moving a large amount of load uses bandwidth and may be infeasible if a node's load changes quickly in relation to the time needed to move objects.

We formalize these goals in Section V.

2.2 Relevance of Load Balancing

In this subsection, we answer two natural questions with respect to the relevance of load balancing in the context of two particular resources: storage and bandwidth.

Is load balancing of storage feasible? This question is raised by the huge disparity between the storage capacity of the end-hosts and the access bandwidth in a wide area

network. Even if the end-hosts have broadband connectivity (cable modem or DSL), it may take more than one hour to transfer 1 GB of data, which is not a large amount of data considering the fact that notebooks today come with 20-30 GB disks.[1] Thus in many situations the amount of data movement necessary to significantly improve the load balance might not be achievable quickly enough. In spite of this fact, we believe that it is feasible to balance storage in other contexts in which DHTs are useful, such as in data centers with thousands or tens of thousands of machines connected by very high speed network connections (e.g., > 1 Gbps).

Why use load balancing for bandwidth? For the purposes of relieving hot-spots, an alternative to load balancing is replication (caching). Why not replicate a popular data item instead of shifting sole responsibility for the popular item to a more powerful node? Replication, though a good solution in the case of immutable data, would require complex algorithms to maintain data consistency in the case of mutable data. Fur-thermoses, many peer-to-peer systems are highly heterogeneous. The uplink capacity of a home user and the uplink capacity of a host on a university network can differ by as much as two orders of magnitude. Thus, moving a data item to a well-connected machine would be equivalent to generating and maintaining as many as 100 replicas of that data item, which may add significant overhead. Finally, we note that replication and load balancing are orthogonal and one can combine them to improve system performance.

3. Background

In this section, we argue for our design decision to use virtual servers as a fundamental unit of load balancing, and describe our earlier load balancing schemes on which the algorithm of this paper is based.

3.1 Use of Virtual Servers

One of the difficulties of load balancing in DHTs is that the load balancer has little control over where the objects are stored. Most DHTs use *consistent hashing* to map objects onto nodes [8]: both objects and nodes in the system are assigned unique IDs in the same identifier space, and an object is stored at the node with the "closest" ID in the space. This associates with each node a region of the ID space for which it is responsible. More generally, if we allow the use of virtual servers, a node may have multiple IDs and therefore owns a set of noncontiguous regions.

Under the assumption that we preserve the use of consistent hashing, the load balancer is restricted to moving load by either remapping objects to different points in the ID space, i.e.,

This is actually an underestimate. One hour corresponds to a bottleneck bandwidth of 1.2 Mbps which is much higher than the uplink bandwidth of DSL and cable modem connections. However, since items are queried by their IDs, changing the ID of an object would make it difficult to locate that object subsequently. Furthermore, some applications compute the ID of an object by hashing its content [7], thus rendering its ID static.

Thus we must change the set of regions associated with a node. Since we wish to avoid large load movement, we need to be able to remove a small fraction of the ID space associated with a node. The design space here is not small and we do not claim that our choice is the only reasonable one. However, our choice is a simple one: we reassign an entire region from one node to another, but ensure that the number of regions (virtual servers) per node is large enough that a single region is likely to represent only a small fraction of a node's load.

One drawback of this approach is that if there are an average of m virtual servers per node, the per-node routing state increases by a factor of m since a node must maintain the links associated with each of its virtual servers (but in Chord, lookup path length does not increase[3]). However, as we will see in Section V-D, we need a relatively modest number of virtual servers per node (e.g., $m = \log N$) to achieve good load balancing and substantially fewer when node capacities are heterogeneous. We believe this overhead is acceptable in practice.

One of the main advantages of using virtual servers for balancing the load is that this approach does not require any changes to the underlying DHT. Indeed, the transfer of a virtual server can be implemented simply as a peer leaving and another peer joining the system. The ID-to-peer (i.e., ID-to-virtual server) and ID-to-object mappings that the underlying DHT performs are unaffected. If a node leaves the system, its share of identifier space is taken over by other nodes which are present in the system just as the underlying DHT would do. In the case of Chord [7], each virtual server v of a node that leaves the system would be taken over by a node that is responsible for a virtual server v^- which immediately succeeds v in the identifier space. Similarly, when a node joins, it picks m random points in the ID space and splits the virtual servers there, thereby acquiring m virtual servers.

We assume that there are external methods to make sure that node departures do not cause loss of data objects. In particular, we assume that there is replication of data objects as proposed

[2]Another approach would be to use indirection: if a large object is hashed onto a heavy node then store only a pointer at the heavy node, and store the object at a light node. However, this approach does not remap responsibility for the object pointer and so would not help when objects are small (e.g., tuples in a database relation). Furthermore, indirection adds complexity and is orthogonal to our solution.

[3]We can compensate for the fact that there are now *mN* peers by using shortcut routing as proposed in [7]: a virtual server may use the outlinks of any of its physical node's virtual servers. To see why this offsets the factor *m* increase in the number of peers, note that we expect *N* virtual servers to lie between each of the *m* virtual servers belonging to a particular node.

in CFS [7], and departure of a node would result in the load being transferred to the neighbors in the identifier space.

3.2 Static Load Balancing Techniques

In a previous paper, we introduced three simple load balancing schemes that use the concept of virtual servers for static systems [5]. Since the algorithm presented in this paper is a natural extension of those schemes, we briefly review them here. The schemes differ primarily in the number and type of nodes involved in the decision process of load balancing.

In the simplest scheme, called *one-to-one*, each lightly loaded node *v* periodically contacts a random node *w*. If *w* is heavily loaded, virtual servers are transferred from *w* to *v* such that *w* becomes light without making *v* heavy.

The second scheme, called *one-to-many*, allows a heavy node to consider more than one light node at a time. A heavy node *h* examines the loads of a set of light nodes by contacting a random *directory node* to which a random set of light nodes have sent their load information. Some of *h*'s virtual servers are then moved to one or more of the lighter nodes registered in the directory.

Finally, in the *many-to-many* scheme each directory maintains load information for a set of both light and heavy nodes. An algorithm run by each directory decides the reassignment of virtual servers from heavy nodes registered in that directory to light nodes registered in that directory. This knowledge of nodes' loads, which is more centralized than in the first two schemes, can be expected to provide a better load balance. Indeed, our results showed that the many-to-many technique performs the best.

Our new algorithm presented in the next section combines elements of the many-to-many scheme (for periodic load bal-acing of all nodes) and of the one-to-many scheme (for emer-gency load balancing of one particularly overloaded node).

4. Load Balancing Algorithm

The basic idea of our load balancing algorithm is to store load information of the peer nodes in a number of *directories* which periodically schedule reassignments of virtual servers to achieve better balance. Thus we essentially reduce the distributed load balancing problem to a centralized problem at each directory.

Each directory has an ID known to all nodes and is stored at the node responsible for that ID. Each node *n* initially chooses a random directory and uses the DHT lookup protocol to report to the directory (1) the loads $\__{v1}, \ldots, \__{vm}$ of the virtual servers for which *n* is responsible and (2) *n*'s capacity c_n. Each directory collects load and capacity information from nodes which contact it. Every *T* seconds, it computes a schedule of virtual server transfers among those nodes with the goal of reducing their maximum utilization to a parameterized *periodic threshold* k_p . After completing a set of transfers scheduled by a directory, a node chooses a new random directory and the process repeats.

When a node *n*'s utilization $u_n = \__n / c_n$ jumps above a parameterized *emergency threshold* k_e , it immediately reports to the directory *d* which it last contacted, without waiting for *d*'s next periodic balance. The directory then schedules immediate transfers from *n* to more lightly loaded nodes.

More precisely, each node *n* runs the following algorithm.

`Node`(time period T , threshold k_e)

- *Initialization:* Send $(c_n, \{\__{v1}, \ldots, \__{vm}\})$ to `RandomDirectory()`
- *Emergency action:* When u_n jumps above k_e:
  1) Repeat up to twice while $u_n > k_e$:
  2) $d \leftarrow$ `RandomDirectory()`
  3) Send $(c_n, \{\__{v1}, \ldots, \__{vm}\})$ to d
  4) `PerformTransfer`(v, n') for each transfer $v \rightarrow n'$ scheduled by d
- *Periodic action:* Upon receipt of list of transfers from a directory:
  1) `PerformTransfer`(v, n') for each transfer $v \rightarrow n'$
  2) Report $(c_n, \{\__{v1}, \ldots, \__{vm}\})$ to `RandomDirectory()`

In the above pseudocode, `RandomDirectory()`

selects two random directories and returns the one to which fewer nodes have reported since its last periodic balance. This reduces the imbalance in number of nodes reporting to directories. `PerformTransfer`(v, n^-) transfers virtual server v to node n^- if it would not overload n^-, i.e. if $\__n + \__v \leq c_n$. Thus a transfer may be aborted if the directory scheduled a transfer based on outdated information (see below).

Each directory runs the following algorithm.

Directory(time period T , thresholds k_e , k_p)

- *Initialization:* $I \leftarrow \{\}$
- *Information receipt and emergency balancing:* Upon receipt of $J = (c_n , \{\_v1 , . . . , \_vm \})$ from node n:
  1) $I \leftarrow I \cup J$
  2) If $u_n > k_e$:
  3) *reassignment* ← ReassignVS(I, k_e)
  4) Schedule transfers according to *reassignment*
- *Periodic balancing:* Every T seconds:
  1) *reassignment* ← ReassignVS(I, k_p)
  2) Schedule transfers according to *reassignment*
  3) $I \leftarrow \{\}$

The subroutine ReassignVS, given a threshold k and the load information I reported to a directory, computes a reassignment of virtual servers from nodes with utilization greater than k to those with utilization less than k. Since computing an optimal such reassignment.

Algorithm runs in $O(m \log m)$ time, where m is the number of virtual servers that have reported to the directory.

`ReassignVS`(Load & capacity information I, threshold k)

1) $pool \leftarrow \{\}$
2) For each node $n \in I$, while $\__n / c_n > k$, remove the least loaded virtual server on n and move it to $pool$.
3) For each virtual server $v \in pool$, from heaviest to lightest, assign v to the node n which minimizes $(\__n + \__v)/c_n$.
4) Return the virtual server reassignment.

We briefly discuss several important design issues.

Periodic vs. emergency balancing. We prefer to schedule transfers in large periodic batches since this gives `ReassignVS` more flexibility, thus producing a better balance. However, we do not have the luxury to wait when a node is (about to be) overloaded. In these situations, we resort to emergency load balancing. See Section V-A for a further discussion of these issues.

Choice of parameters. We set the emergency balancing threshold k_e to 1 so that load will be moved off a node when load increases above its capacity. We compute the periodic threshold k_p dynamically based on the average utilization $\hat{\mu}$ of the nodes reporting to a directory, setting $k_p = (1+ \hat{\mu})/2$. Thus directories do not all use the same value of k_p . As the names of the parameters suggest, we use the same time period T be-tween nodes' load information reports and directories' periodic balances. These parameters control the tradeoff between low load movement and low quality of balance: intuitively, smaller values of T , k_p , and k_e provide a better balance at the expense of greater load movement.

Stale information. We do not attempt to synchronize the times at which nodes report to directories with the times at which directories perform periodic balancing. Indeed, in our simulations, these times are all randomly aligned. Thus, directories do not perform periodic balances at the same time, and the information a directory uses to decide virtual server reassignment may be up to T seconds old.

5. Evaluation

We use extensive simulations to evaluate our load balancing algorithm. We show.

- the basic effect of our algorithm, and the necessity of emergency action (Section V-A);
- the tradeoff between low load movement and a good balance, for various system and algorithm parameters (Section V-B);
- the number of virtual servers necessary at various system utilizations (Section V-C);
- the effect of node capacity heterogeneity, concluding that we can use many fewer virtual servers in a heterogeneous system (Section V-D);
- the effect of no uniform object arrival patterns, showing that our algorithm is robust in this case (Section V-E);

the effect of node arrival and departure, concluding that our load balancer never moves more than 60% as much load as the underlying DHT moves due to node arrivals and departures (Section V-F); and

- the effect of object movement cost being unrelated to object load, with the conclusion that this variation has little effect on our algorithm (Section V-G).

Metrics. We evaluate our algorithm using two primary metrics:

1) *Load movement factor*, defined as the total movement cost incurred due to load balancing divided by the total cost of moving all objects in the system once. Note that since

   the DHT must move each object once to initially insert it, a load movement factor of 0.1 implies that the balancer consumes 10% as much bandwidth as is required to insert the objects in the first place.

2) *99.9th percentile node utilization*, defined as the maxi-mum over all simulated times t of the 99.9th percentile of the utilizations of the nodes at time t. Recall from Section II that the utilization of node i is its load divided by its capacity: $u_i = \__i / c_i$.

The challenge is to achieve the best possible tradeoffs between these two conflicting metrics.

Simulation methodology. Table I lists the parameters of our event-based simulated environment and of our algorithm, and the values to which we set them unless otherwise specified.

We run each trial of the simulation for $20T$ simulated seconds, where T is the parameterized load balance period. To allow the system to stabilize, we measure 99.9th percentile node utilization and load movement factor only over the time period $[10T, 20T]$. In particular, in calculating the latter metric, we do not count the movement cost of objects that enter the system, or objects that the load balancer moves, before time $10T$. Finally, each data point in our plots represents the average of these two measurements over 5 trials.

A. Basic effect of load balancing

Figure 1 captures the tradeoff between load movement and 99.9th percentile node utilization. Each point on the lower line corresponds to the effects of our algorithm with a particular choice of load balance period T . For this and in subsequent plots wherein we vary T , we use $T \in \{60, 120, 180, 240, 300, 600, 1200\}$. The intuitive trend is that as T decreases (moving from left to right along the line), 99.9th percentile node utilization decreases but load movement factor increases. One has the flexibility of choosing T to compromise between these two metrics in the way which is most appropriate for the target application.

The upper line of Figure 1 shows the effect of our algorithm with emergency load balancing turned off. Without emergency balancing, for almost all nodes' loads to stay below some threshold, we must use a very small load balancing period T so that it is unlikely that a node's load rises significantly between periodic balances. This causes the algorithm to move significantly more load, and demonstrates the desirability of

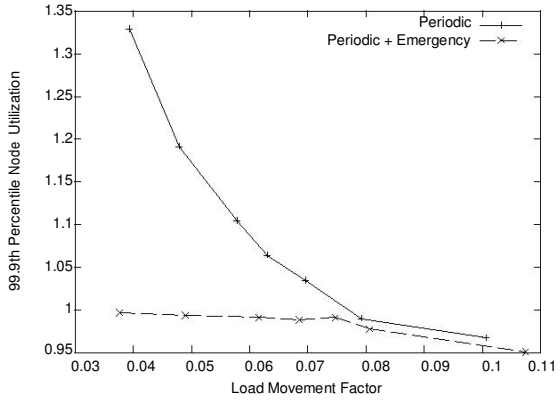

Fig. 1. 99.9th percentile node utilization vs. load moved, for our periodic+emergency algorithm and for only periodic action.

Emergency load balancing. In the simulations of the rest of this paper, emergency balancing is enabled as in the description of our algorithm in Section IV.

B. Load movement vs. 99.9th percentile node utilization

With a basic understanding of the tradeoff between our two metrics demonstrated in the previous section, we now explore the effect of various environment and system parameters on this tradeoff.

In Figure 2, each line corresponds to particular system utilization, and as in Figure 1, each point represents a particular choice of T between 60 and 1200 seconds. Even for system utilizations as high as 0.9, we are able to keep 99.9 percent of the nodes under loaded while incurring a load movement factor of less than 0.08.

Figure 3 shows that the tradeoff between our two metrics

gets worse when the system contains fewer objects of com-mensurately higher load, so that the total system utilization is constant. Nevertheless, for at least 250, 000 objects, which corresponds to just 61 objects per node, we achieve good load balance with a load movement factor of less than 0.11.[4] Note that for 100, 000 objects, the 99.9th percentile node utilization extends beyond the range of the plot, to less than 2.6. However, only a few nodes are overloaded: the 99.5th percentile node utilization for 100, 000 objects (not shown) stays below 1.0 with a load movement factor of 0.22. In any case, we believe that our default choice of 1 million objects is reasonable.

Figure 4 shows that the number of directories in the system has only a small effect on our metrics. For a particular load movement factor, our default choice of 16 directories produces a 99.9th percentile node utilization less than 3% higher than in the fully centralized case of 1 directory.

C. Number of virtual servers

Figures 5 and 6 plot our two metrics as functions of system utilization. Each line corresponds to a different average (over

[4]The spike above utilization 1 in the 500, 000-object line is due to a single outlier among our 5 trials.

slowed significantly. For homogeneous nodes and objects and

a static system, picking $d = 2$ achieves a load balance within a log log N factor of optimal, and when $d = \Theta(\log N)$ the

load balance is within a constant factor of optimal. However, this scheme was not analyzed or simulated for the case of heterogeneous object sizes and node capacities, and in any case is not prepared to handle a dynamic system of the kind which we have described. This is largely complementary to the work presented in this paper.

Adler et al [9] present a DHT which provably ensures that, as nodes join the system, the ratio of loads of any two nodes is $O(1)$ with high probability. The system is organized as a tree, with additional links for routing in a hypercube topology. A joining node considers a small set of leaf nodes of the tree and joins the system by splitting an appropriately chosen leaf. However, no analysis of node departure was given and the system does not deal with varying node capacity or object distribution.

Karger and Ruhl [10] propose algorithms which dynamically balance load among peers without using multiple virtual servers by reassigning lightly loaded nodes to be neighbors of heavily loaded nodes. However, they do not fully handle the case of heterogeneous node capacities, and while they prove bounds on maximum node utilization and load movement, it is unclear whether their techniques would be efficient in practice.

Douceur and Wattenhofer [11] have proposed algorithms for replica placement in a distributed file system which are similar in spirit with our algorithms. However, their primary goal is to place object replicas to maximize the availability in an untreated P2P system, while we consider the load balancing problem in a cooperative system. Triantafillou *et al.* [12] have recently studied the problem of load balancing in the context of content and resource management in P2P systems. However, their work considers an unstructured P2P system, in which meta-data is aggregated over a two-level hierarchy.

There is a large body of theoretical work in load balancing problems similar to ours in that they seek to minimize both maximum load and amount of load moved. This includes Aggarwal et al [13] in an offline setting similar to that of our periodic load balancer, and Westbrook [14], Andrews et al [15], and others (see Azar's survey [16]) in an online setting. It would be interesting to study whether these algorithms can be adapted to our system.

90% while transferring only about 8% of the load that arrives in the system, and performs only slightly less effectively than a similar but fully centralized balancer. In addition, we found that heterogeneity of the system can improve scalability by reducing the necessary number of virtual servers per node as compared to a system in which all nodes have the same capacity.

References

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *Proc. ACM SIGCOMM*, San Diego, 2001.

[2] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proc. ACM SIGCOMM*, San Diego, 2001, pp. 149–160.

[3] Kris Hildrum, John D. Kubatowicz, Satish Rao, and Ben Y. Zhao, "Distributed Object Location in a Dynamic Network," in *Proc. ACM SPAA*, Aug. 2002.

[4] Antony Rowstron and Peter Druschel, "Pastry: Scalable, Dis-tributed Object Location and Routing for Large-scale Peer-to-Peer Systems," in *Proc. Middleware*, 2001.

[5] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica, "Load Balancing in Structured P2P Systems," in *Proc. IPTPS*, Feb. 2003.

[6] John Byers, Jeffrey Considine, and Michael Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables," in *Proc. IPTPS*, Feb. 2003.

[7] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, "Wide-area Cooperative Storage with CFS," in *Proc. ACM SOSP*, Banff, Canada, 2001.

[8] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proc. ACM STOC*, May 1997.

[9] M. Adler, Eran Halperin, R. M. Karp, and V. Vazirani, "A stochastic process on the hypercube with applications to peer-to-peer networks," in *Proc. STOC*, 2003.