Dynamic Load Sharing in Peer-to-Peer Systems: When some Peers are more Equal than Others

Sabina Serbu, Silvia Bianchi, Peter Kropf and Pascal Felber
Computer Science Department, University of Neuchâtel
CH-2009, Neuchâtel, Switzerland
{sabina.serbu, silvia.bianchi, peter.kropf, pascal.felber}@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 keys (objects) and to construct distributed routing structures to support an efficient location. These approaches address the load problem, and load balancing is achieved by moving the keys. However, the problem is still not properly covered. 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 relying on object popularity. Our approach is based on routing table reorganization in order to balance the lookup traffic load. We have implemented this approach in a Pastry-like system. The obtained results demonstrate a better balance of load, which can lead to improved scalability and performance.

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-organizing: they are fully decentralized and any peer, regardless its power, can 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, Freenet) have no precise control over object placement and generally use “flooding” search protocols. In contrast, structured P2P networks (e.g., Chord [1], CAN [2], Pastry [3]), also called distributed hash-tables (DHTs), use specialized placement algorithms to assign responsibility for each object to specific peers, as well as “directed search” protocols to efficiently locate objects. DHTs mostly differ in the rules they use for associating the objects to peers, their routing and lookup protocol, and their topology.

Regardless the nature of the system, a P2P network must scale to large peer 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 and replication, uneven request workloads may adversely affect specific peers responsible for popular objects. Performance may drastically decrease as heavily-loaded peers become hot-spots in the system.

Several strategies have been proposed to improve load balancing by adjusting the distribution of objects among all the participating peers in the system. Such techniques do not, however, satisfactorily deal with the dynamics of the system, or heavy bias and fluctuations in the popularity distribution. In particular, requests in a structured P2P system have been shown to follow a Zipf-like distribution [4], with few highly popular objects being requested most of the times. Consequently, the system shows a heavy lookup traffic load at the peers responsible for popular objects, as well as at the intermediary nodes on the lookup paths to those peers.

This paper presents a study of the load in structured peer-to-peer systems under Zipf-based request workloads. Simulation results demonstrate that, with a random uniform placement of the objects and a powerlaw (Zipf) selection of the requested objects, the request load on the peers 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 peer. One important point that must be considered is that the process of downloading files is out of bound for this study.

Based on our analysis, we propose a novel approach for balancing the system load, by taking into account object popularity for routing. More precisely, we dynamically reorganize the “long range neighbors” in the routing tables to reduce the routing load of the peers that have a high request load, so as to compensate for the bias in object popularity. Our mechanisms require changes neither to the topology, nor to the association rules (placement) of the objects to the peers. They can, however, be combined with load balancing approaches that use these techniques. Simulations show that the resulting network has a more balanced routing traffic.

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, followed by the motivation of the object popularity load in such systems. Then we present simulations showing that a Zipf-like 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. MOTIVATION

In this section, we present the DHT model used in the simulation and some general ideas for introducing our popularity-
based load balancing solution for structured peer-to-peer systems.

A. System Design

In the past few years, several structured P2P systems have been proposed, such as Chord [1], Tapestry [5], CAN [2]. Basically, these DHT approaches differ in the hash space (ring, Euclidean space, hypercube), rules for associating the keys to the peers and the routing algorithm.

In our work, we assume a DHT overlay composed by \( N \) physical nodes and \( K \) objects mapped on a ring. Each node and object has an \( m \)-bit identifier, such that the maximum capacity of the ring (maximum number of nodes and objects) is \( 2^m \).

A node and object identifier is the result of hashing, respectively, the IP address and the name. For consistent hashing, we assigned the identifiers using the SHA-1 cryptographic hash function, so that, with high probability, the distribution is uniform (all nodes receive roughly the same number of keys).

The resulting identifier of a node indicates the region of the identifier space that the node is responsible for. In our model, the assigned region for a node is its neighbourhood: half of the area between its predecessor and itself, and half of the area between itself and its successor.

The routing procedure is based on prefix routing similar to Pastry [3], with at most \( O \log_b N \) messages exchanged between nodes to resolve a request, where the identifiers use a sequence of digits with base \( b \). Each node has a routing table and a leaf set. The routing table is composed of \( \log_b N \) rows with \( 2^b - 1 \) entries each. The \( i \)th entry in the table maps to a node with a common prefix of length \( i \) in the ring. A request is forwarded to the node in the routing table whose ID has the longest common prefix. The procedure is based on finding the proper entry in the routing table and forwarding the request to the node at this entry. Figure 1 presents an example with \( b=1 \) and an identifier space between 0 and \( 2^5 \).

In the routing table, each entry contains a node whose ID has a common prefix of a given length with the current node ID. There may be several nodes suitable for an entry. In Pastry, the selection of the node for each entry is based on a proximity metric. In our work, we propose to reorganize the routing table by selecting the nodes with the lowest load.

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 not consider a retry mechanism to search for a key nor a bootstrap mechanism to join the system;
- **homogeneity**: same characteristics for all nodes (CPU, memory, storage size), same bandwidth for all links, and same size for all keys;
- **no locality**: no topology aware routing in the system.

B. Hot-spots

The load in a P2P system denotes several aspects that degrade the performance of the system. This includes the following:

- distribution of the objects in the system, which may be uneven;
- number of requests for a particular object in a period of time, i.e., its popularity;
- traffic: amount of bytes received and forwarded.

The load caused by an imbalance of the objects distribution in the system has been largely discussed. This imbalance can reach an \( O(\log N) \) factor depending on the consistent hashing, i.e., the fraction of space owned by a peer is exponentially distributed. Many solutions have been proposed based on the concept of virtual servers, first introduced by [6]. This concept consists of a node having multiple IDs in the ring for a better load balancing. Because we focus in our work on object popularity and the associated routing load of the requests, we assume a uniform distribution of the objects and the keys by using a good hash function.

Similar to Web requests [7], the popularity of the objects in DHTs follows a Zipf’s-like distribution [4]. This means that the relative probability of a request for the \( i \)th most popular object is proportional to \( 1/i^\alpha \), where \( \alpha \) is a parameter of the distribution. The request distribution follows a Zipf-like distribution with varying \( \alpha \), resulting in hot-spots for a set of 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 once the object and download it [8], [9], [10].

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, having a Zipf-like distribution, and focuses on improving the degraded performance caused by hot-spots.
that aims to equilibrate the request load and routing load of the system. This graph points the Zipf-like request load and the bias attenuates with the distance of the node to the destination.

Moreover, with a random uniform placement of the objects the system. In our study, we consider the load as the number of received and forwarded requests per unit of time. Some nodes are more popular than others (i.e., have a higher number of received requests), thus being overloads, with a load \( \ell_i > c_i \). Other nodes are less popular, or not popular at all, presenting a small load compared to their capacity \( c_i > \ell_i \). Moreover, with a random uniform placement of the objects and a Zipf-like selection of the requested objects, the request load on the peers also follows a Zipf law. Consequently, the routing load resulting from message forwarding along intermediary nodes exhibits similar powerlaw characteristics, but with an intensity that decreases with the hop distance from the destination peers.

To better understand this problem, we have performed some simulations to see the request load associated to each node in the system. At each node, we keep track of the number of requests received by a node, as well as the number of requests forwarded for a destination node located \( n \) hops away (see Figure 3). A small set of nodes have a much higher number of received requests than others and many nodes have few received requests. Moreover, the node 1-hop away to the destination presents almost twice as more number of forwarded requests in the second row than the third row of a node 2-hops away. The Figure 2 shows the total number of received requests in first row, the forwarded requests in the second row (1-hop away) and the total number of forwarded requests in the 7th row (6-hops away) of all the nodes in the system. This graph points the Zipf-like request load and routing load, where the load and the bias attenuates with the distance of the node to the destination.

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

### III. Load Balancing Solution

Figure 3 shows part of the results of a simulation of an average sized overlay network with 1,000 nodes and 20,000 objects randomly and uniformly distributed, and 100,000 requests following a Zipf-like distribution. As shown, node 105 receives only few requests, but it forwards many requests. Conversely, node 6065 holds a popular object, thus receiving many requests, but it doesn’t forward a lot of requests since it is not on a path to a popular key. Node 12410 presents both a high request load and a high routing load. Thus, nodes 6065 and 12410 become hot-spots.

The mostly used technique to deal with hot-spots is caching and replication [11], [12]. For unstructured peer-to-peer systems, several studies [13], [14] have analyzed the proportion of replication of the objects and their popularity to achieve optimal load balancing. Since DHTs lack on flexibility in data placement, i.e., the objects have a specific position in the overlay network, these techniques have a limited applicability for this kind of system.

We propose an approach based on reorganizing the routing table in order to minimize the number of forwarded requests for the nodes that already have a high number of received requests. Our approach is based on dynamic reorganization of the “long range neighbors” in the routing tables to reduce the routing load of the peers that have a high request load, so as to compensate for the bias in object popularity.

As presented before, Pastry defines for each entry in the routing table a region of potential nodes and it selects the node topologically closest to itself. In our approach, we reorganize the routing table choosing the nodes with the lowest (request and forwarding) load in order to minimize the load for the most loaded nodes. The nodes that have a high load (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 minimize 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.

Figures 4 and 5 show an example of updating the routing table based on this mechanism. In the example, node N4 holds a popular object resulting in a high request load. Its load is too high, so it will be removed from the other nodes routing tables. Node N24 will update its first entry with node N9.
which is less loaded then node N4. Consequently, the load of node N4 will decrease, and the load of node N9 will increase, thus equilibrating the load in the system.

The routing tables updates are performed dynamically, while running the requests, without increasing the number of messages. In our algorithm, each node keeps track of the load \(l_k\) of each node \(n_k\) in its routing table. Before forwarding (or sending) a request message, each node will add itself and its load to the message. The \(i^{th}\) node in the request path will receive in the request message the load information of \(i\) nodes. A node \(n_i\) that receives the request, besides treating it, will use the load information in the message in order to update its routing table with better entries in terms of load. Each node \(n_j\) in the message can match exactly one entry in the routing table of node \(n_i\). If the load is better (smaller) for node \(n_j\) than for node \(n_k\) found in the routing table, the entry will be updated with \(n_j\) and its load. 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 entry. To try to compensate this, we use several techniques: (1) if node \(n_j\) is the same as node \(n_k\), its load will be updated; (2) when a node forwards (sends) a request, it will increment the load of the entry that it used (for estimation we use an increment of 1, knowing that the real load will be incremented with at least 1 from this request); (3) even if the loads for the two nodes \(n_j\) and \(n_k\) are equal, the entry will be updated, since load \(l_j\) is \(n_j\)'s real load but \(l_k\) is only an estimation of \(n_k\)'s load. The routing table update algorithm for a node that forwards a request is shown in pseudo-code in Algorithm 1.

**Algorithm 1** Pseudo-code for the load algorithm at node \(n_i\)

1. (Receive request)
2. for each \((n_j, l_j)\) in the message do
3. \(n_k \leftarrow \) current node at entry
4. if \(n_j \neq n_k\) then
5. \(l_j \leftarrow l_k + 1\)
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. if \(n_i\) not owner of requested key then
14. \(n_k \leftarrow \) next node to forward request
15. \(l_k \leftarrow l_k + 1\)
16. \(\{\text{Add } (n_j, l_j)\} \text{ to the request message to be forwarded}\)
17. end if

First we do a dynamic run, where the routing tables are dynamically updated while running the requests, and then we evaluate the results. After that, we use the obtained routing tables to perform a second dynamic run, to see the improvements when we start with optimized routing tables. Finally, we do a static run with no routing table updates at all. The purpose of this last run is to show that in a system with no load balancing strategy, the results are better when starting with optimized routing tables.

We primarily focus on measuring and analysing the request and routing load in the system, subsequently simply called load.

The simulated system has \(10^3\) nodes and \(2 \times 10^4\) keys, and we issue \(5 \times 10^5\) requests from random sources. The nodes and the key IDs are computed on 15 bits. The system is based on an implementation of Pastry, with base \(b=1\) and a leaf set of \(|L|=4\) nodes.

To analyze the results of our solution, we use the same experimental setup while applying different routing strategies.
popularity, after the second dynamic run of the experiment. As shown in the graph, the highest load is decreased by half. Moreover, the load in Figure 6 tends to 0, while in Figure 7 it remains almost constant (approximately from node 300), showing that most of the nodes have the same load.

Figure 8 shows the rate of updates to the routing tables in the second dynamic run: the rate of updates is high in the beginning, but quickly stabilizes at a small value.

Our algorithm for dynamically updating the routing tables of the nodes in the system shifts the load from the most loaded nodes to less loaded nodes, by having the less loaded nodes forward most of the traffic instead. This way, the highly loaded nodes will get rid of the traffic that they had to forward, and become less loaded. The solution does not deal with distributing the keys. This problem has already been well studied and can be addressed by using virtual servers [15].

Our techniques cannot decrease the load below the number of requests addressed to a node. Thus, we still have a Zipf-like distribution, but with much lower intensity.

The statistical analysis showed that the variance of the system load is decreasing from 7,161 for the results shown in Figure 6, to 2,167 for the results shown in Figure 7. This confirms that the load extremes are getting closer. The load average is slightly increasing from 2,353 to 2,585, because changing the routing tables in the destination node’s closest area might increase in some cases the path length. However, the path length is still in the order of $O(\log_2 N)$, where $N$ is the number of the different nodes in the system.

In order to better perceive the load distribution for the most loaded nodes, Figures 9 and 10 show the same data as Figures 6 and 7 for the first 300 nodes. They also show the number of received requests per node.

The nodes at the left hand side of the graphs are the most loaded ones. Comparing the two graphs, Figure 9 exhibits more nodes with a high load mostly induced by the forwarded requests. In Figure 10, fewer nodes have a high load, which mainly results from the received requests. The most loaded nodes are now the nodes with the highest number of received requests; the next most loaded nodes are their direct neighbours. The less loaded nodes at the right hand side of the graph (see Figure 9) are now more loaded, which results in a more balanced overall load tending towards a constant (see Figure 10).

Until now, we considered a leaf set of 4 nodes. With a higher size of the leaf set, the results are even better, as the routing load is shared by more nodes in the vicinity of a popular node. These results after two dynamic runs are shown in Figure 11.

Table I contains some statistics (load average and variance)
Many load balancing schemes have been proposed for DHT-based systems. They can be classified in three categories: virtual servers, flexible choice, caching and replication. **Virtual Servers.** Most of the schemes focus on the imbalance of the key distribution due to the hash function [16], [17], [15]. They argue that a naive hash function can result in a $O(\log N)$ imbalance in the number of keys associated to nodes. Moreover, some applications associate semantics with object IDs, such as dictionary applications, causing a bigger imbalance factor since the IDs are no longer uniformly distributed. To solve this problem, these approaches make use of virtual servers, first proposed by [6]. The concept is based on simulating at each node a logarithmic number of virtual servers organized as a ring. However, the virtual servers do not completely solve the problem of load balancing, because the method fails to adapt to dynamic request patterns; in that case, frequent reorganization of virtual servers may be required.

The solutions proposed by [15], [18] complement this idea taking into consideration also the dynamism and the heterogeneity of the system. In this context, the imbalance can significantly increase as the heterogeneity and the dynamism (joins and departures) increase.

In [19], [20] the goal is not only to ensure fair load distribution over nodes proportional to their capacity, but also to minimize the load balancing cost by transferring virtual servers between heavily loaded and lightly loaded nodes. These approaches do not consider the load of a specific resource (such as CPU, storage size or bandwidth) since they focus on the distribution of virtual servers. The load caused by popularity is not treated by these load balancing solutions, being considered as a orthogonal problem.

**Flexible choice.** In [21], the focus is also on the imbalance of the key distribution that may be caused by a naive consistent hashing, but they do not use the concept of virtual servers since the number of messages necessary for maintenance and failure detection is high. As an alternative to virtual servers they have proposed to use the power of two choices where two or more hash functions are applied and examined in order to use the node with the lowest storage load.

A better distribution is proposed in [22], by the kind each joining node can choose its position in the hash space by learning the positions of a few existing nodes.

In the same context, in [23], a $k$-Choices load balancing algorithm for DHTs is proposed. The node generates a set of verifiable IDs and at join time it chooses 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. In addition, each node can change its position in the identifier space by choosing another ID.

These solutions can be used to minimize the load traffic caused by popularity by choosing a peer less loaded in the overlay network, but at the expense of more complex overlay maintenance protocols.

Caching and replication. In [24], a lightweight adaptive replication protocol is proposed to solve the problem of hot-spots in DHT systems. The protocol was applied in Chord for replication of objects. However, when an object is replicated, the finger list must also be replicated among the other nodes in the system. This protocol solves the problem of hot-spots at the cost of having a longer routing table to be managed and more objects in the system.

As our technique only updates routing tables and do not change nodes, in principle, they can be combined with another load balancing techniques. We will study this problem as part of our future work.

VI. CONCLUSIONS

In the past few years, several DHT-based abstractions for peer-to-peer systems have been proposed. The basic principle is to associate nodes (peers) with keys (objects) and to construct distributed routing structures to efficiently locate objects. Basically, the existing DHT approaches differ in their topology (ring, multi-dimensional spaces, tree, etc), the rules for associating the objects to the nodes, the construction of the routing tables and their lookup protocols.

In this paper we presented an analysis of the load distribution in structured peer-to-peer systems taking into consideration the load caused by the popularity of the objects. Based on this analysis, we proposed a novel approach to minimize the load generated by popular requests by reorganizing the routing tables accordingly.

Our mechanisms neither require changes to the topology nor to the association rules (placement) of the objects to the peers. Simulations demonstrate a more balanced routing traffic, which can lead to improved scalability and performance.

ACKNOWLEDGEMENTS

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

REFERENCES


