A Structured Peer-to-peer System with Integrated Index and Storage Load Balancing

Viet-Dung Le[1], Gilbert Babin[2], and Peter Kropf[3]

[1] Department of Computer Science and Operations Research, University of Montreal
C.P. 6128, Succursale Centre-Ville, Montréal, (Québec) Canada H3C 3J7
levietdu@iro.umontreal.ca
[2] Information Technologies, HEC Montréal
3000, ch. de la Côte-Sainte-Catherine, Montréal (Québec) Canada H3T 2A7
Gilbert.Babin@hec.ca
[3] Institute of Computer Science, University of Neuchâtel
Rue Emile Argand 11, CH 2000 Neuchâtel, Switzerland
Peter.Kropf@unine.ch

Abstract. Load balancing emerges as an important problem that affects the performance of structured peer-to-peer systems. This paper presents a peer-to-peer system relying on the partitionning of a de Bruijn graph. The proposed system integrates mechanisms that perform index and storage load balancing. Index load refers to the network traffic incurred by a peer in managing an object index, while storage load refers to the storage space and network traffic required to store objects. The proposed mechanisms allow to effectively distribute both index load and storage load according to the peers' capacities.

1 Introduction

A peer-to-peer (P2P) system comprises multiple parties (called *peers*) that can request and provide services at the same time. This decentralized characteristic furthers spreading of workload among all participating peers and thus contributes to solutions for scalability issues in distributed systems. However, in comparison to a centralized system, managing shared objects becomes difficult because of the lack of a central or hierarchical control. Structured P2P systems, such as [3, 4, 7, 9, 11, 12, 14, 17], introduce efficient mechanisms to store and access these distributed objects. The principle inherent to such systems consists in mapping every object onto a key space or index (e.g., by hashing the object identifier), distributing this key space over the available peers, and maintaining a structured connection among the peers according to the keys each peer holds. The connection structure ensures to guide the search for an object to the peer responsible for the object's key in a small number of hops, often $O(\log n)$ in an n-peer system.

The system performance of a P2P network is critically affected by its overload. Indeed, the storage or processing load of the peers, the communication load and the system management load must be carefully handled to obtain satisfactory system performances which may be regarded as the fastest possible response time to user/application requests. Workload distribution and balancing mechanisms contribute to achieve good system performance. However, they may induce expensive restructuring processes, i.e.,

maintenance costs. Our approach aims to balance workload in P2P systems while keeping maintenance costs low. We are interested in two workload aspects: index load and storage load. In P2P systems, finding an object usually requires routing requests through intermediate peers before arriving at destination. The bandwidth used for this task makes up the index load on each peer. The storage load, on the other hand, denotes the usage of each peer's resources in object accommodation. Many load balancing approaches have been proposed. However, to our knowledge, none takes into account these two aspects of load simultaneously.

The present paper introduces a solution that simultaneously handles both index and storage load balancing by separating the concerns of peer identifiers (addresses), key management, and object storage locations. In particular, the proposed P2P structure is based on partitioning a de Bruijn graph where the node identifier space is identical to the key space. Therefore, we will use key or de Bruijn node exchangeably. Each peer holds a non-empty interval of de Bruijn nodes and maintains connections to other peers that hold neighbouring de Bruijn nodes. Based on this structure, looking for a specific key in the P2P system follows appropriate routing paths in the de Bruijn graph.

The **index load balancing** method takes into account the network capacity of the peers. It aims to minimize the network overload that may occur while routing requests in the system. This goal is different from that of most other methods which permanently adjust the load to a target. Since the decrease of the overload reacts only when an overload exists, our method saves on the costs of rebalancing. The balancing method involves two tasks: (1) locally calculating the index load on every peer and (2) dynamically transferring index load from peer to peer by modifying the key interval managed by each peer. We propose efficient mechanisms to perform these two tasks.

The **storage load balancing** method is based on separating the key and the storage location of objects. It eliminates the restriction of an object's residence to its root, where the root refers to the peer responsible for the key interval which includes the object's key. Instead, the root needs only to keep pointers to the location of its objects. This separation enables and facilitates the index load balancing since the move of a key interval from peer to peer entails moving only the involved object pointers (very small in size) instead of the objects themselves. Thus, moving keys does not affect the storage load. Without restriction to the root, the accommodation of objects chooses the storage location such that the storage load on every peer does not exceed the contributed storage capacity. In addition, we take into account the capacity of the peers in serving object requests and migration. We propose a balancing algorithm that minimizes the peer's overload with regards to its capacity. Like the index load balancing, the consideration of overload in this algorithm minimizes rebalancing costs. The algorithm is based on exchanging appropriate objects among pairs of peers in order to decrease the overload whenever it occurs. Finally, a fair advantage of separating key and storage location is the replication facility. The root of an object can maintain a set of pointers to its replicas (placed on different peers). Thus, the object availability is enhanced without further replication techniques.

The rest of this paper is organized as follows. Section 2 summarizes some recent work on load balancing in structured P2P systems. Sections 3 and 4 respectively de-

scribe the methods of index load and of storage load balancing that can operate simultaneously. The last section provides some discussion.

2 Related work

A straightforward approach to load balancing in a structured P2P system is the equalization of the key occupation among the peers (e.g., [1, 8]). The equalization in [1] stochastically makes peers with short key intervals leave and rejoin the system by splitting peers with long key intervals. The method proposed in [8], on the other hand, balances a virtual binary tree whose leaf nodes represent the participating peers. In practice, load balance depends also on the distribution of objects on the peers, the object size, and the storage, processing, and communication capacity of the peers. Equalizing key occupation does not ensure an even load distribution when taking into account all these different factors making up the load.

The application of the *power of two choices* paradigm [2] applies multiple hash functions to map each item to multiple peers. This allows to insert an item on the least loaded peer. The methods in [5, 10] achieve load balance by exchanging key responsibility among the peers. The above approaches cannot simultaneously balance the index load and the storage load because they associate the storage location and the key. Balancing one workload aspect can break the balance of the other one, and vice versa. *PAST* [13] uses a replica diversion process to balance the storage load. However, the concerns of storage location and file identifier in *PAST* are not separated. It maintains an invariant that limits the storage location of a file to the *leaf* sets (see [12] for definition) of a number k of peers. The maintenance of this invariant introduces considerable overhead in a dynamically changing P2P system, e.g., a system with index load balancing.

Expressways [16], an extension of *CAN* [11], proposes an index load balancing method. It structures the network (of size n) as a hierarchy of $\log n$ levels, each one operating like a basic *CAN*. The balancing method is based on promoting peers with higher bandwidth to higher levels in the hierarchy. However, the reaction of the system to balance the load takes place only after aggregating the loads and the capacities of all peers in the system. Moreover, keeping each peer's and the overall system's load/capacity ratio equal can constantly bring the system to restructure itself even if individual peers would not require rebalancing.

The P2P systems introduced in [6, 9, 15] employ the partition of a de Bruijn graph. Like [1, 8], they aim at equalizing key occupation. As discussed above, this equalization is not sufficient for load balancing in structured P2P systems.

3 Index load balancing

3.1 System structure and routing

The P2P network partitions a binary de Bruijn graph $G(V, A)$ of 2^m nodes. The key space is identical to the de Bruijn node identifier space $V = [0, 2^m - 1]$[1]. Obviously,

[1] $[b, e]$ denotes the interval of integers from b to e (inclusive). If $b \leq e$, $[b, e] = \{x \in \mathbb{Z} \mid b \leq x \leq e\}$, otherwise, $[b, e] = [b, 2^m - 1] \cup [0, e]$.

with a large enough m, the number of peers in a real network does not attain 2^m. Each peer holds and is responsible for a non-empty interval of de Bruijn nodes (also called *key intervals*). Every peer is identified by its network (e.g., IP) address. Given a peer p, we denote:

- $p.a$ – the address of p,
- $p.b$ and $p.e$ – respectively the beginning and ending keys of p's key interval,

Two peers p and q must connect, denoted $connect(p, q)$, if there exists at least one arc between any two de Bruijn nodes that fall within the key intervals of p and q respectively, or if their key intervals are numerically adjacent.

$$connect(p,q) = \begin{cases} \text{true} & \text{if } (\exists x, y \mid (x,y) \in A \wedge x \in [p.b, p.e] \wedge y \in [q.b, q.e]) \\ & \quad \vee (p.e = (q.b - 1) \bmod 2^m) \vee (p.b = (q.e + 1) \bmod 2^m) \\ \text{false} & \text{otherwise} \end{cases}$$

These connections are bidirectional, i.e., if $connect(p, q)$ then $connect(q, p)$. Two connecting peers are called neighbours. Each peer maintains a *neighbour list* consisting of a triple $(q.a, q.b, q.e)$ for each neighbour q in the list. The separation between peer address and key means that the peers can dynamically change their key interval $[b, e]$ without affecting the address a.

Loguinov et al. [6], and Naor and Weider [9] introduced a similar structure based on the de Bruijn graph. Their goal is to balance the partitioned zone sizes through different arrival/departure mechanisms. Our focus, however, is in balancing mechanisms taking into account the storage capacity and communication capacity of peers.

The routing function consists in directing a message to the root of a given key x from anywhere in the system. The message follows appropriate de Bruijn routing paths towards x. For convenience, all expressions on the de Bruijn node identifiers are implicitly modulo 2^m, e.g., $x + y$ means $(x + y) \bmod 2^m$. We also refer to the beginning and ending values of interval I as $I.b$ and $I.e$, respectively.

Definition 1 *The distance between two keys x and y, denoted $distance(x, y)$, is the minimum among the length of the de Bruijn routing paths*[2] *from x to y and from y to x.*

Definition 2 *The distance between a key interval I and a key x, denoted $distance(I, x)$, is equal to $distance(v, x)$ where $v \in I$ and $\nexists v' \in I \mid distance(v', x) < distance(v, x)$.*

In the de Bruijn graph of 2^m nodes, each node x has four arcs respectively to nodes $2x$, $2x + 1$, $\lfloor x/2 \rfloor$, and $\lfloor (x + 2^m)/2 \rfloor$. Let the arcs to $2x$ and $2x + 1$ be the fore-arcs and the arcs to $\lfloor x/2 \rfloor$ and $\lfloor (x + 2^m)/2 \rfloor$ be the back-arcs. We use notation $foredistance(x, y)$ to specify the length of the routing path following only fore-arcs from x to y. Similarly, the notation $backdistance(x, y)$ specifies the length of the routing path following only back-arcs. By Definition 1,

$$distance(x, y) = min(foredistance(x, y), backdistance(x, y)).$$

[2] Note that the de Bruijn routing path between two nodes in an undirected de Bruijn graph is not always the shortest path.

Claim 1 *Given a node x, the set of every node y such that $foredistance(x, y) = i$ (with $i \in [0, m]$), denoted $F_i(x)$, is $[x2^i, x2^i + 2^i - 1]$.*

Proof If $i = 0$, it is clear that $F_0(x) = \{x\}$.

If $i > 0$, suppose that $F_{i-1}(x) = [x2^{i-1}, x2^{i-1} + 2^{i-1} - 1]$ is correct. Following the fore-arcs of all nodes in $F_{i-1}(x)$, we have

$$F_i(x) = \bigcup_{y \in F_{i-1}(x)} F_1(y) = [x2^{i-1}2, (x2^{i-1} + 2^{i-1} - 1)2 + 1] = [x2^i, x2^i + 2^i - 1] \quad \square$$

Claim 2 *Given a node x, the set of every node y such that $backdistance(x, y) = i$ (with $i \in [0, m]$), denoted $B_i(x)$, is $\{y_0, y_1, \cdots, y_{2^i-1}\}$ where $y_j = \lfloor x/2^i \rfloor + j2^{m-i}$.*

Proof If $i = 0$, it is clear that $B_0(x) = \{x\}$.

If $i > 0$, suppose that $B_{i-1}(x) = \{y_0, y_1, \cdots, y_{2^{i-1}-1}\}$ where $y_j = \lfloor x/2^{(i-1)} \rfloor + j2^{m-(i-1)}$ is correct. Following the back-arcs of all y_j, we have

$$B_i(x) = \bigcup_{j \in [0, 2^{i-1}-1]} B_1(y_j)$$

where

$$\begin{aligned} B_1(y_j) &= \{\lfloor y_j/2 \rfloor, \lfloor (y_j + 2^m)/2 \rfloor\} \\ &= \{\lfloor (\lfloor x/2^{(i-1)} \rfloor + j2^{m-(i-1)})/2 \rfloor, \lfloor (\lfloor x/2^{(i-1)} \rfloor + j2^{m-(i-1)} + 2^m)/2 \rfloor\} \\ &= \{\lfloor x/2^i \rfloor + j2^{m-i}, \lfloor x/2^i \rfloor + (j + 2^{i-1})2^{m-i}\} \end{aligned}$$

For all $j \in [0, 2^{i-1} - 1]$, the pair $(j, j + 2^{i-1})$ gives all integers in $[0, 2^i - 1]$. $\square$

Given a key interval I and a key x, the $distance(I, x)$ algorithm calculates $F_i(x)$ and $B_i(x)$ for i from 0 to m. If at an iteration d, $F_d(x)$ or $B_d(x)$ has common keys with I, it returns d. This algorithm is efficient because it iterates testing $F_i(x)$ and $B_i(x)$ for at most $m + 1$ times before finding the distance.

Definition 3 *The de Bruijn neighbourhood set of a key interval I, denoted $dbneighbour(I)$, is the set $([I.b \times 2, (I.e \times 2) + 1] \cup [\lfloor I.b/2 \rfloor, \lfloor I.e/2 \rfloor] \cup [\lfloor (I.b + 2^m)/2 \rfloor, \lfloor (I.e + 2^m)/2 \rfloor]) \setminus I$.*

Routing: the following algorithm routes a message from the current peer p to the peer holding key x.

1. if $x \in [p.b, p.e]$, peer p is the destination. Otherwise, continue with step 2;
2. calculate the set $U = dbneighbour([p.b, p.e])$. Find $t \in U$ such that $distance(t, x) = distance(U, x)$. Select neighbour q such that $t \in [q.b, q.e]$. Then continue routing x from q.

The set U may contain several key intervals. We use here the notation $distance(U, x)$ to refer to the minimal distance from the intervals in U to x. The key t satisfying the equality $distance(t, x) = distance(U, x)$ is easily found: we select the key from the intersection of $F_i(x)$ or $B_i(x)$ and the interval (in U) the nearest to x while calculating the distance. This algorithm ensures to reduce the distance from the current position t to x by at least 1 after each hop. The number of routing hops is therefore bound by m.

3.2 Index load calculation

The index load of a peer is defined as the sum of routing message sizes passing through the peer in a unit of time. The idea of index load balancing is to transfer key intervals between peers to minimize the overload. It requires to calculate the routing traffic on different subsets (which we call zones) of each peer's key interval. For large key intervals, registering the routing traffic through all keys is inefficient or even unrealizable. To make this monitoring efficient, we restrict key interval movements. First, a peer p will only transfer keys to the peers holding $p.b-1$ or $p.e+1$. Second, the size of the interval transferred should range from 1 to $s-1$ where s is the whole key interval's size. We further simplify the monitoring by dividing each peer p's key interval into k levels, where $k = \lfloor \log_2(p.e - p.b + 1) \rfloor$. Levels are further broken down into 3 zones. Figure 1 depicts this division.

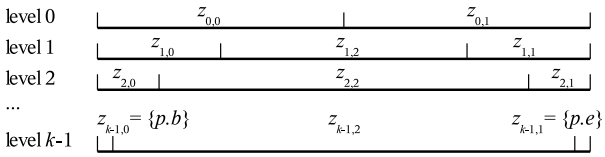

Fig. 1. Zone division at k levels on a peer p

At each level i ($0 \leq i < k$), l_i, the length of zone $z_{i,0}$, is given by:

$$l_i = \begin{cases} \lfloor (p.e - p.b + 1)/2 \rfloor & \text{if } i = 0 \\ \lfloor l_{i-1}/2 \rfloor & \text{if } 0 < i < k \end{cases}$$

Then, we have the zones: $z_{i,0} = [p.b, p.b + l_i - 1]$, $z_{i,1} = [p.e - l_i + 1, p.e]$, and $z_{i,2} = [p.b, p.e] \setminus (z_{i,0} \cup z_{i,1})$. It follows that $z_{k-1,0} = \{p.b\}$ and $z_{k-1,1} = \{p.e\}$. In the special case where $p.b = p.e$, there exists only one level with $z_{0,0} = \{p.b\}$ and $z_{0,1} = z_{0,2} = \emptyset$.

Each peer p constructs a table $G_p[k][3]$. $G_p[i,j]$ registers the routing traffic through zone $z_{i,j}$. This table does not consume much memory space since $k < m$. According to the routing algorithm, when a message λ passes through peer p, λ is oriented via a de Bruijn node $t \in [p.b, p.e]$. For every level i, if $t \in z_{i,j}$ then $G_p[i,j] = G_p[i,j] + |\lambda|$ (where $|\lambda|$ denotes the size of λ). Obviously, the total routing traffic on peer p is $Tr_p = \sum_{j \in [0,2]} G_p[i,j]$, for any i.

Each peer p has a routing traffic capacity C_p. It verifies the index load periodically. We denote the period duration as δt, the beginning time of the current period as t_0, and the current time as t_c. Then, the current index load is $T_p = Tr_p/(t_c - t_0)$. In case $t_c - t_0$ is too small and may produce $Tr_p/(t_c - t_0)$ reflecting an incorrect index load of p, we calculate the load as $T_p = (Tr'_p + Tr_p)/(t_c - t'_0)$ where Tr'_p and t'_0 are, respectively, the routing traffic and the beginning time of the previous period. If $T_p > C_p$, peer p is overloaded. At the end of each verification period, if p is overloaded, it executes the index load balancing algorithm and starts a new period. Any change of $p.b$ or $p.e$ involves also a new period. The beginning of every period resets k and table G_p.

3.3 Index load balancing algorithm

When a peer p discovers that it is overloaded ($T_p > C_p$), it should transfer an appropriate key interval $z_{i,0}$, $z_{i,1}$, $z_{i,0} \cup z_{i,2}$, or $z_{i,1} \cup z_{i,2}$ to the corresponding adjacent neighbour (the peer holding $p.b - 1$ or $p.e + 1$). The transfer must: (1) reduce as much as possible the cumulative overload of the two peers involved, and (2) be as small as possible. These criteria maximize the reduction of the cumulative overload while entailing the least changes. Since peer p only has local information, it does not know which key interval the destination peer can receive. Asking the destination peer for its load information before transferring would slow down the procedure. Furthermore, this may entail an incorrect decision since the status of the destination peer evolves continuously. Our solution allows peer p to propose a set of candidate key intervals to the neighbour. The transfer is completed when the destination peer chooses the most appropriate interval. Such transfer requires only one ask-answer communication between the two peers. Let $w_{h,j}$ (for integers $0 \leq h < 2k$ and $0 \leq j \leq 1$) represent the candidate key intervals to transfer. We determine $w_{h,j}$ using the following rule:

$$w_{h,j} = \begin{cases} z_{k-h-1,j} & \text{if } 0 \leq h < k \\ z_{h-k,j} \cup z_{h-k,2} & \text{if } k \leq h < 2k \end{cases}$$

Thus, the routing traffic load on $w_{h,j}$, denoted $T(w_{h,j})$, is given as:

$$T(w_{h,j}) = \begin{cases} \dfrac{G_p[k-h-1,j]}{t_c - t_0} & \text{if } 0 \leq h < k \\ \dfrac{G_p[h-k,j] + G_p[h-k,2]}{t_c - t_0} & \text{if } k \leq h < 2k \end{cases}$$

Index load balancing algorithm: The index load balancing algorithm (applying the key interval transfer protocol below) on peer p is as follows:

Let $n_0(p)$ denote the adjacent neighbour of p that holds $p.b - 1$ and $n_1(p)$ denote the adjacent neighbour of p that holds $p.e + 1$.

1. select the smallest h such that $\exists j \in \{0, 1\}$ and $T_p - T(w_{h,j}) \leq C_p$. Then, execute the key interval transfer protocol for $w_{h,j}$ from p to $n_j(p)$. If the transfer succeeds, the load balancing stops. Otherwise, continue with step 2;
2. set $l = (j + 1) \bmod 2$. Select the smallest h such that $T_p - T(w_{h,l}) \leq C_p$. Then, execute the key interval transfer protocol for $w_{h,l}$ from p to $n_l(p)$. After this step, the load balancing stops even if the key interval transfer does not succeed.

Key interval transfer protocol: The transfer protocol for the key interval $w_{h,j}$ from peer p to peer $n_j(p)$ tries to move one of the key intervals $w_{0,j}$, $w_{1,j}$,..., $w_{h,j}$ from p to $n_j(p)$ such that the combined overload of p and $n_j(p)$ is minimized. Formally, the overload of p is $O_p = (T_p - C_p + |T_p - C_p|)/2$ and that of $n_j(p)$ is $O_{n_j(p)} = (T_{n_j(p)} - C_{n_j(p)} + |T_{n_j(p)} - C_{n_j(p)}|)/2$. Thus, the transfer must reduce as much as possible $O_p + O_{n_j(p)}$. The key interval transfert protocol involves the following steps:

1. p sends to $n_j(p)$ a key interval transfer proposal including the list ($w_{0,j}$, $w_{1,j}$,..., $w_{h,j}$), the list ($T(w_{0,j})$, $T(w_{1,j})$,..., $T(w_{h,j})$), and O_p;

2. if $n_j(p)$ is not able to receive a key interval or $T_{n_j(p)} \geq C_{n_j(p)}$, it refuses the transfer. Otherwise,
   (a) it searches for the greatest $g \in [0, h]$ such that $T_{n_j(p)} + T(w_{g,j}) \leq C_{n_j(p)}$;
   (b) if no such g exists, $n_j(p)$ searches for the smallest $g \in [0, h]$ satisfying
   $$|T(w_{g,j}) - O_p| + T(w_{g,j}) - O_p + 2(T_{n_j(p)} - C_{n_j(p)}) < 0 \qquad (1)$$
      i. if no such g is found, $n_j(p)$ refuses the transfer because $O_p + O_{n_j(p)}$ cannot decrease;
      ii. if such a g is found, $n_j(p)$ sets the chosen index as g;
3. if an index g is chosen (by step 2a or 2(b)ii), $n_j(p)$ changes its key interval by $[n_j(p).b, n_j(p).e] \cup w_{g,j}$ and establishes connections to the new neighbours. Then, it sends to p an acceptance message specifying the chosen index g;
4. upon receiving the acceptance message with the chosen index g, peer p updates its key interval to $[p.b, p.e] \setminus w_{g,j}$ and releases the unnecessary connections to other peers. The transfer then succeeds;
5. in case $n_j(p)$ refuses the proposal of p, the transfer fails.

Theorem 1 *Given $w_{h,j}$ the interval to be transferred from peer p to peer $n_j(p)$ using the key interval transfer protocol. If $n_j(p)$ chooses an index $g \in [0, h]$, then transferring $w_{g,j}$ will maximize the reduction of the combined overload of p and $n_j(p)$.*

Proof Peer $n_j(p)$ chooses an index $g \in [0, h]$ in step 2a or 2(b)ii of the protocol to accept the transfer of $w_{g,j}$. Recall that at each peer on the routing path, the routing algorithm limits the choice of the next de Bruijn node t (to direct the message to) in the de Bruijn neighbourhood set of the current peer's key interval. Therefore, if $w_{g,j}$ moves from p to $n_j(p)$, $T(w_{g,j})$ is transferred from p to $n_j(p)$ with high probability[3]. The overloads of p and $n_j(p)$ after the transfer are estimated as:

$$O'_p = (T_p - T(w_{g,j}) - C_p + |T_p - T(w_{g,j}) - C_p|)/2$$
$$O'_{n_j(p)} = (T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)} + |T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)}|)/2$$

The condition for reducing the total overload of p and $n_j(p)$ is:

$$\Delta O = O'_p + O'_{n_j(p)} - O_p - O_{n_j(p)} < 0 \qquad (2)$$

If g is set by step 2a, $T_{n_j(p)} + T(w_{g,j}) \leq C_{n_j(p)}$. Thus, $O'_{n_j(p)} = 0$. Since $O'_p < O_p$ and $O_{n_j(p)} = 0$, (2) holds.

If g is set by step 2(b)ii, (1) holds and $O'_{n_j(p)} = T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)}$. It is easy to prove that the left hand side of (1) is equal to $2\Delta O$ and that the smallest chosen index g induces the largest $|\Delta O|$. □

In the key interval transfer protocol, step 2b is mandatory. Study the case where p is overloaded, $n_j(p)$ is underloaded, and there exists no $g \in [0, h]$ such that $T_{n_j(p)} + T(w_{g,j}) \leq C_{n_j(p)}$. Without step 2b, p cannot transfer any key interval to $n_j(p)$. Since $n_j(p)$ is underloaded, it does not intend to take off any part of its key interval. This situation blocks the transfer of load from p. The presence of step 2b allows peer p, in this case, to transfer the least loaded zone $w_{g,j}$ when it reduces the combined overload of p and $n_j(p)$. The load transfer thereby continues until some steady state.

[3] Because of the de Bruijn graph structure, it cannot be guaranteed that all trafic "transferred" will effectively be transferred.

4 Storage load balancing

We define the storage load of a peer as the total of size of the objects it stores. Each peer has a limited capacity available for storage which might be used for object migration. The system's management to store objects requires network bandwidth for object distribution, re-distribution, and associated index management (i.e., routing requests to the network). Consequently, the storage load balancing method has three goals: (1) keeping the storage load under the storage capacity on every peer, (2) adjusting bandwith consumption requirements to bandwidth availability, and (3) minimizing its impact on index load balancing (Sect. 3). To achieve these three goals, we propose to separate the location of the key of an object from the location of the object itself. In this way, objects can reside on arbitrary peers. Therefore, roots are only required to keep pointers to the objects under their responsibility. This approach simplifies the mechanisms required to achieve the first two goals. Finally, the independence of object and key locations enables us to acheive the third goal. Indeed, the key interval transfer remains efficient since only object pointers (very small in size) are required to move when a key interval transfer occurs.

A consequence of this approach is that replication of objects is simplified, hence enhancing object availability, without the need for multiple mapping hash functions (such as e.g. in [2]) or for maintaining invariants that constrain replication to nearby peers (e.g., [13]). A root simply needs to keep pointers to the peers that store the replicas of an object. In this paper, we consider that up to d ($d \geq 1$) replicas of an object may be stored. When a peer departs the network, it must guarantee the objects' availability. By allowing replication, we facilitate this task, since the departing peer only has to wait for objects with unique replicas to be copied elsewhere.

4.1 Object pointer and object insertion

Every peer maintains two tables: $indices$ and $storage$. Each entry of table $indices$ contains the index of an object under the peer's responsibility. The index includes the object identifier (oid), and a list of pointers to the replicas ($replicas$). A replica pointer consists in the replica identifier (rid - a number in $[0, d-1]$), the storing peer address ($location$), and the replica's storage counter ($counter$). This counter is initially set to 0 and incremented after each change of location. Its use will be explained below. The $storage$ table contains the list of objects stored on the peer. For each object, it records the object identifier (oid), the replica identifier (rid), the size ($size$), the address of the root ($root$), and the storage counter ($counter$). In order to maintain $indices$ and $storage$, we propose two protocols, namely the storage notification protocol and the root notification protocol.

Whenever a peer receives an object, it sends to the root of the object a storage notification which contains its address and (oid, rid, $counter$) of the object. The $counter$ field lets the root know whether the notification is newer than the corresponding pointer it holds. If the notification is new, the root updates the pointer. In the notification, the sending peer attaches the $root$ field of the object header, asking whether it keeps the correct root address or not. If the information is incorrect, the root sends back a root notification.

When a peer receives a key interval, it sends root notifications to the storing peer of the objects involved. A root notification contains the root address, (oid, rid, $counter$) of the object, and the storing peer's address ($location$), as known by the root. On receiving the root notification, the storing peer updates the corresponding object's header. If $counter$ or $location$ are incorrect, the storing peer sends a storage notification back.

The maintenance of pointer consistency may seem complicated. However, in comparison to traditional systems which associate storage location and key, the key interval transfer used in our structure requires little effort. It involves the move of a number of pointers and some notifications but does not require any object transfer. The size of an object pointer and of a notification is much smaller than the size of the object.

The object insertion algorithm must ensure that $S_p \leq D_p$ for every peer p, where S_p and D_p are the storage load and capacity of p, respectively. In addition, it tries to store the object on up to d different peers. An insertion request contains the object identifier (oid) and size ($size$). The request is routed to the root of the object. If an index for the object already exists, the insertion algorithm stops. Otherwise, it starts diffusing replicas, with rid from 0 to $d-1$. The diffusion process tries the root first.

A replica diffusion message λ_r contains oid, $size$, and $ridlist$, where $ridlist$ is the list of rids remaining to be assigned. The message traverses multiple peers. A ttl (time-to-live) field limits the number of peers visited. At each peer q, if $S_q + size \leq D_q$ and q does not store any replica of the same object, q extracts a rid from $ridlist$, loads the corresponding replica to the local storage, and sends a storage notification to the root. If $ridlist$ is not empty and $ttl > 0$, q decrements ttl and forwards λ_r to a neighbour not visited. Message λ_r maintains the list of visited peers to perform this verification. If ttl reaches 0 but no replica was stored, the insertion fails. If the number of stored replicas is between 1 and $d-1$, the root starts a new diffusion for the remaining rids.

Object deletion is not considered here since it does not increase the storage load.

4.2 Storage load balancing algorithm

Recall that the first two goals of the storage load balancing are to avoid storage overload and to take into account the bandwidth required respectively available to do so. Implicitely, the storage capacity of a peer D_p corresponds to the real storage available for objects. However, for the system to work properly, another boundary must be defined, which we refer to as the desired capacity on a peer $\overline{D}_p$, with $\overline{D}_p < D_p$. When inserting objects into the system, we ensure that $S_p \leq D_p$, hence allowing S_p to temporarily exceed $\overline{D}_p$ but always limiting it to D_p. Consequently, the storage load balancing problem can be specified as the minimization of the storage overload with regards to $\overline{D}_p$ while keeping $S_p \leq D_p$.

Given $A_p = \overline{D}_p - S_p$ the available space on peer p, a peer is overloaded when the overload $O_p = (-A_p + |A_p|)/2$ is positive, otherwise $O_p = 0$. The storage load balancing algorithm aims at minimizing the overload of all system components. It consists in the decentralized exchange of objects between pairs of peers. Suppose that an overloaded peer p exchanges objects with a peer q. In general, p sends to q a set of objects R_{pq} and q sends back to p a set of objects R_{qp}. Given that S_{pq} and S_{qp} are the storage loads of R_{pq} and R_{qp}, respectively, the combined overload of p and q decreases only if $A_q > 0$ and $0 \leq S_{qp} < S_{pq}$.

Definition 4 *Given that peer q receives a storage load S_{pq} from a peer p and selects a storage load S_{qp} to send back to p, the optimal exchange must (1) reduce the combined overload of p and q the most, and (2) minimize S_{qp}.*

Condition (1) guarantees the fastest reduction of the combined overload, while condition (2) minimizes the data volume sent. Hence, this approach not only reduces the storage overload, but also the bandwidth required to perform storage load balancing.

Theorem 2 *Given two peers p, q, with $A_p < 0$ and $A_q > 0$, and S_{pq}, the optimal exchange occurs when*

$$S_{qp} = \begin{cases} 0 & \text{if } S_{pq} \leq A_q \text{ or } A_q < S_{pq} \leq -A_p \\ \text{closest to } \min(A_p, -A_q) + S_{pq} & \\ \quad \text{such that } 0 \leq S_{qp} < S_{pq} & \text{if } S_{pq} > \max(-A_p, A_q) \\ \quad \text{and } S_{qp} > A_p - A_q + S_{pq} & \end{cases}$$

Because of the limitation of space, we do not present the proof of this theorem. However, it can be found in the full version of this paper.

Storage load balancing algorithm: Each peer p periodically verifies the storage load. If p is overloaded, it starts a balancing session:

1. p diffuses an available space interrogation ϕ, with a limited ttl (time-to-live) field, to its neighbourhood. Each peer q that receives ϕ the first time, processes ϕ, decrements ttl, and forwards ϕ to its neighbours excluding p and the peer from which ϕ comes. q responds to ϕ by sending $A_q = \overline{D}_q - S_q$ to p if $A_q > 0$;
2. for each reply A_q received, if p is still overloaded, p and q exchange objects such that the combined overload of p and q will decrease the most while the object migration is minimized:
   (a) p selects a set of objects R_{pq} to send to q satisfying one of the following conditions: (1) R_{pq} is the smallest that can underload p without overloading q; (2) if (1) cannot be satisfied, R_{pq} is the largest that cannot overload q; and (3) if both (1) and (2) cannot be satisfied, R_{qp} contains only the smallest object;
   (b) q selects a set of objects R_{qp} to send back to p. The selection is based on the optimal exchange condition stated in Theorem 2.

5 Conclusion

We have introduced balancing methods for index load and storage load that can simultaneously operate. The index load balancing is based on the exchange of key intervals among the peers. Unlike the *Expressways* [16] method, which must collect the load information of all peers before redistributing load, our method relies only on local information. We thus avoid the overhead of the load information communication.

The storage load balancing method manipulates the system structure at the object level, instead of the key level (such as the *Virtual servers* [10] method). The manipulation at the key level exhibits less flexibility since the all objects belonging to one key

must move together with the key. Moreover, a move of keys in balancing the storage load also affects index load.

The load balancing methods presented operate on the overload instead of the load itself. Most other methods aim to adjust the load or the load/capacity ratio of every peer with a global objective function. This requires to globally calculate the targeted optimization and to continuously reorganize the system. By relying on the local examination of the overload, we need to react only when the overload exists and when it can be reduced. Experiments to evaluate the proposed load balancing methods are currently being conducted. So far, preliminary results have confirmed their anticipated efficiency. These experimentation results will be presented and discussed elsewhere.

References

1. M. Bienkowski, M. Korzeniowski, and F. M. auf der Heide. Dynamic load balancing in distributed hash tables. In *IPTPS'05*, February 2005.
2. J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash table. In *IPTPS'03*, February 2003.
3. P. Fraigniaud and P. Gauron. Brief announcement: An overview of the content-addressable network d2b. In *ACM PODC'03*, page 151, July 2003.
4. M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *IPTPS'03*, February 2003.
5. D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *ACM SPAA'04*, pages 36–43, June 2004.
6. D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: Routing distance and fault resilience. In *ACM SIGCOMM'03*, August 2003.
7. D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *ACM PODC'02*, pages 183–192, July 2002.
8. G. S. Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *ACM PODC'04*, pages 197–205, July 2004.
9. M. Naor and U. Weider. Novel architectures for p2p application: the continuous-discrete approach. In *ACM SPAA'03*, June 2003.
10. A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *IPTPS'03*, February 2003.
11. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalabale content-addressable network. In *ACM SIGCOMM'01*, pages 161–172, August 2001.
12. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware'01*, November 2001.
13. A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *ACM SOSP'01*, October 2001.
14. I. Stoica, R. Moris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM'01*, pages 149–160, August 2001.
15. X. Wang, Y. Zhang, X. Li, and D. Loguinov. On zone-balancing of peer-to-peer networks: Analysis of random node join. In *ACM SIGMETRICS'04*, June 2004.
16. Z. Zhang, S.-M. Shi, and J. Zhu. Self-balanced p2p expressways: When marxism meets confucian. Technical Report MSR-TR-2002-72, Microsoft Research, 2002.
17. B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerance wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California Berkeley, April 2002.