# Adaptive Load Balancing for DHT Lookups

Silvia Bianchi, Sabina Serbu, Pascal Felber and Peter Kropf
University of Neuchâtel, CH-2009, Neuchâtel, Switzerland
{silvia.bianchi, sabina.serbu, pascal.felber, peter.kropf}@unine.ch

***Abstract*— In the past few years, several DHT-based abstractions for peer-to-peer systems have been proposed. The main characteristic is to associate nodes (peers) with objects (keys) and to construct distributed routing structures to support efficient location. These approaches partially consider the load problem by balancing storage of objects without, however, considering lookup traffic. In this paper we present an analysis of structured peer-to-peer systems taking into consideration Zipf-like requests distribution. Based on our analysis, we propose a novel approach for load balancing taking into account object popularity. It is based on dynamic routing table reorganization in order to balance the routing load and on caching objects to balance the request load. We can therefore significantly improve the load balancing of traffic in these systems, and consequently their scalability and performance. Results from experimental evaluation demonstrate the effectiveness of our approach.**

***Keywords*—peer-to-peer, popularity, traffic load balancing.**

## I. Introduction

Peer-to-peer (P2P) networks are distributed systems where every node—or peer—acts both as a server providing resources and as a client requesting services. These systems are inherently scalable and self-organized: they are fully decentralized and any peer can decide to join or leave the system at any time.

Numerous peer-to-peer networks have been proposed in the past few years. Roughly speaking, they can be classified as either *structured* or *unstructured*. Unstructured P2P networks (e.g., Gnutella [1], Freenet [2]) have no precise control over object placement and generally use "flooding" search protocols. In contrast, structured P2P networks as also called *distributed hash-tables* (DHTs) (e.g., Chord [3], CAN [4], Pastry [5]), use specialized placement algorithms to assign responsibility for each object to specific nodes, and "directed search" protocols to efficiently locate objects. DHTs mostly differ in the rules they use for associating the objects to nodes, their routing and lookup protocol, and their topology.

Regardless of the nature of the system, a P2P network must scale to large and diverse node populations and provide adequate performance to serve all the requests coming from the end-users. A challenging problem in DHTs is that, due to the lack of flexibility in data placement, uneven request workloads may adversely affect specific nodes responsible for popular objects. Performance may drastically decrease as overloaded nodes become hot-spots in the system.

Several strategies have been proposed to improve load balancing by adjusting the distribution of the objects and the reorganization of the nodes in the system. However, such techniques do not satisfactorily deal with the dynamics of the system, or heavy bias and fluctuations in the popularity distribution. In particular, requests in many P2P systems have been shown to follow a Zipf-like distribution [6], with relatively few highly popular objects being requested most of the times. Consequently, the system shows a heavy lookup traffic load at the nodes responsible for popular objects, as well as at the intermediary nodes on the lookup paths to those nodes.

This paper presents a study of the load in structured peer-to-peer systems under biased request workloads. As expected, simulation results demonstrate that, with a random uniform placement of the objects and a Zipf selection of the requested objects, the *request load* on the nodes also follows a Zipf law. More interestingly, the *routing load* resulting from the forwarded messages along multi-hop lookup paths exhibits similar powerlaw characteristics, but with an intensity that decreases with the hop distance from the destination node. One important point that must be noted here is that the process of downloading files is out of band and therefore not considered in this study.

Based on our analysis, we propose a novel approach for balancing the system load, by taking into account object popularity for routing. We dynamically reorganize the "long range neighbors" in the routing tables to reduce the routing load of the nodes that have a high request load, so as to compensate for the bias in object popularity. In addition, we propose to complement this approach by caching the most popular objects along the lookup routes in order to reduce the request load in the nodes that hold those objects.

Our solution has the following characteristics:

- *minimum impact on the overlay*: neither changes to the topology of the system, nor to the association rules (placement) of the objects to the nodes are necessary;
- *low overhead*: no extra messages are added to the system, except for caching. If a node has free storage space, it can dedicate a part of it for caching, which will lead to better load balancing in the system. Other nodes can simply ignore the caching requests;
- *high scalability*: the decision to reorganize routing tables or cache objects are local;
- *high adaptivity*: the routing table reorganization and caching adapt to the popularity dynamics of the objects in the system.

The paper is organized as follows. In Section II we introduce the characteristics of the structured peer-to-peer system taken into consideration in this work. Then we present simulations showing that a Zipf distribution of requests results in an

uneven request and routing load in the system. In Sections III and IV we present, respectively, our approach for popularity-based load balancing and its evaluation. We discuss related work in Section V, and Section VI concludes the paper.

## II. Preliminaries

### A. System Model

In the past few years, several structured P2P systems have been proposed that basically differ in the hash space (ring, Euclidean space, hypercube), the rules for associating the objects to the nodes, and the routing algorithm.

In our work, we assume a classical DHT overlay composed by $N$ physical nodes and $K$ objects mapped on to a ring with a maximum capacity of $2^m$, which corresponds to the maximum number of nodes and objects.

Each node and object has an *m*-bit identifier, obtained by respectively hashing the IP address and the name. For *consistent hashing*, we used the SHA-1 cryptographic hash function, such that, with high probability, the distribution of the assigned identifiers on the ring is uniform, i.e., all nodes receive roughly the same number of keys.

Lookup is based on prefix routing (also similar to Pastry), with at most $O(log_{2^b} N)$ messages necessary to route a request. The identifiers use a sequence of digits with base $2^b$ and each node has a *routing table* and a *leaf set*. The requests are forwarded to the node in these tables that shares the longest common prefix with the requested object.

The routing table is built using the following rule: each entry must contain a node whose ID has a common prefix of a given length (depending on the row number of the entry) with the current node ID, and a different value for the following digit. Note that there are typically more than one node satisfying the rule for an entry. In Pastry, the selection of the node for each entry is based on a proximity metric. In our system we propose to reorganize the routing tables by selecting the nodes with the lowest load. Our solution can be applied to any DHT with neighbor selection flexibility [7].

For the purpose of this study, we assume that the system has the following characteristics:

- *stability*: as churn is not expected to affect the load balancing significantly, no node joins nor leaves the system. As a consequence, we do neither consider a retry mechanism upon lookup failure, nor a bootstrap mechanism to join the system. We briefly discuss the implications of churn in Section IV;
- *homogeneity*: same characteristics for all nodes (CPU, memory, storage size), same bandwidth for all links, and same size for all objects;
- *no locality*: no topology aware routing in the system.

### B. Implications of Zipf-like requests

Similarly to Web requests [8], the popularity of the objects in many DHTs follows a Zipf-like distribution [6]. This means that the relative probability of a request for the $ith$ most popular object is proportional to $1/i^{\alpha}$, where $\alpha$ is a parameter

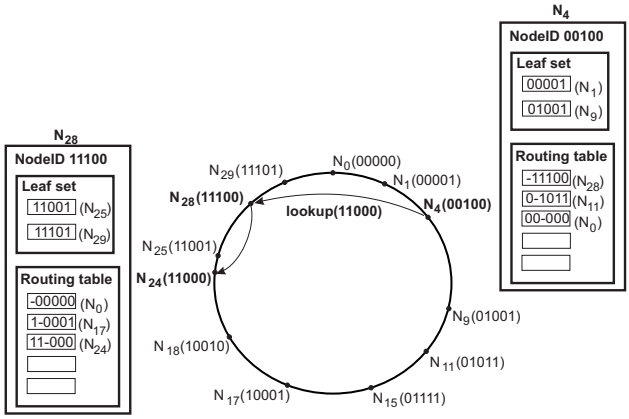


Fig. 1. Overlay and routing structure.

of the distribution, resulting in hot-spots for the nodes that hold the most popular objects.

In case of file sharing applications, many studies have observed that the request distribution has two distinct parts. Very popular files are equally popular, resulting in a linear distribution and less popular files follow a Zipf-like distribution. This usually happens because of the immutability of the objects in file sharing where the clients will request the object only once and then download it (out of band) [9], [10], [11].

In both cases, the amount of traffic received and forwarded by some nodes is much higher than for other nodes. In this context, the paper analyzes the worst case (Zipf-like distribution) and focuses on improving the degraded performance caused by hot-spots.

Each node $n_i$ has a capacity for serving requests $c_i$, which corresponds to the maximum amount of load that it can support. In our study, we consider the load as the number of received and forwarded requests per unit of time. Some nodes hold more popular objects than others (i.e., have a higher number of received requests), thus being overloaded, with a load $\ell_i \gg c_i$. Other nodes hold less or no popular objects thus presenting a small load compared to their capacity $c_i \gg \ell_i$. With a random uniform placement of the objects and a Zipf-like selection of the requested objects, the *request load* on the nodes also follows a Zipf law. Consequently, we expect that the *routing load* resulting from message forwarding along intermediary nodes to the popular objects also exhibits powerlaw characteristics.

To better understand this problem, we have performed simulations to gather request load information associated to the nodes in the system. At each node, we keep track of the number of requests received for a local object, as well as the number of requests forwarded for a destination located $i$ hops away from the current node. Figure 2 illustrates part of the results of a simulation for an overlay network with 1,000 nodes, 20,000 objects randomly and uniformly distributed, and 100,000 requests following a Zipf-like distribution.

The value of an *i-hop away* entry represents the number of requests that it forwards for nodes at a distance of $i$ hops. Node

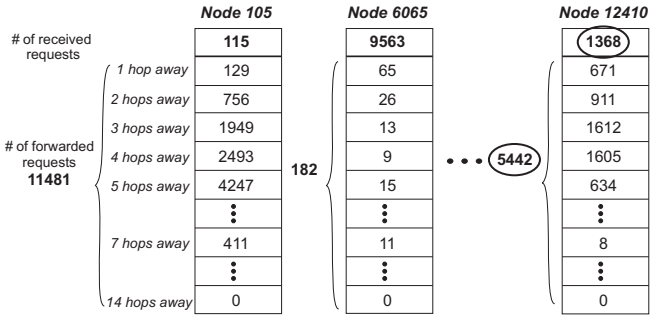


Fig. 2. Statistics of received and forwarded requests.

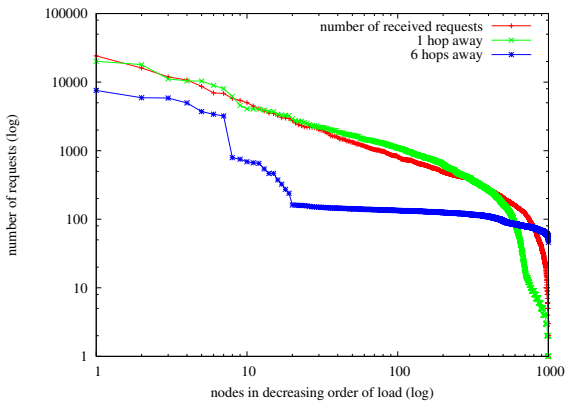


Fig. 3. Request and routing load.

105 receives only few requests, but it forwards many requests. Conversely, node 6,065 holds a popular object. It thus receives many requests, but forwards only few requests since it is not on a path to a popular object. Node 12,410 presents both a high request load and a high routing load. Obviously, nodes 6,065 and 12,410 become hot-spots.

Figure 3 compares the number of requests received by each node, as well as the number of requests it must forward for a destination node that is 1 hop and 6 hops away. All three sources of load follow a Zipf-like distribution, but with an intensity that decreases with the distance from the destination.

In the next section we present our load balancing solution that aims to equilibrate the routing and request load of the nodes in the system.

## III. Adaptive Load Balancing

### A. Routing Tables Reorganization (RTR)

The key principle of our approach is to dynamically reorganize the "long range neighbors" in the routing tables in order to reduce the routing load of the nodes that have a high request load, so as to compensate for the bias in object popularity.

As previously mentioned, each entry of a routing table can be occupied by any one of a set of nodes that share a common prefix. In our approach, we reorganize the routing table by choosing the nodes with the lowest (request and forwarding) load in order to offload the most heavily-loaded nodes. The overloaded nodes (as a consequence of a popular object, or too many forwarded requests or both) are removed from the other nodes' routing tables in order to reduce their load. Instead, the entry will contain another node, from the same region (same prefix), which is less loaded. This way, the nodes that have a high request load will have a small forwarding load, and the nodes with low request load will share the forwarding load.

Figure 4 shows an example of routing table update with no load balancing and Figure 5 illustrates the update based on our load balancing mechanism. In the figures, "++" indicates a high load and "−−" a low one. In the example, node $N_4$ holds a popular object resulting in a high request load. Since it is a heavily-loaded node, it will be removed from the other nodes routing tables. Node $N_{24}$ will update its first entry with node $N_9$, which is less loaded than node $N_4$. Consequently, the load of node $N_4$ will decrease, and the load of node $N_9$ will increase, thus equilibrating the load in the system.

The routing table updates are performed dynamically while routing the requests, without increasing the number of messages. The algorithm is shown in pseudo-code in Algorithm 1, for a node $n_i$ that forwards a request.

Every node keeps track of the approximate load $\ell_k$ of each other node $n_k$ in its routing table. Before forwarding (or sending) a request message, each node adds its load to the message (line 17). The $i^{th}$ node in the request path receives the load information of $i$ other nodes in the request message. A node $n_i$ that receives the request, besides handling it, uses the load information in the message to possibly update its routing table. Each node $n_j$ in the message can match exactly one entry in the routing table of node $n_i$. If the load is lower for node $n_j$ than for node $n_k$ found in the routing table the entry is updated with $n_j$ (lines 5-8).

**Algorithm 1** Pseudo-code for the RTR algorithm at node $n_i$

```
0:  {Receive request}
1:  for each (n_j, ℓ_j) in the message do
2:     entry ← matching entry for n_j in the routing table
3:     n_k ← current node at entry
4:     if n_j ≠ n_k then
5:        if ℓ_j <= ℓ_k then
6:           Replace n_k by n_j at entry
7:           Store ℓ_j at entry
8:        end if
9:     else
10:       Store ℓ_j at entry
11:    end if
12: end for
13:
14: if n_i not owner of requested object then
15:    n_k ← next node to forward request
16:    ℓ_k ← ℓ_k + e
17:    {Add (n_i, ℓ_i) to the request message to be forwarded}
18: end if
```

The load information corresponding to the entries in the routing table of node $n_i$ is not accurate, since the node cannot know at each moment the real values for the load of each

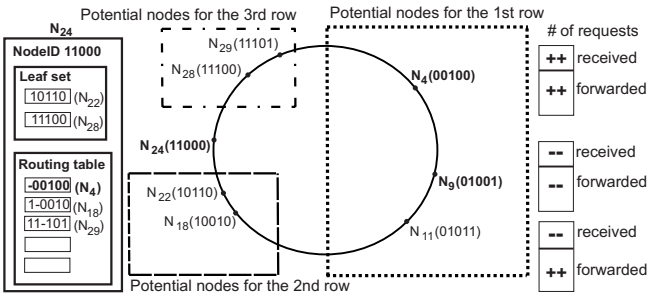


Fig. 4. Routing tables before reorganization for load balancing.

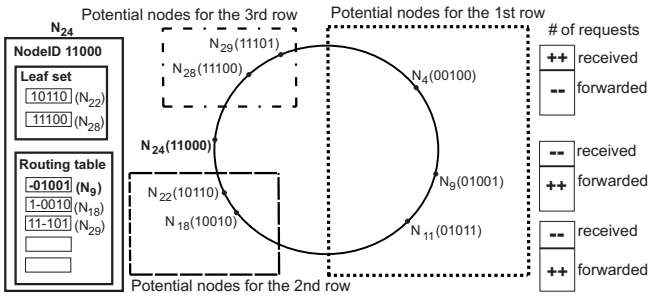


Fig. 5. Routing tables after reorganization for load balancing.

entry. In order to compensate for this limitation, we use several techniques:

- even if the loads for the two nodes $n_j$ and $n_k$ are equal, the entry is updated, since load $l_j$ is $n_j$'s real load but $l_k$ is only an estimation of $n_k$'s load (line 5-8);
- if $n_i$ receives the load information of a node that is already in its routing table (node $n_j$ is the same as node $n_k$), its load is updated (line 10);
- when node $n_i$ forwards (sends) a request to a node $n_k$, $n_i$ updates the load information for $n_k$ in the routing table using an estimation $e$ of the load of $n_k$ (line 16).

In our experiments, we use an estimation $e$ of 1, since we know exactly that the load of $n_k$ will increment by at least 1 from the request that $n_i$ forwards.

This algorithm does neither add extra messages, nor significant complexity to the system. It is based on local load estimations, as well as the information received with each request.

When a new node $n_i$ joins the system and populates its routing table, it takes as initial value for $l_i$ the average of the loads of its new neighbors. Node departures do not require special handling.

As extensions of the algorithm, we might consider a combined metric: proximity (as proposed in Pastry [5]) and load information. This could be a trade-off between proximity-aware routing and load-aware routing. In addition, we might consider adding load information also to the response message of the lookup for an object. Of course, the more information about the system is available, the better lookup traffic can be balanced.

### B. Caching

The routing table reorganization permits us to balance the forwarding traffic of the nodes in the overlay, but the traffic resulting from the received requests still leads to a bottleneck at the destination node. In this section we propose caching as a complementary feature to the routing table reorganization, in order to minimize the number of received requests at the nodes holding popular objects. As a consequence, the request traffic for each cached object will be shared among the node owning the object and the nodes holding the replicas.

Basically, there are two ways to initiate caching: by the client that requests the object and by the server that holds the object [12]. Client-initiated caching is not adequate for applications such as file sharing because a client usually requests an object only once. Therefore, in our approach, the server replicates the object to be cached on some other node(s) in an attempt to reduce its request load. When a request arrives at a node that holds a replica of the requested object in its cache, that node can directly respond to the request.

We refer to two kinds of objects that a node holds:

- *owned object*: an object that belongs to the node according to the initial mapping of objects to nodes;
- *cached object*: a replica of an object owned by another node.

**Algorithm 2** Pseudo-code for the caching algorithm, after node $n_i$ receives a request from $n_j$

```
1:  increment req_recv_counter
2:  increment req_recv[requested_object][n_j]
3:  {Reaching threshold: }
4:  if req_recv_counter = T then
5:     {Compute weights: }
6:     for all objects o on node n_i do
7:        if o is the last cached object then
8:           w[o] ← req_recv[o]/T
9:        else
10:          w[o] ← w[o] * β + (req_recv[o]/T) * (1 − β)
11:       end if
12:    end for
13:    if n_i is loaded then
14:       m_p_o ← o, where w[o] is max
15:       n_c ← n, where req_recv[m_p_o][n] is max
16:       {Will issue a caching request: }
17:       send a request to n_c to cache m_p_o
18:    end if
19:    {Reset counters: }
20:    for all objects o on node n_i do
21:       req_recv[o] ← 0
22:    end for
23:    req_recv_counter ← 0
24: end if
```

The algorithm is shown in pseudo-code in Algorithm 2. We make use of two types of counters for the received

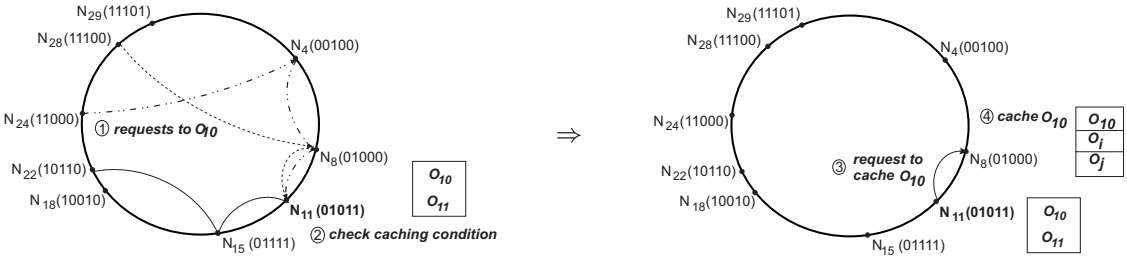


Fig. 6. Example of the caching method.

requests at each node: a per-object counter (for the number of received requests for each object held by the node) and a per-node counter (for the total number of received requests). The counters are incremented at each received request (lines 1-2). A threshold is defined for the per-node counter. Each time the threshold is reached (line 4), a weight is computed for each object held by the node based on the per-object counters (lines 5-12); then, the counters are reset (lines 19-23). The most popular object on the node is the object with the highest weight (line 14).

To compute the popularity of an object, we use a weighted moving average, where the weight is computed as a combination of its previous value and the value computed over the last period (line 10). We use a $\beta$ value of 0.9, such that both terms count in the computation of the object's weight, but the old value (which is a stable information) counts more than the new one. After a caching request has been issued for an object, only the new value is considered (line 8).

For the caching mechanism, the following considerations must be taken into account: storage size of the cache and its policies, when to cache an object and where to store it. In the following we detail all these aspects.

*1) The cache and its policies:* Each node has a cache (storage capacity) for $C$ replicas. Whenever a caching request is received and the storage capacity is exhausted, the replica entry with the lowest weight is discarded and the new replica is stored.

*2) When to cache an object:* A caching request is issued each time the threshold $T$ is reached in case the node is loaded (lines 13-18). Obviously, if the node is not loaded, no caching request is issued, at least until the next threshold.

To know whether a node is loaded or not, we perform two checks:

- *if the node is globally loaded*. We use the load information of the nodes in the routing table; this is not an up-to-date information, yet a rather good estimation of the load of some nodes in the system. A node is globally loaded if its load is bigger than the average load of these nodes;
- *if the node has a lot of received requests*. A node would have a balanced load if the number of received requests is equal to the number of forwarded requests divided by the average path length. Therefore, we compute the average path length of the requests that the node received between two consecutive thresholds. To justify a caching request, a node must satisfy the following condition:

$$recv\_requests > fwd\_requests/path\_avg,$$

where the counters for the number of received and forwarded requests are reset after each threshold.

If both conditions are satisfied, a node will issue a caching request.

*3) Where to store the replica:* Since every message contains information about the request path, the most suitable method is to cache along that path. This can be done (1) on all the nodes in the request path, (2) close to the destination node, (3) at the node that requested the object, or (4) randomly. We choose to do the caching at the last node in the request path. This has the advantage that the object is cached in the neighborhood of its owner where the possibility for a request to hit a replica is much bigger than elsewhere in the system.

Since the requests for a given object may come from any node in the system, the last hop will not always be the same. The node chosen for caching the object is the one that most frequently served as last hop for this object in the lookup paths (line 15).

Figure 6 presents an example of the caching mechanism. Many nodes send requests for the object $O_{10}$ (step 1). After $N_{11}$ receives the requests, it checks the caching condition (step 2) and, if true, computes its most popular object, $O_{10}$, and the node where to store a replica, $N_8$. Then, it issues a caching request for object $O_{10}$ to node $N_8$ (step 3). Finally, $N_8$ caches the object $O_{10}$ (step 4).

## IV. Experimental Evaluation

In this section we evaluate our approach by the means of simulations. First, we present results of the experiments when we apply only routing table reorganizations. Next, we analyze the effect of caching. Finally, some statistics with different Zipf distributions are presented.

### A. Routing Table Reorganization - RTR

The simulated system is an overlay network with 1,000 nodes and 20,000 objects randomly and uniformly distributed, and 500,000 requests following a Zipf distribution with parameter $\alpha = 1$. The routing mechanism is based on Pastry with a leaf set size of 4 entries. The identifiers are a sequence of $m = 16$ bits in base $2^b = 2$.

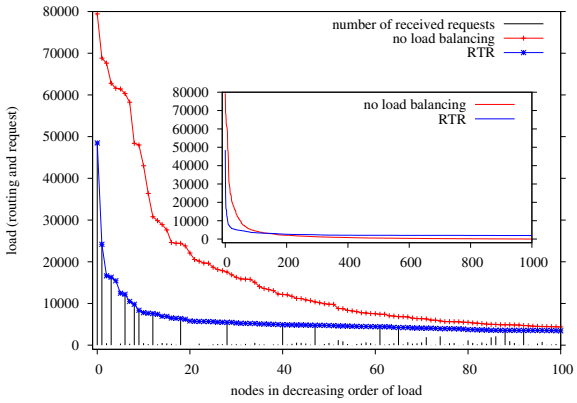


Fig. 7. Load balancing using dynamic routing table updates (*RTR* run2).

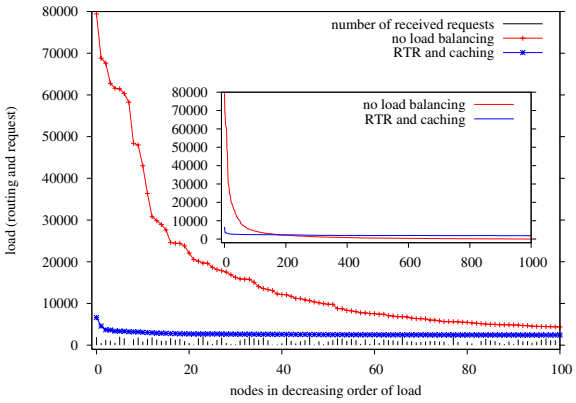


Fig. 8. Load balancing using dynamic routing table updates and caching (*RTR&C* run2).

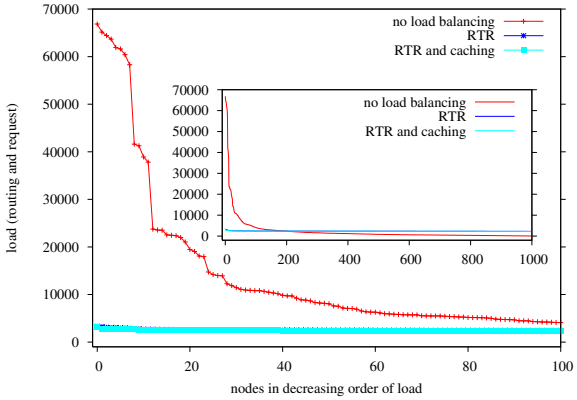


Fig. 9. *RTR* and *RTR&C* using Zipf's $\alpha = 0.5$ (run2).

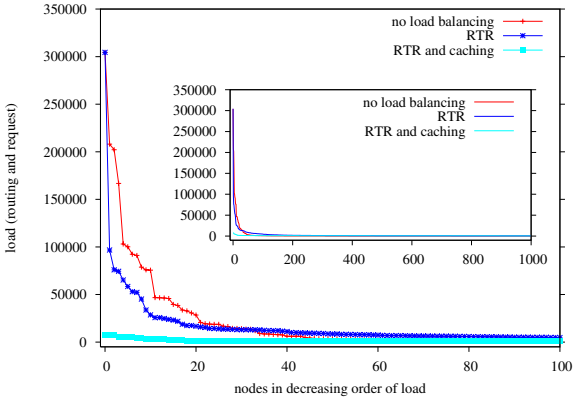


Fig. 10. *RTR* and *RTR&C* using Zipf's $\alpha = 2$ (run2).

To analyze the load balancing algorithms, we use the same experimental setup while applying different routing strategies:

- **run0**: as a base for comparison, the experiment is run with no load balancing;
- **run1**: dynamic run, where the routing tables are dynamically updated while handling the requests;
- **run2**: same as run1, with the difference that the experiment starts with the routing tables obtained after run1.

The results using routing table reorganizations are shown in Figure 7 after run2. The graph shows the load for the first 100 most loaded nodes, while the inner graph presents a global view of the load of all nodes in the system.

The left hand side of the graphs shows that the most loaded nodes are the ones with the highest number of received requests (see the vertical lines). Their load cannot be minimized by the routing table reorganizations, because the number of received requests cannot be decreased (our solution applies to the routing load only, not to the request load). The inner graph confirms that most of the nodes have roughly the same load, thus reaching a good level of load balancing.

In the next subsection we present the results for the routing table reorganization strategy complemented by caching, in order to reduce the load observed at the left hand side of the graph.

### *B. Routing Table Reorganization and Caching - RTR&C*

In our experiments, we used a cache with storage size $C = 3$ and a threshold of $T = 500$ requests. The results using both solutions (routing table reorganization and caching) are shown in Figure 8, after run2. The experiments were done in the same conditions as before, which allows us to distinguish the benefits of using caching as a complementary solution. Comparing Figure 7 with Figure 8, we note the improvement in load balancing for the most loaded nodes (left-hand sides of the graphs), where the load dramatically decreases; the load for the nodes at the right-hand side of the graph slightly increases, which demonstrates that nodes share the load more evenly.

A potential source of overhead resides in the additional messages sent for caching. For the results presented in Figure 8, there are 243 extra messages. This is negligible if we take into consideration the number of requests issued (500,000).

The two principal variables in the system are the size of the cache $C$ and the value of the threshold $T$. Table I presents some statistics of the results obtained while running the experiment with different values for the cache size and the

| Exp | Cache | Tshd | Msg | Avg | Var |
|---|---|---|---|---|---|
| *run0* | - | - | - | 2,403 | 7,243 |
| *run1* | | | | | |
| *RTR* | - | - | - | 2,505 | 2,336 |
| *RTR & C* | 1 | 500 | 253 | 2,248 | 1,285 |
| *RTR & C* | 3 | 500 | 274 | 2,214 | 1,231 |
| *RTR & C* | 1 | 1,000 | 123 | 2,322 | 1,373 |
| *RTR & C* | 3 | 1,000 | 123 | 2,308 | 1,312 |
| *C* | 3 | 500 | 639 | 1,870 | 6,093 |
| *run2* | | | | | |
| *RTR* | - | - | - | 2,563 | 2,056 |
| *RTR & C* | 1 | 500 | 261 | 2,099 | 369 |
| *RTR & C* | 3 | 500 | 243 | 2,059 | 304 |
| *RTR & C* | 1 | 1,000 | 151 | 2,196 | 465 |
| *RTR & C* | 3 | 1,000 | 134 | 2,167 | 380 |
| *C* | 3 | 500 | 904 | 1,657 | 5,871 |

TABLE I
STATISTICS

threshold in the same system. The statistics are for the three experiments: run0, run1 and run2. Besides the load average and variance, we also show the number of messages necessary for the caching requests.

We observe that the variance of the system load decreases from 7,243 to 2,056 when using the *RTR* strategy (Figure 7). The load average slightly increases because changing the routing tables in the destination nodes' closest area might increase in some cases the path length; it remains, however, in $O(log_{2^b} N)$. The variance decreases even more to 304 when using the *RTR&C* solution (Figure 8). The load average also decreases, the path becoming shorter in this case.

A smaller value of the threshold means a higher frequency of caching requests, and consequently more messages. However, there is no notable improvement.

The cache does not need a large storage capacity to be effective. We obtained the same results when using $C = 3$ and $C = 100$. There is a small improvement when using $C = 3$ over $C = 1$ because the most popular objects can remain permanently in the cache.

For comparison purposes, we also ran an experiment using just caching (*C*), with no routing table update strategy. The results show that there is no significant improvement (the variance is still high, 5,871 after run2). This means that caching is no satisfactory solution when used alone.

In these experiments we used a Zipf distribution with parameter $\alpha = 1$ for the request workload. The results using other values for $\alpha$ are shown in the next subsection.

### C. Zipf-like requests with different parameter

The solution proposed in this paper for load balancing is independent of the $\alpha$ parameter of the Zipf distribution. Based on [11] and [8], we performed some experiments varying the value $\alpha$. The caching storage size is set to $C = 3$ and the threshold to $T = 500$ requests.

Table II presents the statistics for $\alpha = 0.5$ and $\alpha = 2$. The results are also plotted in Figures 9 and 10, respectively. For $\alpha = 0.5$, the problem is found in the routing load, which is almost perfectly solved by our routing load balancing solution. The complementary solution (caching) is not necessary here. As shown in the graph, the results using the *RTR* solution and the results using the *RTR & C* solution tend to overlap. In the case of $\alpha = 2$, the number of received requests for the most popular objects is very high compared to the other objects; the problem is thus only partially solved by the routing load balancing strategy and caching becomes necessary.

| | $\alpha = 0.5$ | | | $\alpha = 2.0$ | | |
|---|---|---|---|---|---|---|
| **Exp** | **Msg** | **Avg** | **Var** | **Msg** | **Avg** | **Var** |
| *run0* | - | 2,392 | 6,639 | - | 2,321 | 16,568 |
| *run1* | | | | | | |
| *RTR* | - | 2,339 | 708 | - | 2,625 | 13,185 |
| *RTR & C* | 110 | 2,336 | 684 | 546 | 990 | 2,215 |
| *run2* | | | | | | |
| *RTR* | - | 2,336 | 96 | - | 2,683 | 11,661 |
| *RTR & C* | 252 | 2,318 | 74 | 328 | 719 | 574 |

TABLE II
STATISTICS USING $\alpha = 0.5$ AND $\alpha = 2.0$ ZIPF PARAMETER

## V. RELATED WORK

### A. Node and Object Reassignment

*1) Single ID per Node: Karger et al.* [13] propose to balance the address space by activating only one of $O(\log N)$ virtual servers at a given time and to balance objects by relocating the nodes to arbitrary addresses. Since the reassignment of IDs is done by join and leave operations, this solution increases the churn of the system. *Byers et al.* [14] propose to use the *power of two choices* where each object is hashed onto $d \geq 2$ different IDs and placed in the least loaded node. The algorithm provides a partial traffic balancing by having each node request at random one of the $d$ possible nodes, which all maintain redirection pointers. Since these nodes do not hold the objects, additional messages are issued. The *k-Choices* load balancing algorithm proposed in [15] presents a similar approach. All these approaches focus on objects and nodes distribution imbalance. Therefore, even if they can avoid more than one popular object to be stored in the same node, the request distribution at the destination remains skewed as a consequence of the popularity of some objects in the overlay.

*2) Multiple ID per Node: Karthik et al.* [16] propose three different schemes for load balancing based on the transfer of virtual servers from heavily loaded to lightly loaded nodes. *Godfrey et al.* [17] complement this idea by also taking into consideration the dynamism and the heterogeneity of the system. The proposed algorithm combines elements of the many-to-many scheme and one-to-many scheme proposed in [16]. This solution suffers from additional network traffic and temporary storage for virtual servers to be reloaded. *Zhu et al.* [18], [19] improve communication costs at the expense of a more complex structure ($k - ary$ tree) that must be maintained. Again, these solutions do not take into account

object popularity. However, the virtual servers can be balanced such that a node does not hold more than one popular object.

### B. Caching and Replication

Solutions addressing the uneven popularity of the objects are based on replication and caching. *Lv et al.* [12] propose three replication strategies. With the "owner replication" the requesting node keeps a copy. The number of replicas increases thus proportionally to the number of requests to the object. In [20] and [21], a threshold is used to minimize the number of replicas. These strategies work well only when a node requests the same object many times. The second strategy, "random replication", creates copies on randomly chosen nodes along the lookup path. Simulation results [12] have demonstrated that the third strategy, "path replication", which replicates objects on all nodes along the lookup path, performs best. As an example, [22] proposes DHash replication of $k$ successors and caching on the lookup path. *Yamamoto et al.* [23] propose the path adaptive replication method, which determines the probability of replication according to a predetermined replication ratio and storage capacity. Our solution is similar to this approach except that we do not determine the probability of the replication. In case the node does not have enough capacity to store the replica, it ignores the caching request. *Ramasubramanian et al.'s* [24] strategy replicates objects based on the Zipf parameter $\alpha$ in order to minimize the resource consumption. This solution requires to exchange several messages to decide the replication level for each object. *Swart* [25] proposes to use a second hash function to obtain a subset of $r+1$ virtual servers and place the object on the $r$ nodes with the lowest load. Since DHTs exhibit path convergence [7], this solution is less adapted to the popularity problem than path replication.

## VI. Conclusions

In this paper, we proposed a novel approach to balancing the traffic load in peer-to-peer systems. For the routing and request load balancing, we proposed, respectively, routing table reorganizations and adaptive caching based on the popularity of the objects. Our solution requires neither changes to the topology, nor to the association rules (placement) of the objects to the nodes. Only caching requires some extra messages to be exchanged. Results from experimental evaluation demonstrate a more balanced traffic and, consequently, improved scalability and performance.

## Acknowledgements

This work is supported in part by the Swiss National Foundation Grant 102819.

## References

[1] "Gnutella." [Online]. Available: http://www.gnutella.com/

[2] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Proceedings of ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000. [Online]. Available: http://freenet.sourceforge.net

[3] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM SIGCOMM*, 2001, pp. 149–160.

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proceedings of ACM SIGCOMM*, 2001, pp. 161–172.

[5] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science*, vol. 2218, pp. 329–350, 2001.

[6] A. Gupta, P. Dinda, and F. E. Bustamante, "Distributed popularity indices," in *Proceedings of ACM SIGCOMM*, 2005.

[7] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The impact of DHT routing geometry on resilience and proximity," in *Proceedings of ACM SIGCOMM*, 2003, pp. 381–394.

[8] L. Breslau, P. Cao, G. P. L. Fan, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proceedings of IEEE Infocom*, 1999, pp. 126–134.

[9] K. Sripanidkulchai, "The popularity of gnutella queries and. its implications on scalability," Carnegie Mellon University," White Paper, 2001.

[10] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan, "Measurement, modeling, and analysis of a peer-to-peer file-sharing workload," in *Proceedings of 19th ACM SOSPS*, 2003, pp. 314–329.

[11] A. Klemm, C. Lindemann, M. K. Vernon, and O. P. Waldhorst, "Characterizing the query behavior in peer-to-peer file sharing systems," in *Proceedings of ACM Internet Measurement Conference*, 2004.

[12] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proceedings of 16th International Conference on Supercomputing*, 2002, pp. 84–95.

[13] 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.

[14] J. Byers, J. Considine, and M. Mitzenmacher, "Simple load balancing for distributed hash tables," in *Proceedings of 2nd IPTPS*, 2003.

[15] J. Ledlie and M. Seltzer, "Distributed, secure load balancing with skew, heterogeneity, and churn," in *Proceedings of IEEE Infocom*, 2005.

[16] A. R. Karthik, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in structured P2P systems," in *Proceedings of 2nd IPTPS*, 2003, pp. 68–79.

[17] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," in *Proceedings of IEEE Infocom*, 2004.

[18] Y. Zhu and Y. Hu, "Towards efficient load balancing in structured P2P systems," in *Proceedings of 18th International Parallel and Distributed Processing Symposium*, 2004.

[19] Y. Zhu and Y. Hu., "Efficient, proximity-aware load balancing for DHT-based P2P systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 4, pp. 349–361, 2005.

[20] M. Naor and U. Wieder, "Novel architectures for p2p applications: the continuous-discrete approach," in *Proceedings of 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, 2003, pp. 50–59.

[21] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher, "Adaptive replication in peer-to-peer systems," in *Proceedings of 24th ICDCS*, 2004, pp. 360–369.

[22] F. Dabek, M. Kaashoek, D. Karger, R.M., and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of 18th ACM SOSP*, 2001, pp. 202–215.

[23] H. Yamamoto, D. Maruta, and Y. Oie, "Replication methods for load balancing on distributed storages in p2p networks," in *Proceedings of Symposium on Applications and the Internet*, 2005, pp. 264–271.

[24] V. Ramasubramanian and E. G. Sirer, "Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays," in *Proceedings of Networked System Design and Implementation*, 2004.

[25] G. Swart, "Spreading the load using consistent hashing: A preliminary report," in *Proceedings of Models and Tools for Parallel Computing on Heterogeneous Networks*, 2004, pp. 169–176.