A Scheme for Balancing Heterogeneous Request Load in DHT-based P2P Systems

Juan Li
University of British Columbia
2366 Main Mall
Vancouver, B.C.
Canada
juanli@cs.ubc.ca

Billy Cheung
University of British Columbia
2366 Main Mall
Vancouver, B.C.
Canada
bccheung@cs.ubc.ca

Son Vuong
University of British Columbia
2366 Main Mall
Vancouver, B.C.
Canada
vuong@cs.ubc.ca

ABSTRACT

DHT-based P2P systems have been proven to be a scalable and efficient means of sharing information. With the entrance of quality of services concerns into DHT systems, however, the ability to guarantee that the system will not be overwhelmed due to load imbalance becomes much more significant, especially when factors such as item popularity and skewing are taken into consideration. In this paper, we focus on the problem of load imbalance caused by skewed access distribution. We propose an effective load balancing solution, which takes the peer heterogeneity and access popularity into account to determine the load distribution. Our algorithm achieves load balancing by dynamically balancing the query routing load and query answering load respectively. Experimentations performed over a Pastry-like system illustrate that our balancing algorithms effectively balance the system load and significantly improves performance.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *distributed applications.*

General Terms

Algorithms, Performance

Keywords

Load-balancing, distributed hash tables (DHT), peer-to-peer (P2P) systems, quality of service (QoS)

1. INTRODUCTION

Distributed Hash Tables (DHTs) [1][2][3][4] provide a reliable, scalable, fault tolerant and efficient way to manage P2P networks. Typically employing some variant of consistent hashing to associate keys with nodes, each node is mapped to a unique ID and owns the set of objects whose IDs are "closest" to it, while object lookup queries consist of following well-defined paths from a querying node to a destination node that holds the index entries pertaining to the query. In theory, DHTs can provide fair and scalable service because they achieve a balanced partition of workload amount the nodes in the system. In practice however, this is not always the case. Given the heterogeneous nature of individual peer capacity in a P2P network (due to non-uniform computational power, storage capacity and network bandwidth difference between peers), even a uniform workload distribution amongst peers can still lead to load imbalance problems, additionally encumbered by the fact that the consistent hash used by DHTs can cause certain peers to have up to O(logN) times as many objects as the average peer in the network [3], intensifying the imbalance.

Furthermore, since objects and queries within the system tend to be skewed [5][6] (i.e. certain objects are significantly more popular than others), heavy lookup traffic load is experienced at the peers responsible for popular objects, as well as at the intermediary nodes on the lookup paths to those peers. When subsequent tasks are then obliviously assigned to the already overloaded node, the average response time consequently increases drastically. This paper aims at balancing the highly unbalanced load caused by skewed object and query distribution through the use of a comprehensive balancing mechanism, which includes an adaptive load redistribution scheme as well as a dynamic routing table reconfiguring scheme.

2. RELATED WORK

There have been many load balancing schemes proposed for DHT-based systems. Roughly, we divide them into four categories:

The **virtual server** approach [7][8][9][18] focuses on the imbalance of the key distribution due to the hash function. Each physical node instantiates O(logN) number of virtual servers with random IDs that act as peers in the DHT, which reduces the load imbalance to a constant factor. To address peer heterogeneity, each node selects a number of virtual servers to create proportional to its capacity. Unfortunately, the usage of virtual servers greatly increases the amount of routing metadata needed on each peer and causes more maintenance overhead. In addition, the number of hops per lookup (and latencies) increases. Moreover, it doesn't take object popularity into account.

Unlike the virtual server approach, the **dynamic ID** approach uses just a single ID per node [10][11][12][19]. The load of a peer can be adjusted by choosing a suitable ID in the namespace. However, all such solutions requires IDs to be reassigned to maintain load balance as nodes dynamically join and leave the system, resulting in a high overhead because it involves transferring objects and updating overlay links.

The third class of approaches uses **multiple hash functions** to balance the load. The *power of two choices* [13] uses two or more hash functions to map a key to multiple nodes and store the key on the peer that is the least loaded. In the *k-choice* [14] load balancing algorithm, the node uses multiple hashes to generate a set of IDs and at join time selects an ID in a way to minimize the discrepancies between capacity and load for itself and the nodes that will be affected by its join time. While such a strategy is simple and efficient, it increases the computational overhead for publishing and retrieving content, since multiple hash functions have to be computed each time; in addition, it is a static allocation, and does not change in the case that the workload distribution shifts.

The last category of balancing schemes is by **caching and replication** [2][3][20]. Hotspots and dynamic streams are handled by using caches to store popular objects in the network, and lookups are considered resolved whenever cache hits occur along the path. Pastry [2] and Chord [3] replicate an object on the k servers whose identifiers are closest to the object key in the namespace to improve the availability, but it also help balance the load of a popular topic. Unfortunately, the last few hops of a lookup are precisely the ones that can least be optimized [15]. Moreover, since the query load is dynamic, a fixed number of replicas do not work well; if the number is chosen too high, then resources may be wasted, and if it is set too low, then these replicas may not be enough to support a high query load.

3. ADAPTIVE LOAD BALANCING SCHEME

In this section, we detail our load balancing scheme, focusing on the imbalance caused by heterogeneous object popularity. We propose a comprehensive load balancing strategy, which address this problem by dynamically re-distributing the load of hot spots to other 'cold spots'. Particularly, we distinguish two types of load: query answering load and query forwarding load (query load and routing load for short). Aiming at balancing these two kinds of load, three balancing strategies have been proposed: (1) adaptive object replication scheme, which targets balancing the query load; and (2) adaptive routing replication and (3) dynamic routing table reconfiguration, both aimed at balancing the system's routing load. Each node analyzes the main cause of its overloading and uses a particular balancing algorithm to correct its situation.

3.1 Load Metric

Our load balancing scheme involves a load metric to gauge the activity of each peer node and make the necessary adjustments. Each peer p in the network has a capacity C for serving requests, which corresponds to the maximum amount of load that it can support. In our paper, this is derived from the maximum number of queries that can be routed, answered, or queued per second by the peer. It is assumed that any arriving traffic that can not be either processed or queued by the peer is dropped. It is also assumed that nodes will be able to define their capacity consistently via a globally ratified/used metrics scale.

At any given time, the load of peer p is defined as the number of requests received per unit of time. We focus on two kinds of requests: the query routing request, and query answering requests. On receiving a routing request, the peer checks its routing table and forwards the query to next hop. If it receives a query answering request (meaning that it has a locally stored solution to that request), it serves that request according to the application's needs (For example, answering a complex query, or transferring a file, and so on). In this paper, the current load value L of a node is defined in Equation (1) as the sum of its current routing load and its current query load:

$$L = Lr + Lq \tag{1}$$

$$L = (a \times \sum q_i + b \times \sum r_j) \times l \tag{2}$$

Both the routing and query load can be represented by the number of requests received in unit time. Assuming that the unit load is l, and each routing request creates a unit load while each query request creates b unit load, then (1) can be converted to (2), in which $\sum q_i$ is the number of query requests in unit time period, and $\sum r_i$ is the number of routing requests in unit time period.

For any given peer p, we also define an overloading threshold value, To, which represents the point after which additional workload placed on the peer will induce overloading, and trigger load redistribution for p. This value can be represented as a portion of the peer's capacity (e.g., $To = 0.8C$, which means that p is considered overloaded when it reaches 80% of its capacity). We also introduce another load threshold value, Ts, that represents the 'safe' workload capacity for a peer. A peer will agree to accept redistributed load from the overloaded peer only when its load is below Ts. The goal of load redistribution is to make the workload on all participating peers fall below their respective Ts in order to guarantee that none of them will again be overloaded soon after the redistribution.

3.2 Adaptive Object Replication Algorithm

Nodes storing very popular objects are susceptible to becoming overwhelmed due to external requests for those objects. In this case, attempting to redistributing the load via shedding objects and keys to other nodes does not guarantee any noticeable improvement, since even one very popular key could overload a node. Therefore, we suggest a replication-based method to relieve the load of overwhelmed nodes. By replicating popular keys of overloaded nodes to lightly loaded nodes, we help to balance the network load. While this idea of balancing by replication is by itself not new, the when, where, and how we propose are. Specifically, when does replication occurs, where do we locate the candidates to help out an encumbered node, and how do the consequences of the redistribution get announced to the rest of the system.

When: Each peer periodically checks its current load via the previously mentioned load metrics. If it is above the overloading threshold (i.e., $L > To$), and this overloading is caused mainly by query loads (i.e., $a \times \sum q_i \geq (b \times \sum r_j)$), it will pick a light loaded node to replicate its keys thus sharing the load. When more than one peer is responsible for a popular key, each responsible peer only manages part of the load, and reducing the chance of overloading.

Where: Upon detecting that it has crossed the 'overload' threshold, a node will issue a replica discovery query to the network, broadcasted (with limited steps) down the DHT broadcast tree (with the querying node as the root). Any lightly loaded nodes (defined previously as nodes with current load $L<Ts$) in the path of the tree will reply with its load information. Once enough responses have been received, the overloaded node begins transferring its keys and objects to these candidates, creating replica nodes of itself.

How: Once replicas are created, dissemination of information about the existence of these new replica must occur. For prefix-based DHTs like Pastry or Tapestry, the replica informaiton is updated at all the peers in the original peer's neighborhood set, leaf set, and routing table. Those nodes in turn update their own state based on the information received. Similar to the node joining process, the total cost for the replica update in terms of the number of messages exchanged is $O(\log_2^b N)$. Similarly, for Chord-based DHTs, the replica info is updated at the fingers and predecessors of the related nodes to reflect the addition of this replica, requiring $O(\log^2 N)$ messages. This process can be carried out asynchronously, since the peers in the routing table already have a pointer to the original peers and asynchronous update will not negatively affect the correctness of the system. When a query needs to be forwarded to a popular key, neighbouring nodes can now pick peers in a round-robin fashion from the list of available peers holding the key. Thus, the queries for the hot key are now partitioned among the multiple peers storing the key.

When a popular key later becomes unpopular, the replica nodes can just get rid of the replicated keys, using access history to gauge the popularity of the replica.

3.3 Adaptive Routing Replication Algorithm

Replicating popular keys relieves the query answering load of the nodes responsible for the keys. However, another major source of workload in DHT overlays is caused by relaying queries among nodes. A node may be overwhelmed simply by the traffic of forwarding incoming routing queries. For example, the last hop neighbours of a popular key can be overloaded by forwarding queries to the popular node. While this problem can be partially solved by the aforementioned duplication of popular keys to disperse the traffic, it cannot completely alleviate the problem since certain nodes in the system might still be functioning effectively as traffic hubs for popular sections of the network. To address this problem, we propose a balancing scheme which actively redistributes the routing load of an overloaded node by duplicating its routing table to other nodes, thereby sharing its routing load. When a node is overloaded by routing loads, it will pick a light loaded node to replicate its routing table, so that the replica node can share its routing load. As with the object replication algorithm, the routing replica information should be propagated to other related nodes. These nodes subsequently update their respective routing tables by adding a replica entry to the entry of the original node so that future queries can be routed to either the original node or the new node, all the while maintaining system network correctness. Besides load balancing, replication approach can also improve the routing resiliency in the face of network failures.

Figure 1 shows an example of the Pastry structure with the replication of routing tables. The query for item ID 0221, which is actually served by node 0222, is initiated at node 2012. According to its routing table, node 2012 chooses 0021 as the next hop. Node 0021 determines that node 0200 should be the right node to forward the query. Since node 0200 has a replica at node 1102, node 0021 may choose 1102 as the next hop. When the query is sent to 1102, it uses the duplicated routing table for 0200 to serve the query and send the query to the destination node 0222. When node 0200 is exposed to a high load, the replicas will share some of the traffic, preventing overload.

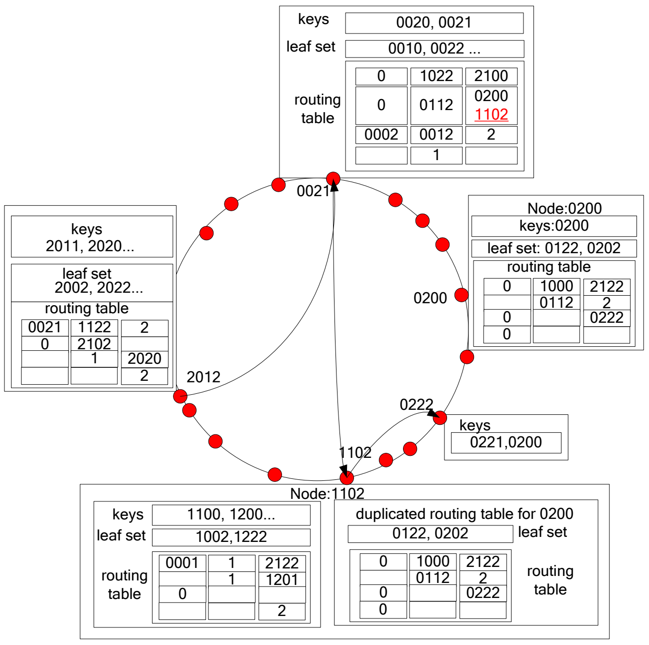


Figure 1. An example of adaptive routing replication algorithm

3.4 Dynamic routing load adjusting algorithm

In addition to the use of replication, another scheme to balance the routing load is by dynamically reconfiguring the routing table. In the previously mentioned methods, an overloaded node actively redistributes its own load, but in cases where external policies or the network environment prevents the redistribution effort, replacing routing table content can help relieve highly loaded nodes.

This algorithm is tailored specifically for DHTs like Pastry or Tapestry. In those systems, many nodes with the same prefix can be potentially filled in a node's routing table; the one in the table is the one the node knows, and with topological metric considered, it will 'pick' the one closest to itself. We propose changing the strategy of choosing nodes in the routing table to balance routing load (especially to relieve heavily loaded nodes). In lieu of simply choosing according to a proximity metric, we choose with lower routing load instead. Whenever an overloaded node receives a querying message from its neighbour, it will reply with a message indicating its overloaded status. This neighbour, receiving the message, will, at the earliest opportunity possible, replace the entry of the overloaded node in its routing table with another node of the similar prefix. The light-loaded candidate nodes are learned from forwarded query messages which include IDs of passed nodes. By doing so, traffic is alleviated from the overloaded load as long as it is not the actual 'end target' of the query request, as the replacement node will be able to direct any queries the original node could've, and forwarding traffic is spread out more evenly.

Continuing from our example in Figure 1, in node 1102's routing table, let us assume that a neighbor node, 2122, (1st row 3rd column) is heavily-loaded. When a query passes through node 2012 to 0021 and then comes to node 1102, since 2012 shares the identical first digital prefix (2) as the overloaded neighbour 2122 in 1102's routing table, the entry of 2122 will be replaced with 2012. This way, the traffic to the more heavily loaded 2122 will be redirected to the more free 2012.

4. EXPERIMENTAL RESULTS

In this section, we examine the experimental effectiveness of our proposed load balancing schemes. We applied each of our schemes to the Pastry system individually and evaluated the difference in performance. Then, we examined their combined effect on the system.

4.1 Setup

Our balancing algorithm is experimented on Pastry. Each peer is assigned a 128-bit identifier, using a sequence of digits with base 2^b. In our simulation, the value of base b is 1. Each node is randomly assigned a value C representing its capacity ($C = 5^i, i \in \{0,1,2,3,4\}$). A node's current load is represented by the number of query forwarding requests and query answering requests it receives per unit time. The load caused by the two kinds of requests has different weight to simulate the different causes they would incur. In our experimentations, we assume that the query load is similar to that of a simple question answering procedure, such that we can set the ratio of the weight of query answering load vs. query routing load to 5 (i.e. $a{:}b = 5{:}1$ in Equation (2)). Given the lightness of the query answering process in the current experiment, this would be a reasonable projection. In the case of more significant operations, such as file transfers, the ratio will be larger by several orders of magnitude.

The simulation is carried out on an overlay network with 1,000 nodes and 20,000 objects (2,000 distinct ones) randomly distributed throughout the nodes. Queries are issued with different frequencies and distributions (random distribution and Zipf distributions with different α value, which represents how skewered the distribution is, with a larger α value indicating greater levels of skewness). For the purpose of our experiments, the *To* (overload) threshold for each node was set at 0.8, and the *Ts* (safety) threshold at 0.6 of its maximum capacity. Each experiment is run ten times with different random seeds, and the results are the average of these ten sets of results.

Four different load balancing strategies were evaluated and analyzed; 1) Simple Pastry: this is the basic Pastry system with no load balancing strategy used (represented by *Non* in the following figures). 2) Reconfiguring the routing table (*RR*). 3). Duplicating objects (*DO*). 4) Duplicating the routing table (*DR*). and 5) Integrating all of the previous three balancing schemes (*All*). The performance metric we used is the *load/capacity* ratio.

4.2 Results

4.2.1 Effect of query distribution

In an open, live P2P environment, query distribution follows a Zipf distribution [17]. Figure 2 shows the effect of query distribution on a node's load burden (without any balancing mechanism used), indicating the mean, 1st and 99th percentiles of the peer workload/capacity ratio. This percentile represents the workload variances on the peers, such that the greater the difference, the less evenly the load is being distributed. In the experiment, we increase the skew degree of the query distribution from random to Zipf with α=1.25. We can see that query distribution has a significant impact on peer load. The more skewed the query distribution, the more unevenly distributed the load becomes, causing some nodes to suffer from a very high load when the query is sufficiently skewed.

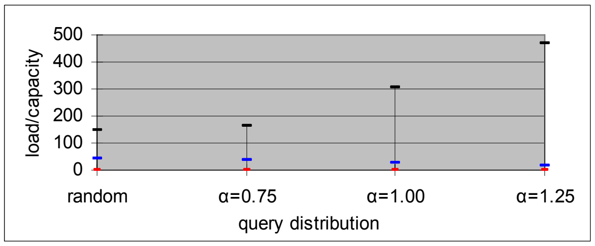


Figure 2. Mean, 1st and 99th percentiles of the ratio of load/capacity under different query distribution

4.2.2 Performance of load balancing schemes under different query distributions

Overloading a node can induce an overflow to its request queue, causing new coming queries to be dropped, which in turn deteriorates the system performance. Figure 3 shows an overview of the fraction of dropped queries under different query distributions and with each of our load balancing schemes. We can clearly see that each of our load balancing algorithms reduce the query drop fraction, thus improving the system performance. Specifically, algorithm *All*, which integrates all of the other algorithms we presented earlier, experiences the best performance in terms of minimizing the query drop rate even under a highly skewed query distribution.

A caveat worth mentioning is that in Figure 3, we can see that duplicating routing table scheme reduces more dropped queries compared to duplicating objects. Note that this is dependent on the parameters we set, particularly the query load to routing load ratio (*a:b=5:1*). If the ratio is larger, it means that the query answering is more complex compared to the query forwarding, thereby accounting for more of the total load. From the figure, we see clear indication of the effectiveness of our proposed algorithms. The following is a more in-depth examination of the results of each of our balancing schemes:

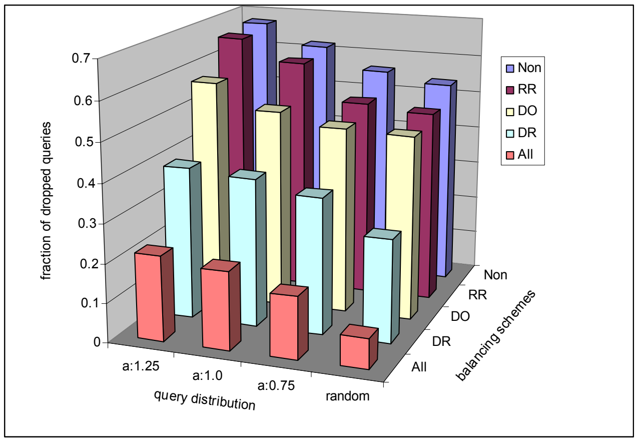


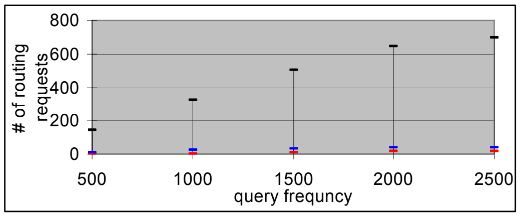
Figure 3. Fraction of dropped queries under different query distribution and load balancing schemes.

4.2.3 *Balancing of routing load*

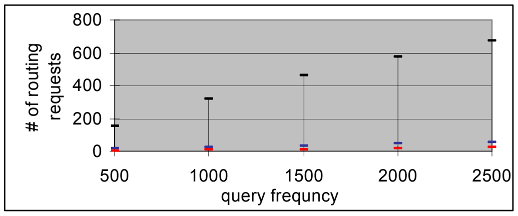
Figure 4 illustrates the performance of each query routing-related balancing algorithm relative to the query insertion rate. The network size is 10^3 and the query distribution is Zipf (α =1). The figures show the percentile of the routing load in terms of query forwarding requests received. As mentioned, the smaller the difference, the better the load balancing performs. We can see that, as we increase the query frequency, the variance for all the algorithms becomes invariantly larger. This is because query distribution is skewed, so increasing the query frequency will result in more unbalanced requests, exacerbating the existing imbalance problem.

While the majority of the experimental results were as we expected, the re-configuring routing table scheme contributed surprisingly little to performance gain. We attribute this observation due to the following: (1) Prefix requirements for the bottom rows of a node's routing table are more stringent, so candidates for the replacement nodes of these rows are more difficult to find, resulting in the algorithm being unable to efficiently adjust this part of the routing (2) Consequently, the last-hops-neighbor of a node cannot find replacements to route to that node, so neighbours (in ID space) of a popular node can not be relieved.

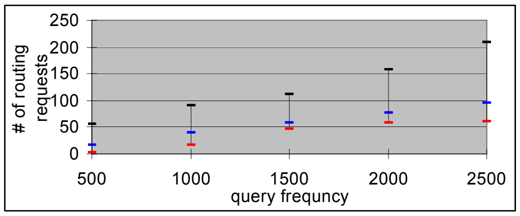
We can also observe from Figure 4 (d) that by integrating all of the schemes together, we were able to achieve performance beyond the sum of the benefits from just reconfiguring the routing table and duplicating the routing table. We surmise that this is due to the fact that although duplicating-objects does not balancing routing loads directly, it redistributes the load of hot spots, helping to relieve the traffic towards the hot spots and thus avoiding overloading the neighborhood with forwarding requests.



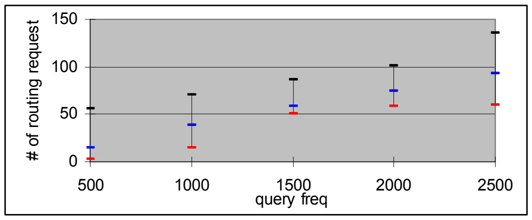
(a) Pastry, without any balancing adjustment



(b) Balancing by dynamic re-configuring routing table



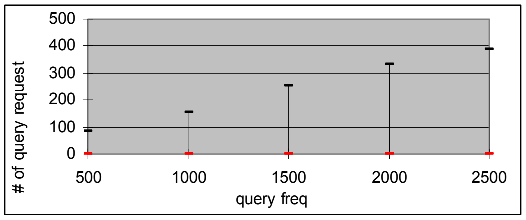
(c) Balancing by duplicating routing table



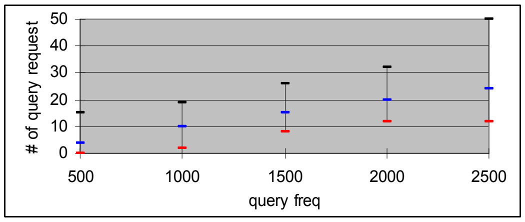
(d) Balancing by all combination

Figure 4. Mean, 1st and 99th percentiles of the routing load (in terms of number of routing request) under different query frequency

4.2.4 *Balancing of query answering load*



(a) Pastry, without any balancing adjustment



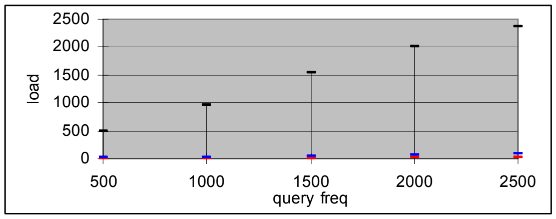
(b) Balancing by duplicating objects

Figure 5. Mean, 1st and 99th percentiles of the query load (in terms of number of query answering request) under different query frequency

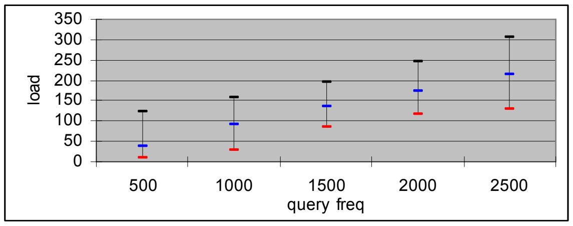
Figure 5 shows the result of the adaptive object replication algorithm. We can see that the algorithm effectively relieves the overloaded nodes and balances the load because the hot items are quickly replicated in other nodes in the network.

4.2.5 *Balancing of the whole system load*

Figure 6 shows the results of the combined algorithm in balancing system load. (Note: the ratio of the weight of query answering load and the weight of query forwarding load is 5:1. With different ratio, the figure may change a little bit.) The results of the experiment clearly indicate significant and drastic effect on system load balances.



(a) Pastry, without any balancing adjustment



(b) Balancing by combine all schemes

Figure 6. Mean, 1st and 99th percentiles of the system load under different query frequency

5. CONCLUSION

We have presented an effective approach to balance load in DHT systems. Our work distinguishes routing load and retrieval load, and deals them separately. By dynamically replicating different potions of the overloaded node based on source of the overloading (replicating either its routing table or its keys), we overcome the restrictive nature of traditional balancing schemes that assumes homogeneity among peers and the type of load they incur. This approach enables the system good load balance even when demand is heavily skewed. Extensive simulation results indicate significant improvements in maintaining a more balanced system, leading to improved scalability and performance.

6. REFERENCES

[1] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," *Technical Report, UCB/CSD-01-1141*, April 2000.

[2] A. Rowstron and P. Druschel. "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Middleware*, November 2001.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H.Balakrishnan. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *ACM SIGCOMM, August 2001*, pp. 149-160.

[4] S. Ratnasamy, P.Francis, M.Handley, R.Karp, and S. Shenker. "A Scalable Content-Addressable Network," *ACM SIGCOMM, August 2001*, pp. 161-172

[5] L. Breslau, P. Cao, L. Fan. G. Phillips, and S. Shenker. Web Caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the Conference of the IEEE Communications Society, (INFOCOM 1999)*, pages 126-134, Mar 1999

[6] K. Sripanidkulchai. "The popularity of Gnutella queries and its implications on scalability". In *O'Reilly's www.openp2p.com*, Feb. 2001.

[7] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," in *Proceedings of the 23rd Conference of the IEEE Communications Society (Infocom 2004)*, 2004.

[8] D. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," in *Proceedings of 16th ACM Symposium on Parallelism in Algorithms and Architectures*, 2004, pp. 36–43.

[9] A. R. Karthik, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in structured P2P systems," in *Proceedings of 2ndInternational Workshop on Peer-to-Peer Systems, 2003*, pp. 68–79.

[10] M. Naor and U. Wieder. "Novel architectures for P2P applications: the continuous-discrete approach." In *Proceedings of the Fifteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2003)*, June 2003.

[11] G. Manku. "Balanced binary trees for ID management and load balance in distributed hash tables." In *Proceedings of Twenty-Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2004),* 2004.

[12] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proceedings of the Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA* 2004)

[13] J. Byers, J. Considine, and M. Mitzenmacher, "Simple load balancing for distributed hash tables," in *Proceedings of 2nd International Workshop on Peer-to-Peer Systems, 2003*, pp. 80–87.

[14] J. Ledlie and M. Seltzer, "Distributed, secure load balancing with skew, heterogeneity, and churn," in *Proceedings of the 24rd Conference of the IEEE Communications Society, (Infocom 2005)*, 2005.

[15] MJ Freedman, D Mazieres, "Sloppy hashing and self-organizing clusters" in *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003, Berkeley, CA, USA.

[16] Plaxton C. G., R. Rajaraman, and A. W. Richa. "Accessing nearby copies of replicated objects in a distributed environment ". *Theory of Computing Systems,* 32:241-280, 1999.

[17] George K. Zipf, “Human Behaviour and the Principle of Least-Effort”, *Addison-Wesley, Cambridge MA*, 1949

[18] Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: “Load balancing in dynamic structured P2P systems”. in *Proceedings of the 23rd Conference of the IEEE Communications Society (Infocom 2004)*, 2004.

[19] Zhiyong Xu, and AXaxmi Bhuyan, "Effective Load Balancing in P2P Systems", in *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid,(CCGrid2006),* 2006.

[20] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive replication in peer-to-peer systems. In *Proceedings of 24th International Conference on Distributed Computing Systems (ICDCS)*, 2004.