BALLS: a structured peer-to-peer system with integrated load balancing

Viet DUNG LE*, Gilbert BABIN**, Peter KROPF***

Abstract

Load balancing is an important problem of structured peer-to-peer systems. We consider two aspects of load: index management load that specifies the consumption of network bandwidth for routing traffic, and storage load that denotes the usage of computer resources for object (file) accommodation. This paper presents a structured peer-to-peer overlay that supports simultaneously the index management load balancing and storage load balancing. We used experiments to evaluate the effectiveness of the proposed load balancing methods and to validate their advantages.

Key words : Structured peer-to-peer systems, load balancing.

BALLS : UN SYSTÈME PAIR-À-PAIR STRUCTURÉ AVEC L'EQUILIBRAGE DE CHARGE INTÉGRÉ

Résumé

L'équilibrage de charge est un problème important des systèmes pair-à-pair structurés. Nous considérons deux aspects de charge : la charge de gestion d'index qui spécifie la consommation de la bande passante du réseau pour le trafic de routage et la charge de stockage qui dénote l'utilisation des ressources d'ordinateur pour le stockage des objets (par exemple, des fichiers). Cet article présente un système pair-à-pair structuré qui supporte l'équilibrage intégré de la charge de gestion d'index et de la charge de stockage. Nous utilisons des expériences pour évaluer l'efficacité des mécanismes d'équilibrage proposés et pour valider leurs avantages.

Mots clés : Systèmes pair-à-pair structurés, équilibrage de charge.

Contents

* University of Montreal, Canada.
** *HEC* Montréal, Canada.
*** University of Neuchâtel, Switzerland.

The original publication is available at www.springerlink.com ANN. TÉLÉCOMMUN., 61, n° 11-12, 2006

Abstract

Load balancing is an important problem for structured peer-to-peer systems. We are particularly interested in the consumption of network bandwidth for routing traffic and in the usage of computer resources for object storage. In this paper, we investigate the possibility to simultaneously balance these two types of load. We present a structured peer-to-peer overlay that efficiently performs such simultaneous load balancing. The overlay is constructed by partitioning the nodes of a de Bruijn graph and by allocating the partitions to the peers. Peers balance network bandwidth consumption by repartitioning the nodes. Balancing of computer resources for storage is enabled by dissociating the actual storage location of an object from the location of its search key. The paper presents and analyzes the protocols required to maintain the overlay structure and perform load balancing. We demonstrate their efficiency by simulation. We also compare our proposed overlay network with other approaches.

Keywords—structured peer-to-peer systems, load balancing

Résumé

L'équilibrage de charge est un problème majeur des systèmes pair-à-pair structurés. Nous nous intéressons plus particulièrement à l'utilisation de la bande passante pour le routage de messages et à l'utilisation des ressources informatiques pour le stockage d'objets. Dans cet article, nous présentons un réseau pair-à-pair structuré permettant l'équilibrage efficace et simultané de ces deux types de

charge. Le réseau est construit en partitionnant les nœuds d'un graphe de De Bruijn et en assignant les partitions aux pairs. Les pairs équilibrent l'utilisation de la bande passante en repartitionnant les nœuds. On équilibre l'utilisation des ressources informatiques pour le stockage en dissociant l'emplacement d'un objet de celui de sa clé de recherche. L'article présente et analyse les protocoles requis pour maintenir la structure de réseau et pour équilibrer la charge. Nous démontrons leur efficacité par simulation. Nous comparons également notre réseau à d'autres approches.

Mots clés—systèmes pair-à-pair structurés, équilibrage de charge

7.1 Introduction

A peer-to-peer (P2P) system consists of multiple parties (peers) similar in functionality that can both request and provide services without a centralized control. This feature allows the system to spread the workload over the participants, hence aggregating their resources to more efficiently provide services or execute computational tasks. Roughly speaking, P2P networks can be classified as either structured or unstructured. Unstructured P2P networks (e.g., the Gnutella [gnu01] and KaZaA [kaz05] file sharing systems) have no particular control over the file placement and generally use "flooding" search protocols. In contrast, structured P2P networks (e.g., Chord [SMK+01], CAN [RFH+01], Pastry [RD01a], P-Grid [ACMD+03], D2B [FG03], Koorde [KK03], Viceroy [MNR02], DH DHT [NW03a], Tapestry [ZHS+04]) use specialized placement algorithms to assign responsibility for each file to specific peers. A structured P2P network distributes the responsibility for a key space over the available peers and maintains the peers' connection structure based on the set of keys each peer holds. Objects are mapped to the key space (e.g., by hashing the object id) and then assigned to

the peers (say roots) responsible for the corresponding keys. Such a system provides efficient routing and object location in which the request, directed by the structured connection, arrives at the destination within a small number of hops, often $O(\log n)$ in an n-peer system. However, the peer dynamicity (or churn, i.e., peer arrivals and departures) introduces expensive restructuring costs. These costs are called maintenance costs.

7.1.1 Resolving the load problem

In structured P2P systems, the performance, such as the response time to user requests or the capacity to store objects is critically affected by workload distribution. Overload in communication, system management, storage, or processing reduces performance. In the following, we consider two aspects of workload : index management load and storage load. P2P routing usually traverses intermediate peers between the source and the destination. This routing process uses the bandwidth of the peers along the path. Index management load refers to the bandwidth consumption for this task. The P2P system spreads objects over the peers. The resource usage for object accommodation on each peer makes up its storage load. To our knowledge, no existing system balances these two workload aspects simultaneously.

This article describes a P2P structure (namely BALLS – Balanced Load Supported P2P Structure) with the ability to simultaneously balance the index management load and the storage load while keeping the maintenance costs low. A brief description of this system was published in [LBK06c]. BALLS partitions a de Bruijn graph among the peers participating in the P2P network. By partitioning a de Bruijn graph, we harness the advantages of this type of graph, that is, low peer degree and efficient routing.

The **index management load balancing** method we propose takes into account the heterogeneity of peers, where each peer has its own capacity (called index management capacity) to accommodate for this load. The goal of index management load balancing is to minimize

the possible overload of the whole system. Unlike other load balancing methods that permanently restructure the system to direct the load on every peer to a target load or a global load/capacity ratio, our method adjusts the system only when overload occurs. It therefore saves on the cost of restructuring.

BALLS separates the peer id (peer address) from the keys it holds. This allows the peers to dynamically modify their key responsibility. Since the P2P routing is based on the de Bruijn routing paths, arrangement of key responsibility among the peers can adjust the peers' index management load, and thus enables load balancing.

The **storage load balancing** method presumes that each peer has limited space available to store objects. To facilitate storage load balancing, BALLS allows objects to be located at peers different from the peer holding the key. Hence, an object can reside on any peer, regardless of its root (i.e., the peer holding the object key). The root needs only to keep a pointer to the location of the object. This separation enables the system to employ available space from any peer to place objects in case the corresponding root has reached its capacity. This enlarges thus the overall storage capacity. It also enables and facilitates simultaneous index management load balancing since modifying the key responsibility of a peer requires that only pointers of the involved objects be moved, but not the objects themselves. Moreover, this separation simplifies object replication, in which case the root of an object keeps the pointers to its replicas (on different peers). It thus enhances object availability without the need for further techniques.

Storage management must also consider resources required for file transfer. To that end, we take into account the network capacity of the peers to support object access and migration. The storage load balancing algorithm aims to minimize the overload with regard to these capacities (storage, access, migration) in the whole system. As it is the case with index management load balancing, the overload minimization goal allows us to save on rebalancing cost.

The load balancing algorithm is based on transferring storage load between pairs of peers if it results in a decrease of the global overload.

7.1.2 Related work

There exist several approaches to load balancing in structured P2P systems. A straightforward approach is the equalization of key responsibility assigned to the peers. It aims to maximize a metric called *smoothness*, defined as :

$$\min_{\forall p,q} \frac{|k_p|}{|k_q|},$$

where $|k_i|$ is the number of keys managed by peer i. Note that smoothness is usually defined as $\max \frac{|k_p|}{|k_q|}$ (i.e., inverse of $\min \frac{|k_p|}{|k_q|}$), which we believe is less intuitive as we would then minimize "smoothness" rather than maximize it. Bienkowski et al. [BKadH05] proposed a balancing method for P2P systems in which each peer occupies a key interval. The method categorizes the peers into *short*, *middle*, and *long* according to their interval's length. In order to increase smoothness, the balancing algorithm continuously makes short peers leave and rejoin by halving the interval of an existing long peer. Manku [Man04] aims at maximizing smoothness using a virtual balanced binary tree where the leaf nodes represent the current peers of the system and the path from the root to each leaf represents the splitting chain of the key space (being $[0, 1)$) to yield the corresponding peer's interval. The author proposed appropriate peer arrival and departure algorithms that maintain the balance of the binary tree. It thereby leads to a balanced key responsibility among the peers. In practice, the balance of load in structured P2P systems also depends on the size of objects, the popularity of objects, the distribution of objects over the peers, and the peers' capacities. If these factors are accounted for, a good smoothness does not ensure load balance.

There are some structured P2P systems [LKRG03, NW03a, WZLL04] using de Bruijn graphs for topology construction. Their load balancing methods also aim to increase smoothness via different arrival and departure algorithms. As discussed above, however, smoothness is not sufficient to balance load.

Several methods [KR04a, KR04b, RLS+03] transfer key responsibility among the peers to balance load. In such methods, heavily loaded peers move part of their key responsibility to lightly loaded peers, thereby shedding some load. The above methods tie the objects' location to the objects' key. They therefore do not enable index management load and storage load to be simultaneously balanced since balancing one aspect can break the balance of the other, and vice versa.

PAST [RD01b] is a P2P storage management system based on the Pastry [RD01a] routing substrate. Its load balancing approach uses a replica diversion technique that allows an object (file) to reside on a peer in the *leaf* set of its root[2] (refer to [RD01a] for the *leaf* set definition) when the root is full. However, PAST does not separate object location from object key. It maintains an invariant that limits object location within the root and the leaf set of the root.

Another method applies the *power of two choices* paradigm [BCM03] in which a set of multiple hash functions is used to map an object to a set of keys. This method balances storage load by storing each object at the least loaded peer in the set of peers responsible for the object's keys. The above methods relax the storage policy by breaking the tie of an object to one peer. However, they restrict object storage to a set of peers, which introduces a considerable network overhead in a dynamic environment.

Tapestry [ZHS+04] does not restrict where an object is stored (i.e., its storage location). Indeed, the publication of an object places pointers to the object on the peers along the routing

[2]In PAST, an object is replicated into a number κ of copies assigned, respectively, to κ peers with ids that are the closest to the object's id. The root here denotes the peer to which a copy is assigned.

path from the storing peer to the object's root. However, Tapestry does not support the independence of peer id with respect to key responsibility. With its high peer degree and complex maintenance procedure, this feature prevents the integration of efficient index management load balancing based on the dynamic transfer of key responsibility.

Expressways [ZSZ02], an extension of CAN [RFH+01], constructs the P2P network as a virtual zone span hierarchy where leaves are basic CAN zones. Based on this hierarchy, routing has logarithmic performance. This structure supports index management load balancing in which peers with higher capacities tend to shoulder the task of higher expressway levels where routing traffic has more probability to pass. This method has some disadvantages : (1) peer degree is high due to the multi-level routing table ; (2) balancing occurs only after aggregating the load and capacity information of all peers in the whole system ; and (3) the balancing goal is to keep equal the load/capacity ratio of the individual peers, which yields continuous restructuring even when it is not required.

7.1.3 Paper organization

The remainder of this paper is organized as follows. Section 7.2 describes BALLS and its load balancing methods in details. Section 7.3 presents the experimental evaluation of the system. Section 7.4 validates the advantages of BALLS. Section 7.5 provides a discussion about the system and the experimental results. The last section concludes the paper.

7.2 P2P system and load balancing algorithms

This section first describes the BALLS topology and its support for routing and maintenance. It is followed by the presentation of the two load balancing mechanisms.

7.2.1 The P2P topology

The construction of BALLS was inspired by de Bruijn graphs. A de Bruijn graph defines a set of nodes V and a set of arcs A as follows :

- $V = [0, k^m - 1]$ where $m \in \mathbb{N}$, $k \in \mathbb{N}$, and $k > 1$. Each de Bruijn node is labelled by an m-(k-ary) digit identifier,
- $A = \{(u, v) \,|\, u \in V, v \in V, v = (uk + l) \mod k^m, l \in [0, k - 1]\}$. Because of this rule, the degree (including both in and out arcs) of each de Bruijn node does not exceed $2k$.

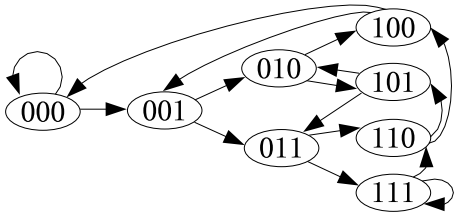

FIG. 7.1 – Binary de Bruijn graph with 8 nodes

Figure 7.1 shows an example of a binary de Bruijn graph. The de Bruijn graphs support efficient routing : The $route(u, v)$ algorithm described below performs routing from node u to node v. It requires at most m hops between any two nodes. This expresses a logarithmic bound of the routing cost in relation with the number of nodes.

route(u, v)

if $(u \neq v)$ {

$s = maxCRLSubstring(u, v)$;

$route((uk + v_{|s|}) \mod k^m, v)$;

}

Definition 1 *The longest common right-left substring of two nodes u and v, denoted*

$maxCRLSubstring(u, v)$, *is a string* $s \in \{0, \cdots, k-1\}^*$ *such that* $u = u_0 \cdots u_{m-|s|-1}s$, $v = sv_{|s|} \cdots v_{m-1}$, *and* $u_{m-|s|-1}s \neq sv_{|s|}$.

BALLS partitions a binary undirected de Bruijn graph $G = (V, A)$ of 2^m nodes (m is the predefined node id length). The node id space $[0, 2^m - 1]$ is identical to the key space. In the following, *de Bruijn node* and *key* are used interchangeably. Each peer p holds (is responsible for) a non-empty key interval, denoted[3] $[p.b, p.e]$. For convenience, all expressions on keys are implicitly modulo 2^m, e.g., $x + y$ indicates $(x + y) \bmod 2^m$. BALLS identifies each peer p by its network address, denoted $p.a$. The separation between peer id and keys enables the change of $[p.e, p.b]$ without affecting $p.a$.

BALLS maintains an invariant : every two peers p and q are connected, denoted $connect(p, q)$, if there is at least one de Bruijn arc connecting a key in $[p.b, p.e]$ and a key in $[q.b, q.e]$. In addition, p and q are connected if their key intervals are adjacent in the circular key space. We refer to this last connection type as a ring connection. The neighbourhood made up by a ring connection is called a ring neighbour. Ring connections will be needed in key interval exchange operations (e.g., index management load balancing, departure).

Definition 2 *The de Bruijn neighbourhood set of a key interval* I*, denoted* $dbneighbour(I)$*, is the set of every key connected to any key in* I *by a de Bruijn arc, except the keys in* I *itself.*

It is easy to show that $dbneighbour([b, e]) = ([2b, 2e + 1] \cup [\lfloor b/2 \rfloor, \lfloor e/2 \rfloor] \cup [\lfloor (b + 2^m)/2 \rfloor, \lfloor (e + 2^m)/2 \rfloor]) \setminus [b, e]$. Given two peers p and q with $I = [p.b, p.e]$ and $J = [q.b, q.e]$, we have the following formula.

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

$$connect(p,q) = \begin{cases} \text{true} & \text{if } (p.e = (q.b-1) \vee p.b = (q.e+1) \\ & \quad \vee (dbneighbour(I) \cap J \neq \emptyset) \\ & \quad \vee (dbneighbour(J) \cap I \neq \emptyset)) \\ \text{false} & \text{otherwise} \end{cases}$$

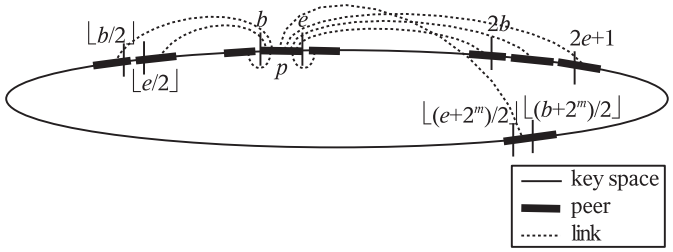

FIG. 7.2 – Example of connections from peer p in BALLS

Figure 7.2 illustrates the connections of a peer p whose key interval is $[b, e]$. The connection includes the links to the ring neighbours and to the peers holding a key interval that overlaps $dbneighbour([b, e])$.

Each peer p maintains a *neighbour list* that contains a triple $(q.a, q.b, q.e)$ for every neighbour q, i.e., the peers connected with p. Since the peer connection is undirected, p exists also in the neighbour list of q when q exists in the neighbour list of p. This facilitates key responsibility notification among the peers in comparison with other P2P systems that use directed peer connections (e.g., [KK03, MNR02]).

Loguinov et al. [LKRG03], and Naor and Weider [NW03a] use a P2P system similar to ours. They concentrate on the smoothness increase goal based on different arrival and departure algorithms. These approaches assume homogeneity of peers. In BALLS, we assume that peers are heterogeneous, that is, they have different network and storage capacities. Therefore, load balancing must be done for both types of resources.

BALLS's routing directs a message towards the peer holding a given key. We use a greedy algorithm that decreases step-by-step the distance from the current peer to the desti-

nation key. Before presenting the routing algorithm, we present some useful definitions.

In a binary 2^m-node de Bruijn graph, each node x has 4 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. Given two nodes x and y, the length of the de Bruijn routing path from x to y that follows only fore-arcs is called the fore-distance from x to y, denoted $foredistance(x, y)$. Similarly, the length of the de Bruijn routing path that follows only back-arcs is called back-distance, denoted $backdistance(x, y)$.

Definition 3 *The distance*[4] *between two keys x and y, denoted $distance(x, y)$, is the minimum among $foredistance(x, y)$ and $backdistance(x, y)$.*

Definition 4 *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 $\nexists v' \in I \mid distance(v', x) < distance(v, x)$.*

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

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

□

[4]Note that a routing path between two nodes in an undirected de Bruijn graph is not always the shortest path. The distance notation here does not imply the length of the shortest path. It is merely used for decision making in the greedy P2P routing algorithm.

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 $[0, 2^i - 1]$.

□

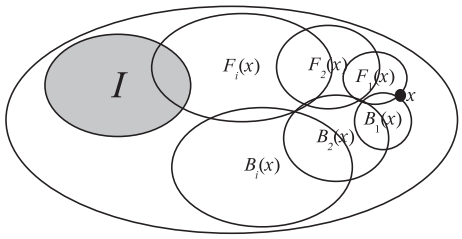

FIG. 7.3 – A set illustration of the $distance(I, x)$ algorithm

The $distance(I, x)$ algorithm shown below verifies the intersections $F_i(x) \cap I$ and $B_i(x) \cap I$ for i from 0 to m. If any $F_i(x) \cap I$ or $B_i(x) \cap I$ is not empty, it returns i. Figure 7.3 illustrates an approximate progression of the algorithm. The first non-empty intersection $F_i(x) \cap I$ gives distance i. The algorithm is efficient since it involves at most $m + 1$

iterations.

distance(I, x)

for ($i = 0$; $i \leq m$; i++) {

if ($[x * 2^i, x * 2^i + 2^i - 1] \cap I \neq \emptyset$) return i ;

$y_0 = \lfloor x/2^i \rfloor$; $s = \lfloor (I.e - y_0)/2^{m-i} \rfloor$;

if (($y_0 + s * 2^{m-i}$) $\in I$) return i ;

}

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

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

The set Γ can contain several disjoint key intervals. The notation $distance(\Gamma, x)$ specifies the smallest distance from the key intervals in Γ to x. Key t is determined by randomly choosing a value in $F_i(x) \cap \Gamma$ or $B_i(x) \cap \Gamma$ when distance i is found. The routing algorithm decreases $distance(t, x)$ by at least 1 at each hop. Thus, the routing cost, i.e., the number of routing hops, is bounded by m. Experiments in Section 7.3.1 yield even better performances.

BALLS uses straightforward peer arrival and departure algorithms. Since the system focuses on load balancing but not on smoothness, these algorithms can be simple and yet efficient. When a peer p joins the network, it finds the root of a random key via a known peer. As soon as the root (say r) is found, r splits $[r.b, r.e]$ into two, and transfers one half to p. Since the random key is evenly selected, peers with larger key intervals have higher probability to split. Peer p informs its new neighbours (whose addresses are also obtained from r) about its new key interval and sends an acceptance message to r. Peer r then updates its key interval, informs its neighbours about the new key interval, and adjusts its neighbour

list. Suppose that the average peer degree is d, this arrival algorithm needs on average $2d + 2$ messages to maintain the P2P connection structure. Experiments in Section 7.3.1 show that d is a constant and confirm the relationship between the arrival cost and d.

When a peer q departs, it selects its ring neighbour holding the shortest key interval (say o). Then q transfers its key interval to o. If o accepts, o updates its key interval as $[o.b, o.e] \cup [q.b, q.e]$, informs the neighbours about the new key interval, and replies with an acceptance message to q. Peer q then informs its neighbours about the departure and leaves after receiving their confirmation. This departure algorithm uses on average $3d + c + 1$ messages where d is the average peer degree and c is a constant denoting the number of messages needed to select the ring neighbour o. Experiments in Section 7.3.1 confirm this cost calculation.

In order to ensure integrity of the topology, when a peer is accepting the arrival or departure of another peer, it refuses any other concurrent arrival and departure requests addressed to it. There is also an integrity problem induced from concurrent decentralized updates. Examine the following scenario :

- at time t_i, peer p_1 transfers part of its key interval to peer p_2, which involves sending p_2 a link to peer p_3 : ($p_3.a, p_3.b, p_3.e$) ;
- at time t_{i+1}, p_3 updates its key interval and informs p_1 of the new $[p_3.b, p_3.e]$ (at this time, p_3 does not know about p_2) ;
- at time t_{i+2}, p_2 informs p_3 of its new key interval.

After time t_{i+2}, p_2 and p_3 connect but p_2 keeps incorrect information about p_3 : $[p_3.b, p_3.e]$. To solve this problem, node p sends its current values of $[q.b, q.e]$ to node q along with the key interval notification. If $[q.b, q.e]$ as known by p is different from the actual key interval, q sends a key interval notification back to p to correct it. This key interval notification protocol launches just enough messages for key interval updates. It does not introduce any additional costs to the maintenance.

7.2.2 Index management load balancing

Index management load refers to the network bandwidth consumed by each peer for routing messages. Due to the heterogeneous nature of the system, peers contribute different capacities, called the index management capacity, to support this load. The determination of the capacity available at each peer is out of the scope of this paper.

BALLS uses a decentralized method to distribute the index management load over the peers taking into account their capacity and current load state. A peer is overloaded when the load rises above the available capacity. The goal of load balancing is to minimize the global overload of the system. This goal reduces rebalancing costs since rebalancing is not required as long as the capacity supports the load. The balancing strategy is based on rearranging keys inside each pair of peers such that their combined overload is minimized. Because we separate peer ids and keys, a peer p may transfer some of the keys under its responsibility, thus some index management load, to a ring neighbour (i.e., a peer holding $p.b-1$ or $p.e+1$). Before going into the load balancing algorithm, we first describe how index management load is computed.

Index management load computation

The proposed index management load balancing approach requires the ability to determine the load on different subsets of a key interval. A simple solution to this requirement is to keep track of the routing traffic passing through each key in the interval. This solution becomes inefficient or even impossible when the key space size (2^m) is much higher than the number of peers.

Recall that a peer can only transfer keys to its ring neighbours to ensure that every peer holds a continuous key interval. Therefore, we only need to determine the routing traffic

passing through key subsets at the two ends of each peer's key interval.

Consequently, we propose the following approach to record traffic and compute index management load. We divide the key interval of each peer p into k levels where $k = \lfloor \log_2 s \rfloor$ and s is the size of $[p.b, p.e]$ (i.e., $s = p.e - p.b + 1$). Each level i, $i \in [0, k)$, is further broken down into 3 zones (see Fig. 7.4) : $z_{i,0} = [p.b, p.b + l_i - 1]$, $z_{i,1} = [p.e - l_i + 1, p.e]$, and $z_{i,2} = [p.b, p.e] \setminus (z_{i,0} \cup z_{i,1})$ where $l_i = \lfloor s/2^{i+1} \rfloor$. In the special case where $p.b = p.e$, only one level exists with $z_{0,0} = \{p.b\}$ and $z_{0,1} = z_{0,2} = \emptyset$.

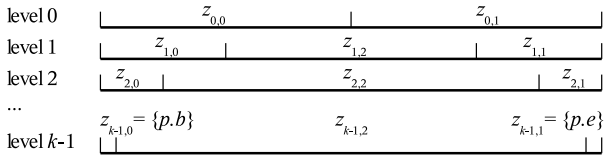

FIG. 7.4 – Dividing the key interval of peer p into zones

Each peer p manages a table $G_p[k][3]$ to register the routing traffic through the zones. According to the routing algorithm (Sect. 7.2.1), a routing message λ lands on a key t on each peer p in the path. For every level $i \in [0, k)$, if $t \in z_{i,j}$ then $G_p[i, j] = G_p[i, j] + |\lambda|$ where $|\lambda|$ denotes the size of λ. The routing traffic through peer p is $Tr_p = \sum_{j \in [0,2]} G_p[i, j]$, for any i.

The size of table G_p is always small because $k < m$. It ensures the efficiency of the routing traffic registration. Furthermore, this method allows us to determine the routing traffic through different portions at the two ends of $[p.b, p.e]$ with sizes ranging from 1 (e.g., $z_{k-1,0}$ or $z_{k-1,1}$) to $s - 1$ (e.g., $z_{k-1,0} \cup z_{k-1,2}$ or $z_{k-1,1} \cup z_{k-1,2}$).

In order to monitor index management load, which can dynamically vary over time, we must periodically reset table G_p. Denoting the period as δt, the starting time of the current period as t_0, and the current time as t_c, the current index management load is $T_p = Tr_p/(t_c - t_0)$. We sometimes need to calculate T_p when $t_c - t_0$ is too small which may result in an incorrect load. In this case, we use the formula $T_p = (Tr'_p + Tr_p)/(t_c - t'_0)$ where Tr'_p and

t'_0 are, respectively, the routing traffic and the starting time of the previous period.

We denote the index management capacity of peer p as C_p. The overload is formally given by $O_p = (T_p - C_p + |T_p - C_p|)/2$. Every peer p permanently executes the following load monitoring procedure :

1. if p has changed $p.b$ or $p.e$, go to step 4. Otherwise, go to step 2 ;
2. if $t_c - t_0 < \delta t$, go to step 1. Otherwise, go to step 3 ;
3. if $O_p > 0$, execute the index management load balancing algorithm on p. Go to step 4 ;
4. reset table $G_p[k][3]$ with $k = \lfloor \log_2(p.e - p.b + 1) \rfloor$. Set $t_0 = t_c$. Go to step 1.

Index management load balancing algorithm

The index management load balancing algorithm on a peer p starts when $O_p > 0$. It consists in transferring some part of $[p.b, p.e]$ to a ring neighbour q so as to minimize the combined overload of p and q. We denote the ring neighbours of p as $n_0(p)$ (that holds $p.b-1$) and $n_1(p)$ (that holds $p.e+1$). The zones that can be moved to $n_j(p)$ are $z_{i,j}$ and $z_{i,j} \cup z_{i,2}$ (for $i \in [0, k)$ and $j \in [0, 1]$). The interval selected to be moved must : (1) maximize the reduction of the combined overload $O_p + O_{n_j(p)}$, and (2) be as small as possible. These criteria ensure that the global overload is minimized while entailing the least rebalancing cost.

Selecting the optimal zone to move requires knowledge on the load and capacity of p and $n_j(p)$. Maintaining such information among neighbouring peers is very expensive. If p asks $n_j(p)$ about this information before transferring, it slows down the procedure. Our solution allows p to propose a set of candidate zones to $n_j(p)$. On its side, $n_j(p)$ selects the best zone based on its local load status and the information received. This solution needs only one query-answer exchange between the peers. We let $w_{h,j}$ ($h \in [0, 2k)$ and $j \in [0, 1]$) denote the candidate zones for transfer. They are determined as follows (Fig. 7.5) :

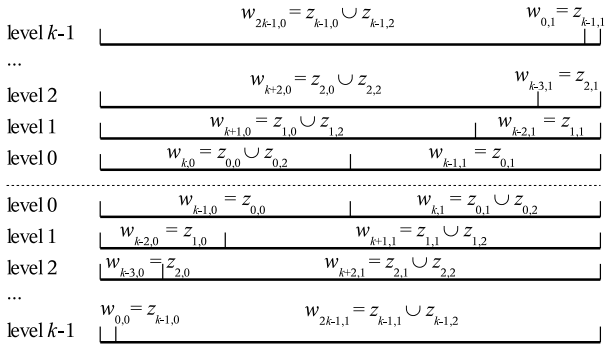

FIG. 7.5 – Candidate zones to transfer ($w_{h,j}$)

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

The index management load on $w_{h,j}$, denoted $T(w_{h,j})$, is given by :

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

The index management load balancing algorithm on a peer p (launched when $O_p > 0$) :

1. select the smallest $h \in [0, 2k)$ such that $\exists j \in [0, 1]$ and $T_p - T(w_{h,j}) \leq C_p$. Execute the key interval transfer algorithm (presented below) for $w_{h,j}$ from p to $n_j(p)$. If the transfer succeeds, stop the load balancing algorithm. Otherwise, continue with step 2 ;
2. set $l = (j+1) \bmod 2$. Select the smallest $h \in [0, 2k)$ such that $T_p - T(w_{h,l}) \leq C_p$. Execute the key interval transfer algorithm for $w_{h,l}$ from p to $n_l(p)$. This step stops the load balancing algorithm even if the transfer does not succeed since both ring neighbours of p have been tried.

The key interval transfer algorithm for $w_{h,j}$ from a peer p to a ring neighbour $n_j(p)$ allows $n_j(p)$ to receive one of the zones $w_{0,j}, w_{1,j}, \cdots, w_{h,j}$ from p that will minimize the

combined overload $O_p + O_{n_j(p)}$. The algorithm is straightforward. It first selects the zone that decreases the overload of p the most while keeping $n_j(p)$ underloaded (step 2a). It may be the case that no such zone is found. If the selection stops here, the transfer of load from p to $n_j(p)$ is blocked because $n_j(p)$ (being underloaded) does not intend to shed any part of its key interval. In this case, the algorithm selects the smallest zone that can decrease the combined overload to continue the load balancing (step 2b). The transfer involves the following steps :

1. p sends to $n_j(p)$ a key interval transfer message containing O_p, the list $(w_{0,j}, w_{1,j}, \cdots, w_{h,j})$, and the list $(T(w_{0,j}), T(w_{1,j}), \cdots, T(w_{h,j}))$;
2. if $n_j(p)$ is busy with another operation[5] or $O_{n_j(p)} \geq 0$, it refuses the transfer. Otherwise,
   (a) $n_j(p)$ searches for the greatest $g \in [0, h]$ that gives $T_{n_j(p)} + T(w_{g,j}) \leq C_{n_j(p)}$;
   (b) if no such g is found, $n_j(p)$ searches for the smallest $g \in [0, h]$ satisfying

$$|T(w_{g,j}) - O_p| + T(w_{g,j}) - O_p + 2(T_{n_j(p)} - C_{n_j(p)}) < 0 \tag{7.1}$$

      i. if no such g exists, $n_j(p)$ rejects the transfer ;
      ii. if g is found, $n_j(p)$ sets the chosen candidate zone index as g ;
3. if a candidate zone index g is chosen (by step 2a or 2(b)ii), $n_j(p)$ adds $w_{g,j}$ to its key interval, adjusts the connections to its neighbours, and replies to p an acceptance message specifying g ;
4. on receiving the acceptance message from $n_j(p)$, p removes $w_{g,j}$ from its key interval and releases the connections to the peers that are no longer its neighbours. The transfer then succeeds ;
5. if the proposal of p is rejected by $n_j(p)$, the transfer fails.

[5] $n_j(p)$ may be participating in the arrival, departure, or index management load balancing with another peer.

Conjecture 1 *The key interval transfer algorithm for an interval $w_{h,j}$ from a peer p to a ring neighbour $n_j(p)$, if it succeeds, will minimize the combined overload of p and $n_j(p)$.*

Proof : The key interval transfer succeeds only if a candidate zone index g is chosen in step 2a or 2(b)ii. Recall that the routing algorithm (Sect. 7.2.1) limits the choice of the next step key t in the de Bruijn neighbourhood set of the current peer. It follows that moving $w_{g,j}$ will transfer a load approximately[6] equal to $T(w_{g,j})$. Before the move, the overloads of the peers are, respectively, $O_p = (T_p - C_p + |T_p - C_p|)/2 = T_p - C_p$ and $O_{n_j(p)} = 0$. After the move, the overloads become, respectively :

$$\begin{aligned} O'_p &= (O_p - T(w_{g,j}) + |O_p - T(w_{g,j})|)/2 \\ O'_{n_j(p)} &= (T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)} + |T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)}|)/2 \end{aligned}$$

The condition for minimizing the combined overload is to minimize $\Delta O = O'_p + O'_{n_j(p)} - O_p - O_{n_j(p)}$.

If g is chosen by step 2a, $T_{n_j(p)} + T(w_{g,j}) \leq C_{n_j(p)}$. Then $O'_{n_j(p)} = 0$ and $\Delta O = O'_p - O_p < 0$. Because g is the largest possible, it induces the largest possible $T(w_{g,j})$ and thus the most negative ΔO.

If g is chosen by step 2(b)ii, $T_{n_j(p)} + T(w_{g,j}) > C_{n_j(p)}$ and (7.1) holds. It is easy to prove that the left hand side of (7.1) is $2\Delta O$ and the smallest possible g chosen induces the most negative ΔO.

□

The above algorithm ensures criterion (1) for the key interval transfer. On the other hand, the selection of the smallest h in steps 1 and 2 of the index management load balancing

[6]Because of the de Bruijn graph routing property, it cannot be guaranteed that an exact traffic through the zone moved is actually transferred to the new peer.

algorithm satisfies criterion (2), which aims at making the minimal update.

7.2.3 Storage location and key separation

The storage load of a peer p (denoted S_p) is the sum of the sizes of all objects the peer currently stores. Each peer p contributes a limited space to store objects, called the storage capacity (D_p). The system ensures the invariant $S_p \leq D_p$ on every peer p. In a P2P system where objects must reside on their root, maintaining this invariant yields the rejection of inserting a new object when its root is full (even if storage space is available on other peers). This degrades the system's storage capacity. Moreover, a redistribution of the storage load to fit the peers' storage capacity affects key distribution, which is managed by the index management load balancing.

BALLS solves the above inconveniences by allowing objects to be located at peers other than their root (i.e., the peer holding their key). A root keeps track of the objects under its responsibility via pointers, called *storage pointers*, that keep the address of the peers storing the objects. An object keeps track of its root through a pointer, called *root pointer*. This approach greatly improves the utilization of the globally available storage capacity. It enables new objects to be stored as long as sufficent space remains in the system. More importantly, rearranging objects when performing storage load balancing (presented in Sect. 7.2.4) does not influence index management load balancing. This approach also ensures the efficiency of the index management load balancing because the transfer of a key interval only moves the storage pointers of the objects involved, not the objects themselves. This therefore saves on data migration costs.

Another advantage of this approach is that it simplifies replication, which enhances object availability. Because of the separation of object locations and keys, storing replicas of an object on different peers is natural without the need of an additional special technique such

as multiple mapping hash functions (e.g., [BCM03]) or overlapping key responsibility (e.g., [NW03b]). BALLS allows up to κ replicas of an object to be stored. The root keeps storage pointers to the replicas as for individual objects. For convenience, we will use the term object to imply an object or one of its replica interchangeably in this paper.

Pointer management

Pointer management maintains the consistency of storage pointers and root pointers. Every peer disposes of two tables : $indices$ and $storage$. Each item in table $indices$ represents the index of an object managed by the peer. Such an item consists of the object id (oid) and the list of storage pointers to the object's replicas. A storage pointer consists of the replica id (rid - an integer in the range $[0, \kappa)$), the address of the storing peer ($location$), and the storage counter ($counter$). The $counter$ field (its use will be explained below) is initially set to 0 and incremented after each change of $location$. The $storage$ table contains the objects currently stored on the peer. The header of each item in this table consists of the object id (oid), the replica id (rid), the size ($size$), the root pointer ($root$ being the address of the root), and the storage counter ($counter$). BALLS uses two protocols : *storage notification* and *root notification* to keep the $indices$ and $storage$ tables consistent.

The storage notification protocol informs the root of an object about the object's location. When a peer p accepts an object o, $o.counter$ is incremented. Peer p then sends a storage notification message σ to the actual root of o. The message contains $p.a$ and (oid, rid, $counter$) of o. We anticipate the following problems :

– due to the heterogeneity and decentralization properties of the P2P system, there may be more than one storage notification message (of an object) arriving at the root in an order other than the order of the generation time. The $counter$ field lets the root know whether the notification received is newer than the corresponding storage pointer it

holds. If this is the case, the root updates the storage pointer;

- the root of o may change when p launches the notification message. Message σ therefore attaches a $root$ field specifying the root pointer as known by p. If a peer that receives σ is not the actual root of o, or if the sending peer p finds that the peer at address $\sigma.root$ no longer exists (because it has left the overlay network), σ is sent to o's key using the routing algorithm. Forwarding σ in this case does not influence index management load balancing since the load calculation only counts routing messages created by lookups from users but not by notification messages. In other words, it determines the load according to the requests but not to object migration. If the actual root of o (say r) receives σ, it compares $\sigma.root$ with $r.a$. If $\sigma.root \neq r.a$ (i.e., p keeps an incorrect root pointer), r sends back a root notification (explained below) to p for the correction;
- in the case where o moves from peer p to peer q, it may still be accessed on p after the move but before the root receives the storage notification. To solve this problem, p keeps a storage pointer to q. As soon as the root updates o's storage pointer, the corresponding pointer on p is deleted.

The root notification protocol informs an object about its root. When a key interval moves from a peer to another peer (say q), q sends root notification messages to the peers storing the objects of which the key was moved. A root notification message ρ for an object o contains $q.a$, (oid, rid, $counter$) of o, and a $location$ field specifying the address of the peer storing o as known by q. When a peer s that actually stores o receives ρ, it updates the corresponding root pointer and verifies that q keeps a correct storage pointer (using the information received). If this is not the case, s sends a storage notification message to q to perform the correction.

The two notification protocols introduced seem to complicate system maintenance. However, in comparison to traditional P2P systems that tie objects to their roots, the transfer of

a key interval in BALLS remains much less costly. It only requires sending storage pointers and notification messages whose sizes, in practice, are much smaller than the object sizes.

Object insertion

The object insertion algorithm adds up to κ replicas of a new object to different peers while ensuring $S_p \leq D_p$ on every peer p. An insertion request containing the object id (oid) and the size ($size$) is routed to the root of oid. If an index entry with the same oid as the object to insert already exists, the request is rejected. Otherwise, a replica diffusion process is launched to store the object on different peers, starting at the root.

This process sends a replica diffusion message λ_r containing oid, $size$, and $ridlist$ – the list of the remaining rids to be assigned. This message traverses multiple peers in order to distribute the replicas. It limits the number of peers visited using a ttl (time-to-live) field, which is decremented after each hop. At each peer q on the way, if $S_q + size \leq D_q$ and q does not store any replica of the same object, q accepts one replica with a rid extracted from $\lambda_r.ridlist$. If $\lambda_r.ridlist$ is not empty and $\lambda_r.ttl > 0$, λ_r is forwarded to a randomly selected not-visited peer neighbouring q. The message keeps a list of the visited peers to perform this verification. The insertion stops when κ replicas have been stored. If $\lambda_r.ttl = 0$, there are two cases : (1) if no replica was stored, the insertion fails, and (2) if the number of stored replicas falls between 1 and $\kappa - 1$, the root starts a new replica diffusion message for the remaining rids. During the diffusion process, if the key of the inserted object moves to another peer, the new root continues the process perserving the current state.

We do not discuss the object deletion here since it never increases the storage load on any peer.

7.2.4 Storage load balancing

In addition to storage space, storing objects consumes other computational resources, e.g., network bandwidth for object access and migration. These resources must be considered in the balancing algorithm.

For the system to work properly, we need another boundary for the storage load called the desired storage capacity $\overline{D}_p$ on every peer p, with $\overline{D}_p \leq D_p$. The storage load S_p is always limited by D_p but can temporarily exceed $\overline{D}_p$. Given $A_p = \overline{D}_p - S_p$ denoting the available space on peer p, we define the storage overload as $W_p = (|A_p| - A_p)/2$. Peer p is overloaded when $W_p > 0$. The goal of storage load balancing is to minimize the storage overload of the whole system ($\sum_{\forall p} W_p$) while keeping $S_p \leq D_p$ on every peer.

The storage load balancing algorithm consists in transferring storage load within pairs of peers so as to minimize their combined overload. Such transfers lead to the reduction of the global overload. Suppose that we perform a storage load transfer within a pair of peers p and q where p is overloaded ($A_p < 0$). The transfer decreases the combined overload $W_p + W_q$ only if $A_q > 0$. The storage load balancing should minimize the global overload while saving on the transfer cost (i.e., the data volume migrated). Therefore, the elementary storage load transfer within each pair of peers has the following objectives : (1) minimizing their combined overload, and (2) minimizing transfer cost.

In general, we cannot achieve both objectives at the same time. Therefore, priority must be given to one or the other : (1) before (2) or (2) before (1). These two different orders are referred to as storage load transfer strategies. The overload-oriented transfer strategy minimizes overload first, while the cost-oriented transfer strategy considers transfer cost first.

Another element to consider in storage balancing is whether transfer is unidirectional (i.e., only one peer transfers object) or bidirectional (i.e., both peers may transfer objects).

We refer to these as the 1-direction transfer and 2-direction transfer, respectively.

The 1-direction storage load transfer

An 1-direction storage load transfer is a storage load S_{pq} transfered by overloaded peer p to a peer q (with $A_q > 0$).

Definition 5 *The optimal 1-direction storage load transfer is an 1-direction storage load transfer that upholds :*

- *(1) the combined overload of p and q is minimized,*
- *(2) S_{pq} is the smallest possible.*

Therefore, the optimal 1-direction storage load transfer is the least costly transfer that yields the fastest decrease of the combined overload $W_p + W_q$.

Definition 6 *The bounded optimal 1-direction storage load transfer is an 1-direction storage load transfer that upholds :*

- *(1) the combined overload of p and q is minimized,*
- *(2) S_{pq} is smaller or equal to reduction of the combined overload.*

Hence, the bounded optimal 1-direction storage load transfer is the most effective transfer (i.e., the one minimizing $W_p + W_q$), in which the migration cost does not exceed the benefit of overload reduction.

Theorem 1 *Given two peers p, q, with $A_p < 0$ and $A_q > 0$,*

- *the optimal 1-direction storage load transfer occurs when S_{pq} is the closest to $\min(-A_p, A_q)$ such that $S_{pq} < -A_p + A_q$;*
- *the bounded optimal 1-direction storage load transfer occurs when S_{pq} is the greatest satisfying $S_{pq} \leq \min(-A_p, A_q)$.*

The proof is presented in Appendix 7.A.

The 2-direction storage load transfer

The 2-direction storage load transfer is a pair of storage loads (S_{pq}, S_{qp}) such that S_{pq} is transfered from peer p to peer q, and S_{qp} is transfered from q to p, where $A_p < 0$ and $A_q > 0$. Obviously, $W_p + W_q$ decreases only if $0 \leq S_{qp} < S_{pq}$.

Definition 7 *The optimal 2-direction storage load transfer is the 2-direction storage load transfer that upholds condition (1) first, then condition (2) :*

- *(1) the combined overload of p and q is minimized,*
- *(2) S_{qp} is the smallest possible.*

As in the 1-direction transfer mode, the conditions of this definition guarantee the greatest reduction of $W_p + W_q$ while requiring the smallest S_{qp} to save on transfer cost. We do not consider the "bounded" constraint here because a 2-direction transfer usually incurs a migration cost higher than the overload reduction.

Theorem 2 *Given two peers p, q, with $A_p < 0$ and $A_q > 0$, and S_{pq}, the optimal 2-direction storage load transfer is obtained as follows :*

- *if $S_{pq} \leq A_q$ or $A_q < S_{pq} \leq -A_p$, $S_{qp} = 0$;*
- *if $S_{pq} > \max(-A_p, A_q)$, S_{qp} is the closest to $\min(A_p, -A_q) + S_{pq}$ such that $0 \leq S_{qp} < S_{pq}$ and $S_{qp} > A_p - A_q + S_{pq}$.*

See Appendix 7.B for the proof of this theorem.

It follows from these theorems that the most effective ΔW that both 1-direction and 2-direction transfers can achieve is $\max(A_p, -A_q)$ (also refer to the appendices). In other words, these transfer styles have the same bound.

We now return to the overload-oriented and cost-oriented transfer strategies. An overload-oriented transfer tries to minimize the combined overload before considering the transfer cost involved. On the other hand, a cost-oriented transfer restricts the transfer cost not to exceed the gain while decreasing the combined overload. To fulfil its requirement, the cost-oriented transfer strategy applies the bounded optimal 1-direction transfer. The overload-oriented transfer strategy is more complicated. It first searches for the optimal 1-direction transfer. If no S_{pq} relevant to such a transfer is found, it searches for the optimal 2-direction transfer.

In a decentralized environment, an object move consists in the proposal of an object set R by the sending peer (say p) and the acceptance of a subset $R' \subseteq R$ by the receiving peer (say q). To avoid sending an object to different peers simultaneously, we define two object states : *moving* (i.e., the object is in a proposal) and *normal* (i.e., the object is not in any move). Therefore, when peer p proposes set R to q, these objects are marked as *moving* on p. After q accepts objects in R', p deletes R' and restores the *normal* state of the remaining objects in $R \setminus R'$. Other simultaneous moves only select objects in the *normal* state. We denote the set of the objects stored on p as R_p, the set of its objects in the *normal* state as $\overline{R}_p$, the storage load of $\overline{R}_p$ as $\overline{S}_p$, and $\overline{A}_p = \overline{D}_p - \overline{S}_p$.

The storage load balancing algorithm : each peer p periodically verifies its overload state. When $\overline{A}_p < 0$, p starts a balancing session :

1. p broadcasts to its neighbourhood an available space interrogation message ϕ with a ttl (time-to-live) field defining the diffusion depth. If it has not received ϕ before, each receiving peer q processes ϕ and decrements $\phi.ttl$. After decrementing, if $\phi.ttl > 0$, q forwards ϕ to its neighbours except p and the peer from which ϕ arrives. The processing of ϕ on q involves replying $A_q = \overline{D}_q - S_q$ to p if $A_q > 0$;
2. for each reply A_q, if $\overline{A}_p < 0$ still holds, p proposes to q an object set $R_{pq} \subseteq \overline{R}_p$ that

satisfies one of the following conditions (denoting the storage load of a set R as $S(R)$) :

$$\begin{cases} S(R_{pq}) \leq A_q \\ S(R_{pq}) \geq -\overline{A}_p \\ \nexists r \in R_{pq} \,|\, S(R_{pq} \setminus \{r\}) \geq -\overline{A}_p \end{cases} \tag{7.2}$$

$$\begin{cases} S(R_{pq}) \leq A_q \\ S(R_{pq}) < -\overline{A}_p \\ \nexists r \in \overline{R}_p \setminus R_{pq} \,|\, S(R_{pq} \cup \{r\}) \leq A_q \end{cases} \tag{7.3}$$

$$\begin{cases} S(R_{pq}) > A_q \\ |R_{pq}| = 1 \\ \nexists r \in \overline{R}_p \,|\, S(\{r\}) < S(R_{pq}) \end{cases} \tag{7.4}$$

(7.2) selects the smallest $S(R_{pq})$ that cancels overload on p without overloading q. If (7.2) cannot be satisfied, (7.3) selects the greatest $S(R_{pq})$ that reduces W_p without overloading q. If both (7.2) and (7.3) cannot be satisfied (i.e., every object in $\overline{R}_p$ is greater than A_q), (7.4) assigns the smallest object of $\overline{R}_p$ to R_{pq} ;

3. on receiving the R_{pq} proposal, q behaves differently according to the storage load transfer strategy (cost-oriented or overload-oriented) selected a priori.

In the cost-oriented storage load transfer, q selects $R'_{pq} \subseteq R_{pq}$ as follows, where $pivot = \min(-\overline{A}_p, A_q)$:

$$\begin{cases} 0 < S(R'_{pq}) \leq pivot \\ \nexists r \in R_{pq} \setminus R'_{pq} \,|\, S(R'_{pq} \cup \{r\}) \leq pivot \\ S(R'_{pq}) + S_q \leq D_q \\ \nexists (r, r') \,|\, r \in R'_{pq} \wedge r' \in R_q \wedge r.oid = r'.oid \end{cases} \tag{7.5}$$

Condition (7.5) selects R'_{pq} that will make the bounded optimal 1-direction transfer as indicated by Theorem 1. R'_{pq} also ensures that $S_q \leq D_q$ and does not contain any replica of an object already stored on q (which causes a replica conflict). If no such R'_{pq} is found, q refuses the proposal. Otherwise, it accepts R'_{pq}. The transfer is completed.

In the overload-oriented storage load transfer, q first selects an object set R'_{pq} as follows, where $pivot = \min(-\overline{A}_p, A_q)$:

find $R1 \subseteq R_{pq}$ satisfying (7.6)

$$\begin{cases} 0 < S(R1) < pivot \\ \nexists r \in R_{pq} \setminus R1 \,|\, S(R1 \cup \{r\}) < pivot \\ S(R1) + S_q \leq D_q \\ \nexists (r, r') \,|\, r \in R1 \wedge r' \in R_q \wedge r.oid = r'.oid \end{cases} \tag{7.6}$$

find $R2 \subseteq R_{pq}$ satisfying (7.7)

$$\begin{cases} pivot \leq S(R2) < -\overline{A}_p + A_q \\ \nexists r \in R2 \,|\, S(R2 \setminus \{r\}) \geq pivot \\ S(R2) + S_q \leq D_q \\ \nexists (r, r') \,|\, r \in R2 \wedge r' \in R_q \wedge r.oid = r'.oid \end{cases} \tag{7.7}$$

identify R'_{pq} as follows, where $\Delta W(S_{pq})$ is the variation of the combined overload as defined by (7.10) :

$$R'_{pq} = \begin{cases} R1 & \text{if } \exists R1 \wedge \nexists R2 \\ R1 & \text{if } \exists R1 \wedge \exists R2 \wedge \Delta W(S(R1)) \leq \Delta W(S(R2)) \\ R2 & \text{if } \exists R2 \wedge \nexists R1 \\ R2 & \text{if } \exists R1 \wedge \exists R2 \wedge \Delta W(S(R1)) > \Delta W(S(R2)) \end{cases}$$

The above conditions try to select R'_{pq} to make the optimal 1-direction transfer as stated in Theorem 1. They also ensure that R'_{pq} does not contain any replica of an object already stored on q and guarantees $S_q \leq D_q$. If such an R'_{pq} is identified, q accepts R'_{pq} and the transfer is completed. Otherwise (i.e., $\nexists R1$ and $\nexists R2$), the transfer continues in the 2-direction transfer mode.

In the 2-direction transfer, q selects an acceptable object set $R'_{pq} \subseteq R_{pq}$ such that

$$\begin{cases} S(R'_{pq}) + S_q \leq D_q \\ \nexists r \in R_{pq} \setminus R'_{pq} \,|\, S(R'_{pq} \cup \{r\}) + S_q \leq D_q \\ \nexists (r, r') \,|\, r \in R'_{pq} \wedge r' \in R_q \wedge r.oid = r'.oid \end{cases}$$

This selection gives the greatest R'_{pq} that guarantees $S_q \leq D_q$ and does not contain any replica of an object already on q. If no such R'_{pq} is found, q refuses the proposal. In case q finds an R'_{pq}, it selects an object set to send back R_{qp} according to the following rule, where $pivot = \min(\overline{A}_p, -A_q) + S(R'_{pq})$:

find $R1 \subseteq \overline{R}_q$ satisfying (7.8)

$$\begin{cases} 0 \leq S(R1) < pivot \\ S(R1) > \overline{A}_p - A_q + S(R'_{pq}) \\ \nexists r \in \overline{R}_q \setminus R1 \,|\, S(R1 \cup \{r\}) < pivot \end{cases} \tag{7.8}$$

find $R2 \subseteq \overline{R}_q$ satisfying (7.9)

$$\begin{cases} S(R2) \geq pivot \\ S(R2) < S(R'_{pq}) \\ \nexists r \in R2 \,|\, S(R2 \setminus \{r\}) \geq pivot \end{cases} \tag{7.9}$$

identify R_{qp} as follows, where $\Delta W(S_{qp})$ is the variation of the combined overload as

defined by (7.12) :

$$R_{qp} = \begin{cases} R1 & \text{if } \exists R1 \wedge \nexists R2 \\ R1 & \text{if } \exists R1 \wedge \exists R2 \wedge \Delta W(S(R1)) \leq \Delta W(S(R2)) \\ R2 & \text{if } \exists R2 \wedge \nexists R1 \\ R2 & \text{if } \exists R1 \wedge \exists R2 \wedge \Delta W(S(R1)) > \Delta W(S(R2)) \end{cases}$$

The selection aims to identify an object set R_{qp} that results in the optimal 2-direction storage load transfer, as required in Theorem 2. If $\nexists R1$ and $\nexists R2$, R_{qp} is not identified and q refuses the proposal. Otherwise, it accepts R'_{pq} and proposes R_{qp} to p.

Upon receiving the R_{qp} proposal, p selects $R'_{qp} \subseteq R_{qp}$ such that

$$\begin{cases} S(R'_{qp}) + S_p \leq D_p \\ \nexists r \in R_{qp} \setminus R'_{qp} \,|\, S(R'_{qp} \cup \{r\}) + S_p \leq D_p \\ \nexists (r, r') \,|\, r \in R'_{qp} \wedge r' \in R_p \wedge r.oid = r'.oid \end{cases}$$

This maximizes R'_{qp} while guarantying that $S_p \leq D_p$, and makes no replica conflict. Then p accepts R'_{qp} and the transfer is completed.

The storage load balancing algorithm uses $\overline{A}_p = \overline{D}_p - \overline{S}_p$ instead of $A_p = \overline{D}_p - S_p$ (as in Theorems 1 and 2) to enable parallel object transfers from the overloaded peer p to different underloaded peers. The inherent parallelism accelerates the minimization of the global overload.

Our implementation uses a combination of several greedy algorithms for the selection of R_{pq}, R'_{pq}, R_{qp}, and R'_{qp}. Although the greedy algorithms do not always give the optimum, they ensure good computational efficiency.

7.3 Experimental evaluation

In order to evaluate the performance of the proposed P2P system and load balancing methods, we have performed various simulations. The simulator was developed in Java. We executed simulations on different computers with Pentium 4 CPU, (512MB - 2GB) RAM, and OS Windows XP or Red Hat Fedora Core 3. The reader can refer to [LBK06b] for details about the simulator. This section presents the results obtained. The simulations were designed to evaluate the P2P topology, the index management load balancing, and the storage load balancing. We also tested the influence of the index management load balancing on the storage load balancing, and vice versa. In all experiments presented, the P2P network partitions a de Bruijn graph of 2^{32} nodes ($m = 32$). It thus manages the key space $[0, 2^{32} - 1]$.

Each characteristic was evaluated with multiple simulation runs. This required us to perform statistical analyses of the results obtained. The analyses are of two types : (1) verification of the confidence interval of the results, and (2) comparison of two or more groups of results. We usually present the results in the form of a graph of averages of the measured values. The first analysis calculates the confidence interval of each point in the graph. The confidence interval of a variable x is the interval $[x_l, x_u]$ such that $P(\overline{x} \notin [x_l, x_u]) \leq \alpha$ with a given α and with $\overline{x}$ denoting the mean of x. Narrower confidence intervals show more closeness of the measured means to the true means. In this paper, we use $\alpha = 1\%$.

Comparisons between groups of results are used to determine if the groups differ. A group of results is produced by an experiment condition and presented by a graph (of average values). Through the comparisons, we observe the effect of the chosen conditions on the experiment results. Suppose that we compare two graphs X and Y. The comparison consists in calculating the confidence interval (with a given α) of $(X_i - Y_i)$ for all values i on the horizontal axis. If the confidence interval of $(X_i - Y_i)$ does not include 0, X_i and Y_i are statistically different. We name the rate of pairs (X_i, Y_i) having confidence interval disjoint

from 0 as the difference rate. Higher difference rates show more evidence of differences between X and Y. With lower difference rates (e.g., under 10%), we cannot conclude that X and Y are different. When performing a comparison between more than two graphs, we perform a pair-wise comparison.

7.3.1 P2P topology evaluation

The evaluation of the P2P topology consists in observing the following properties : average and distribution of peer degrees, average routing costs, average arrival costs, and average departure costs. The graphs in this subsection plot the average or combined results of 30 simulations. A simulation starts from a system of one peer. Peers arrive and depart, respectively, with probabilities related to the current number of peers. The arrival probability is higher than the departure probability. This makes the system size grow as a Markov chain. We measured the properties until the system reached 2100 peers.

During the simulation, index management load balancing is active. Peer arrivals/departures and index management load balancing continuously modify the key intervals of the peers, which causes high dynamicity.

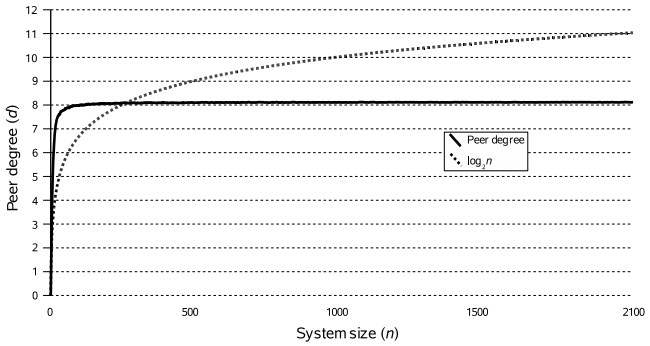

FIG. 7.6 – Average peer degree d vs. system size n

Figure 7.6 shows the average peer degree in function of system size (n). Regardless of the value of n, the average peer degree stays near 8. We obtained the same results, regardless

of the de Bruijn node id length m. We computed the confidence interval of the average peer degree for all system sizes from 1 to 2100. The limits obtained for each system size are so close to the measured average peer degree that we cannot distinguish them. The distance of the upper and lower boundaries of these intervals to the measured average peer degree gives the following statistics : max = 0.13, mean = 0.01, and median = 0.01. That is, the confidence interval for any system size is on average $[8.05, 8.07]$. The intervals are very close in comparison to the measured average peer degree 8.06.

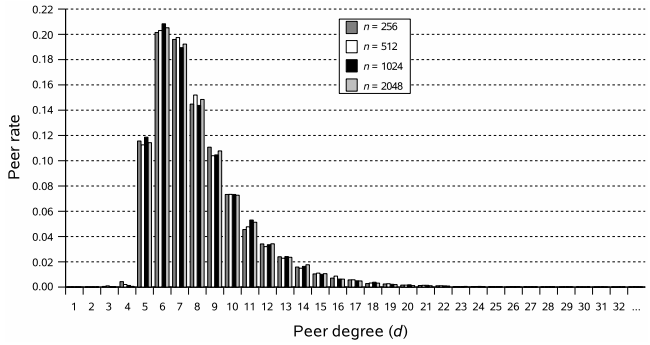

FIG. 7.7 – Peer degree distribution for different system sizes

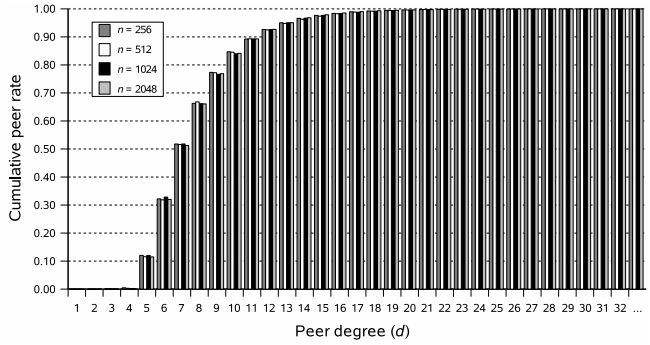

FIG. 7.8 – Cumulative peer degree distribution for different system sizes

Figure 7.7 shows the peer degree distribution calculated when n reaches 2^8, 2^9, 2^{10}, and 2^{11}. Figure 7.8 plots the cumulative rate of the peers having a degree lower than or equal to each value from 1 to 32. Although the network size increases, the distribution shape (in both graphs) almost does not change. The highest rates belong to the peers with degree from 5 to

11. It approaches 0 when the peer degree exceeds 20. Theses observations demonstrate the constant peer degree feature of the P2P topology.

It should be noted that low peer degree may lead to a fragile P2P system. We can resolve this problem by accepting some redundancy, e.g., letting each peer maintain links to multiple peers in a larger neighbourhood. The trade-off needs to be made between low peer degree and resilience to failure. Note, however, that this problem is beyond the scope of the present paper. In the following, we therefore only consider the case without redundancy.

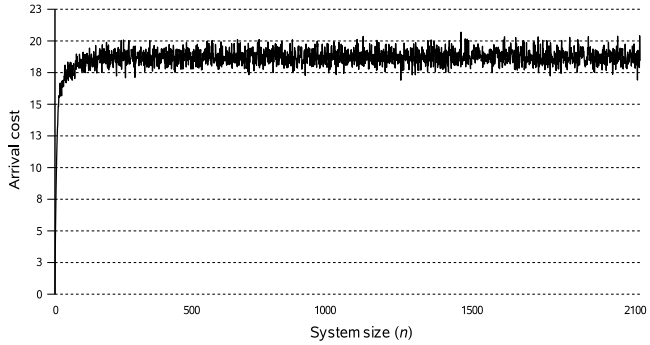

FIG. 7.9 – Arrival cost vs. system size n

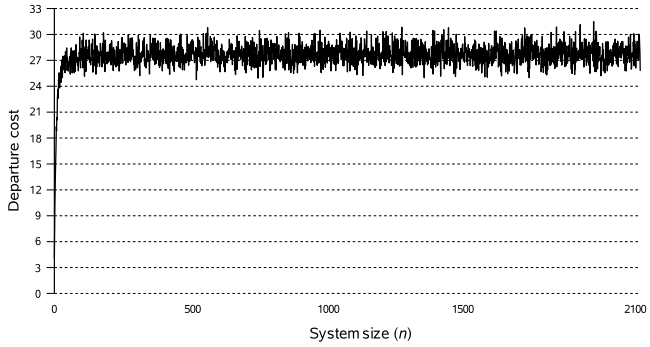

FIG. 7.10 – Departure cost vs. system size n

Figures 7.9 and 7.10 present the costs of peer arrival and departure, respectively. They show that although the system size increases, the arrival costs and the departure costs are on average 18.61 and 27.58, respectively. According to the arrival and departure algorithms described in Section 7.2.1, the average arrival and departure costs are $2d + 2$ and $3d + c + 1$,

respectively (where c is the number of messages needed to find the ring neighbour that can receive the departing peer's key interval). With $d = 8.06$ (as observed in Fig. 7.6) and $c = 2.4$, the above arrival cost and departure cost formulae give 18.12 and 27.58, respectively, which are consistent with the experimental results.

As for the peer degree evaluation, we calculated the confidence intervals of the arrival cost and of the departure cost for all system sizes from 1 to 2100. The distance between the bounds found and the measured average arrival costs has max = 2.36, mean = 1.40, and median = 1.38. Similarly, the distance between the bounds and the measured average departure costs has max = 6.81, mean = 2.74, and median = 2.66. These numbers show that the confidence intervals of the arrival costs and of the departure costs are close.

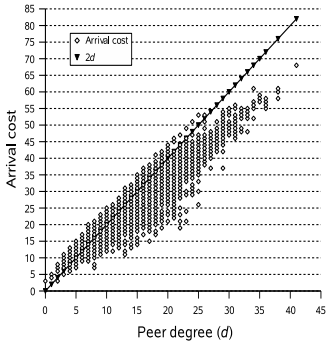

FIG. 7.11 – Arrival cost vs. peer degree d

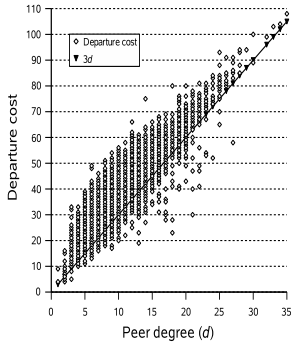

FIG. 7.12 – Departure cost vs. peer degree d

Figures 7.11 and 7.12 explore the relationships between the arrival/departure costs and the peer degree. Figure 7.11 plots the arrival costs corresponding to the degree (d) of the splitting peer. It shows that the arrival costs fall around $2d$. Similarly, Figure 7.12 plots the departure costs corresponding to the degree (d) of the departing peer. In this case, the departure costs cling to $3d$. The above observations suggest linear relationships between the arrival/departure costs and the peer degree. This explains the constant arrival/departure costs feature, which follows from the constant peer degree feature.

Figure 7.13 compares the average routing cost measured with $\log_2 n$. Although de Bruijn

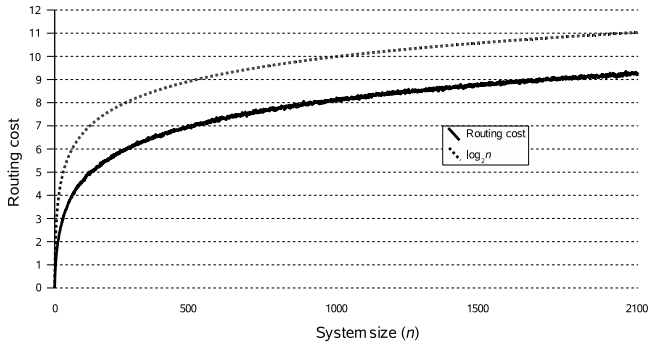

FIG. 7.13 – Routing cost vs. system size n

graph diameter m (being 32 here) is the upper bound of the routing cost, the graph shows that the average routing cost is under $\log_2 n$. With on average over 4100 routing requests finished at each system size from 1 to 2100 peers, we calculated the distance from the measured average routing costs to one limit of the confidence intervals. The observed distance having max = 0.13, mean = 0.09, and median = 0.09 shows very close confidence intervals. These results confirm the routing cost efficiency of the P2P topology.

7.3.2 Index management load balancing evaluation

The evaluation of index management load balancing explores how the global overload is influenced by the balancing algorithm. The metric for evaluating effectiveness is the index management overload ratio $\Omega = (\sum_{\forall p} O_p)/(\sum_{\forall p} T_p)$. To demonstrate the effectiveness of the load balancing, the experiments follow a 3-phase scenario. This scenario consists in running the simulation without load balancing for the first 30 simulation cycles (phase 1), then running with load balancing for the next 70 simulation cycles (phase 2), and finally running without load balancing for additional 30 simulation cycles (phase 3). When load balancing is active, the overloaded state is verified every simulation cycle.

As suggested by numerous research (e.g., [GFJ+03, LRS02, SGD+02, ZSZ03]) we ap-

ply the Zipf distribution[7] for the peers' capacity and the routing skewness. In particular, the Zipf exponent for the peers' index management capacity distribution is $\varepsilon = -1.2$. The routing source probability and the routing target probability[8] distributions are under the Zipf law with exponent $\varepsilon = -1.9$. Note that we have realized experiments with different Zipf exponents for the above distributions and also with a uniform distribution. However, they all gave results similar to those described below.

We consider three factors that can affect the load balancing result : the index management utilization ratio, the dynamicity of the routing target, and the dynamicity of the peers. The index management utilization ratio (U) is the ratio of the total index management load over the total capacity : $U = (\sum_{\forall p} T_p)/(\sum_{\forall p} C_p)$. The dynamicity of routing target (τ) denotes the change of routing target popularity in the system. It is the percentage of times a key changes its routing target probability at each simulation cycle (e.g., $\tau = 20\%$ means twenty changes within 100 cycles). The peer dynamicity (π) specifies the probability that a peer splits its keys with a joining peer and the probability that a peer departs in one simulation cycle. In these experiments, the arrival and departure probabilities are equal to maintain a relatively stable system size. Since we choose a certain U for each experiment to evaluate the Ω produced, the evolution of the system size is not required. Therefore, if a simulation cycle starts with n peers, it will have approximately $n\pi$ arrivals and $n\pi$ departures. The following experiments ran from a starting system size of 2048 peers. The graphs in this section show the average results of 20 experiments in each case.

Figure 7.14 plots the results of experiments in three cases with the same $\tau = 0\%$ and $\pi = 0\%$ but different U levels, which are, respectively, $[25\%, 30\%]$, $[55\%, 65\%]$, and $[100\%, 110\%]$. In these experiments, phase 1 (without load balancing) raises Ω to the maxi-

[7]The Zipf distribution defines that the i^{th} popular value is proportional to i^{ε} for some so called Zipf exponent ε.

[8]The probabilities for selection of, respectively, the originating peer and the destination key in lauching a simulated routing request.

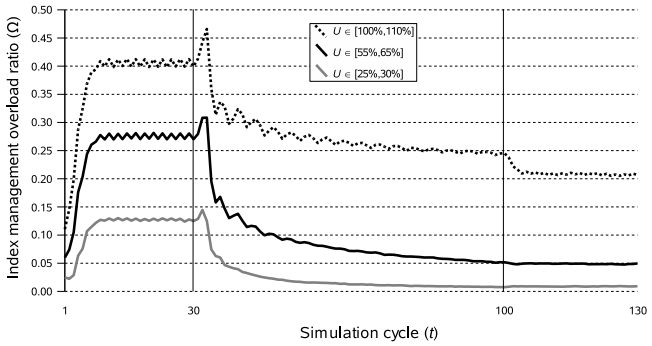

FIG. 7.14 – Index management overload ratio Ω vs. simulation cycle t for different index management utilization ratios U ($\tau = 0\%$, $\pi = 0\%$)

mum (e.g., 41.3% when $U \in [100\%, 110\%]$ or 12.9% when $U \in [25\%, 30\%]$). Phase 2 rapidly decreases Ω as a result of load balancing. In phase 3, when the load balancing is turned off, Ω remains stable at a low level (e.g., 20.6% when $U \in [100\%, 110\%]$ or 0.9% when $U \in [25\%, 30\%]$). These results demonstrate the effectiveness of the balancing algorithm. Once the load balance is established by phase 2, the global overload does not increase (in phase 3) even if the load balancing has stopped.

It is natural that higher values of U induce higher Ω. However, we compared the experiments under different U ratios to confirm that the load balancing method takes effect in a large range of capacity utilization. Even if the global load/capacity ratio falls within $[100\%, 110\%]$, the balancing method can reduce the global overload by half.

In the above graphs, there is a peak at the beginning of phase 2 and a fall at the beginning of phase 3. This follows from the difference in operation between phase 2 and phases 1 and 3. The load balancing in phase 2 requires every overloaded peer p to keep table G_p's value as long as key interval changes occur. However, phases 1 and 3 reset G_p every cycle. This makes Ω at the beginning and the end of phase 2 higher than that at, respectively, the end of phase 1 and the beginning of phase 3.

In order to observe the effect of routing target dynamicity on load balancing, we ran

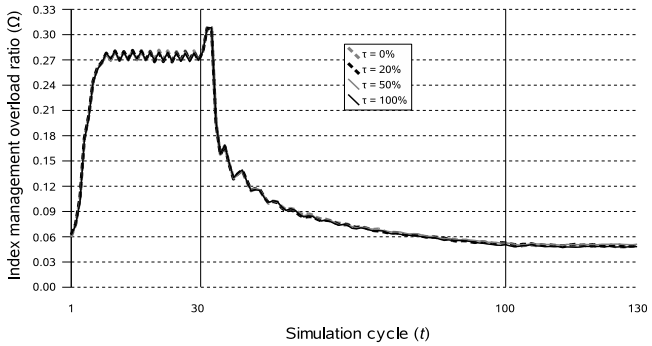

FIG. 7.15 – Index management overload ratio Ω vs. simulation cycle t for different routing target dynamicity factors τ ($U \in [55\%, 65\%]$, $\pi = 0\%$)

experiments at the same U and π but with different τ values. Figure 7.15 depicts the experimental results in four cases where $U \in [55\%, 65\%]\%$, $\pi = 0\%$, and τ, respectively, 0%, 20%, 50%, and 100%. In spite of the routing target dynamicity, which varies from 0% to 100%, the efficiency of load balancing is almost the same across the experiments. When a pairwise comparison of the results of Figure 7.15 is performed, the difference rates obtained are all 0%, which implies that we cannot recognize any difference between the graphs. This expresses a weak effect of routing target dynamicity on load balancing. This can be explained by the good randomization characteristics of the de Bruijn graph when transferring routing requests over de Bruijn nodes.

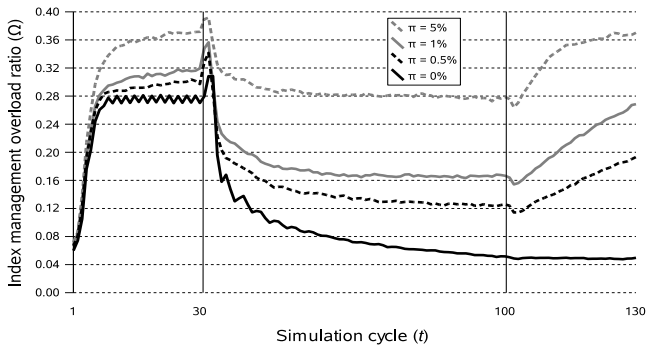

FIG. 7.16 – Index management overload ratio Ω vs. simulation cycle t for different peer dynamicity factors π ($U \in [55\%, 65\%]$, $\tau = 0\%$)

Figure 7.16 compares the results of experiments in four cases with $U \in [55\%, 65\%]$,

$\tau = 0\%$, and π values of 0%, 0.5%, 1%, and 5%, respectively. We need to explain why 5% is chosen as the highest π for experiments. The duration of a simulation cycle is the maximal time needed for the communication between any pair of peers. Suppose that the system runs in a slow environment, which requires up to 60 seconds for a pairwise communication. Thus, with $\pi = 5\%$ per cycle, 40% of peers arrive and 40% of peers depart every 8 minutes. In other words, each peer makes 144 arrivals and departures per day. A study on peer availability in the P2P file-sharing system Overnet [BSV03] observed that the rates of long-term peer arrivals and departures do not exceed 40% of peers per day. On the other hand, each peer joins and leaves on average 6.4 times per day. When comparing with these results, it is clear that $\pi = 5\%$ indicates a very high dynamicity of peers.

The load balancing method takes effect in all four cases. However, higher π generate higher Ω. When a new peer p joins the system, it needs some time to establish a connection with the neighbourhood. During this time, the neighbours that have not yet received its notification, direct routing messages to other peers instead. Peer p does not bear enough routing traffic as it should and thus we observe a little increase of the global overload at the end of its arrival.

We also see that in systems with higher peer dynamicity, Ω decreases more slowly in phase 2 and increases more rapidly in phase 3. This phenomenon results from the insertion and deletion of peers, which break the load balance state previously established. The active load balancing procedure in phase 2 keeps reducing Ω. In phase 3, Ω does not increase only if $\pi = 0\%$. Otherwise, Ω rises due to the lack of rebalancing.

7.3.3 Storage load balancing evaluation

The principal metric for evaluating the storage load balancing method is the storage overload ratio $\Psi = (\sum_{\forall p} W_p)/(\sum_{\forall p} S_p)$. The smaller Ψ is, the more storage load balance

it achieves. The evaluation involves multiple experiments following different scenarios. All experiments start with 2048 peers. We limited the peers' storage capacity and desired storage capacity to the range [100MB, 3.2GB]. Again, we selected the distribution of the desired storage capacities of the peers under the Zipf law with the exponent $\varepsilon = -1.2$. As suggested by different research (e.g., [Dow01b, DB99]), we apply the log-normal distribution[9] to the object sizes, which fall within the range [1MB, 100MB]. The chosen parameters $\mu = 2$ and $\sigma = 0.84$ yield an object size mean of 10.5MB and a median of 7.4MB. We also have performed experiments with different distribution parameters (both for the desired storage capacity and object size) and even with the uniform distribution. They all gave performance results similar to the experiments presented herein.

We observed the performance of the storage load balancing method subject to three factors : the storage utilization ratio, the peer dynamicity, and the storage load transfer strategy applied (cost-oriented or overload-oriented). The storage utilization ratio Z is the ratio of the total storage load over the total desired storage capacity, $Z = (\sum_{\forall p} S_p)/(\sum_{\forall p} \overline{D}_p)$. Peer dynamicity ($\pi$) has the same meaning as for index management load balancing evaluation (Sect. 7.3.2). By comparing the impact of the storage load transfer strategies, we can observe their strength and weakness relative to each other. Different combinations of factors produce numerous cases. The graphs in this section plot the average results for at least 20 experiments in each case.

The experiments presented in Figure 7.17 continuously insert objects in the system to increase the utilization ratio Z. The graph plots the overload ratio Ψ corresponding to the increase of Z (from 1% to 150%). In order to confirm the effectiveness of load balancing, we compared the results under two modes : active load balancing and inactive load balancing. We also compared the impact of three peer dynamicity levels : $\pi = 0\%$, $\pi = 1\%$, and $\pi = 5\%$.

[9]The log-normal distribution of a variable x defines its density probability as $P(x) = e^{-(\ln x - \mu)^2/(2\sigma^2)}(x\sigma\sqrt{2\pi})^{-1}$ with given μ and σ parameters, which are respectively mean and standard deviation of $\log x$.

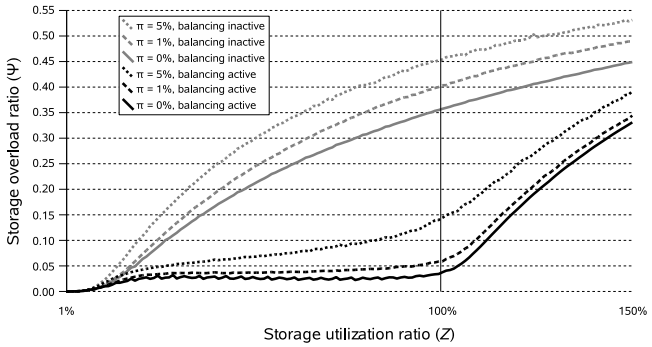

FIG. 7.17 – Storage overload ratio Ψ vs. storage utilization ratio Z for different peer dynamicity factors π (with cost-oriented balancing or without balancing)

These experiments apply the cost-oriented storage load transfer strategy. The graph shows that without load balancing, Ψ increases rapidly along with the increase of Z. However, with load balancing, the increase of Ψ is drastically slowed down as long as $Z \leq 100\%$. It rises very fast only when Z exceeds 100% (i.e., the system lacks available space to rearrange objects). We see that even with a high dynamicity ($\pi = 5\%$), the storage load balancing method is effective. Of course, higher dynamicity causes higher overload and reduces the storage load balancing effectiveness.

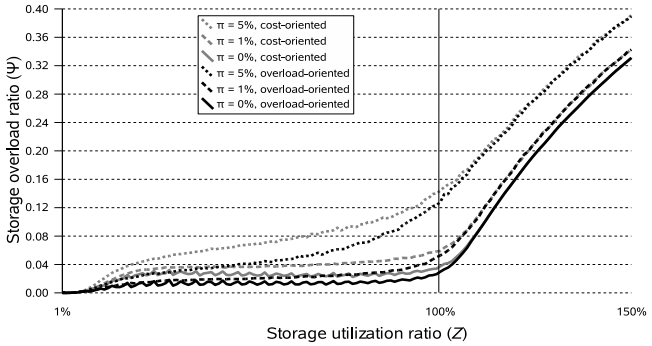

FIG. 7.18 – Storage overload ratio Ψ vs. storage utilization ratio Z for different peer dynamicity factors π (with cost-oriented balancing or overload-oriented balancing)

Figure 7.18 compares the effectiveness of the two storage load transfer strategies : cost-oriented and overload-oriented. The experiments were run with load balancing and with peer dynamicities π of 0%, 1%, and 5%, respectively. In all experiments, when π and $Z \leq 110\%$

are the same, the balancing with overload-oriented transfers produces lower Ψ than cost-oriented transfers. Overload-oriented transfers achieve higher effectiveness than cost-oriented transfers since they have more possibilities to reduce the combined overload. For $Z > 110\%$, the effectiveness of the two transfer strategies becomes the same since the possibility to rearrange objects is very small in this case.

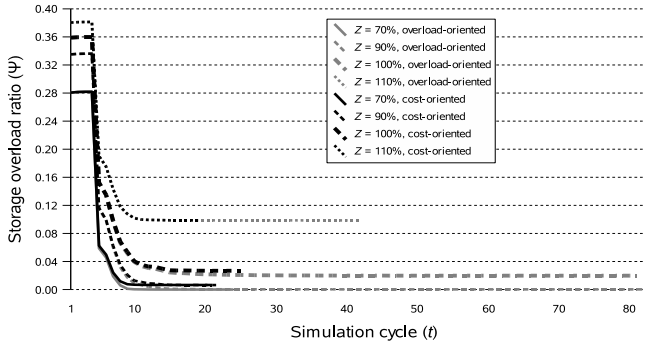

FIG. 7.19 – Storage overload ratio Ψ vs. simulation cycle t for different storage utilization ratios Z (with cost-oriented balancing or overload-oriented balancing ; $\pi = 0\%$)

In order to evaluate the convergence of the storage load balancing method, we executed experiments at a specific utilization ratio Z. The experiments ran with load balancing (using cost-oriented or overload-oriented strategy), $\pi = 0\%$, and without object insertion. This ensures that neither peer dynamicity nor storage load changes will affect the storage load balancing while verifying its convergence. The experiments stop when Ψ reaches 0 or its minimal value. Figure 7.19 shows the results of experiments with utilization ratios 70%, 90%, 100%, and 110%, respectively. In all experiments, the overload ratio Ψ decreases rapidly after a small time interval (being equal to the duration of a period of the overload verification on each peer). These results confirm that the load balancing method converges. We refer to the minimum value of Ψ as the stable storage overload ratio (denoted Ψ_f). Higher values of Z induce higher initial storage overload ratio and higher stable storage overload ratio. This is expected since a high Z corresponds to small available space and consequently to small possibilities of global overload reduction.

In all experiments using the same Z, the load balancing with overload-oriented transfers yields smaller Ψ_f than that for cost-oriented transfers. However, the time to achieve the stable state (called the stabilization time, denoted t_f) of the overload-oriented transfers is longer than that of the cost-oriented transfers. We find the details of these differences in Figures 7.20 and 7.21, which respectively compare the stable overload ratio and the stabilization time of the two transfer strategies. These figures additionally present the observation for $Z = 95\%$ and $Z = 105\%$. The differences follows from the fact that an overload-oriented storage load transfer among two peers has more chances to take place than a cost-oriented storage load transfer. Therefore, the load balancing that applies overload-oriented transfers involves more transfers. This results in a longer stabilization time but a better stable overload ratio.

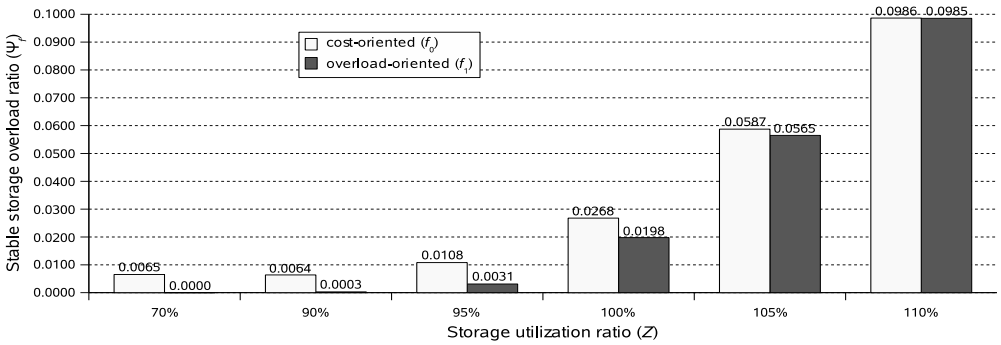

FIG. 7.20 – Stable storage overload ratio Ψ_f vs. storage utilisation ratio Z for cost-oriented and overload-oriented transfer strategies

The graph in Figure 7.21 shows that experiments at $Z = 100\%$ have the longest stabilization time. It is reduced when $Z < 100\%$ or $Z > 100\%$. When $Z > 100\%$, the lack of available space diminishes the chance for storage load transfers to take place. Therefore, the stable state is near the initial state. However, when $Z < 100\%$, the profusion of available space accelerates storage load transfers in reducing the global overload. It thus shortens the stabilization.

Let Ψ_{f_0} and Ψ_{f_1} be Ψ_f produced by the cost-oriented and overload-oriented transfer strategies, respectively. We statistically compared Ψ_{f_0} and Ψ_{f_1} under the given values of Z.

Z	Confidence interval of $\Delta\Psi_f = \Psi_{f_0} - \Psi_{f_1}$	Z	Confidence interval of $\Delta\Psi_f = \Psi_{f_0} - \Psi_{f_1}$
70%	$[0.0064, 0.0067]$	100%	$[0.0065, 0.0075]$
90%	$[0.0059, 0.0063]$	105%	$[0.0016, 0.0029]$
95%	$[0.0072, 0.0082]$	110%	$[-0.0003, 0.0005]$

TAB. 7.I – Comparison of stable storage overload ratio Ψ_f for cost-oriented (f_0) and overload-oriented (f_1) transfer strategies ($\alpha = 1\%$)

Table 7.I shows the confidence interval of $\Delta\Psi_f = \Psi_{f_0} - \Psi_{f_1}$. Except for the case $Z = 110\%$, the confidence interval of $\Delta\Psi_f$ does not include 0, which indicates a statistical difference between Ψ_{f_0} and Ψ_{f_1} and thus a considerable effect of the two transfer strategies on Ψ_f. At $Z = 110\%$, the confidence interval of $\Delta\Psi_f$ includes 0. This indicates a negligible difference of Ψ_f produced by the two transfer strategies. As explained before, in this case, the lack of available space prevents the possibility to arrange objects.

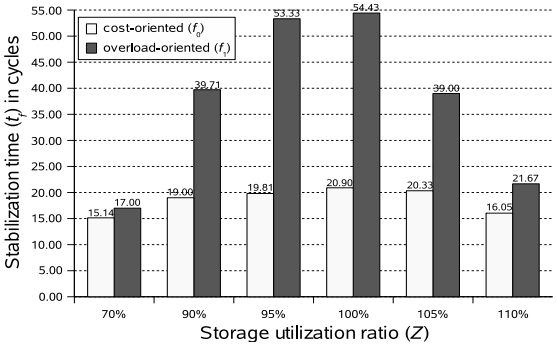

FIG. 7.21 – Stabilization time t_f vs. storage utilisation ratio Z for cost-oriented and overload-oriented transfer strategies

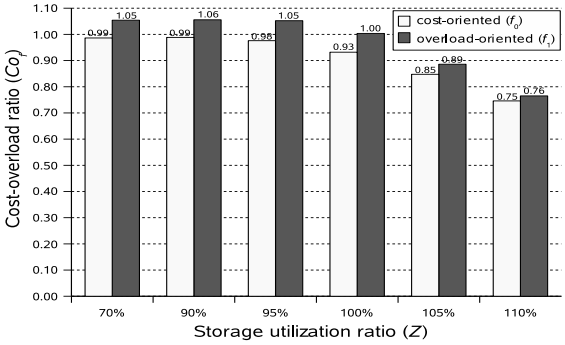

FIG. 7.22 – Cost-overload ratio Co_f vs. storage utilisation ratio Z for cost-oriented and overload-oriented transfer strategies

Similarly, we denoted the stabilization time obtained by the cost-oriented and overload-oriented transfer trategies as t_{f_0} and t_{f_1}, respectively. A statistical comparison of t_{f_0} and t_{f_1} allows us to evaluate the effect of the transfer strategies on the stabilization time. Table 7.II shows the confidence intervals of $\Delta t_f = t_{f_0} - t_{f_1}$.

Except for the case $Z = 70\%$, t_{f_0} and t_{f_1} are statistically different. This implies that the strategy has a considerable effect on the stabilization time. At $Z = 70\%$, although the

Z	Confidence interval of $\Delta t_f = t_{f_0} - t_{f_1}$	Z	Confidence interval of $\Delta t_f = t_{f_0} - t_{f_1}$
70%	$[-3.83, 0.12]$	100%	$[-38.63, -28.42]$
90%	$[-30.92, -10.51]$	105%	$[-24.01, -13.32]$
95%	$[-40.92, -26.13]$	110%	$[-8.90, -2.34]$

TAB. 7.II – Comparison of stabilization time t_f for cost-oriented (f_0) and overload-oriented (f_1) transfer strategies ($\alpha = 1\%$)

mean of t_{f_0} is a bit lower than the mean of t_{f_1}, the confidence interval of their difference covers 0. The difference between t_{f_0} and t_{f_1} in this case is not clear. The explanation of this phenomenon comes from the profusion of available space that diminishes the number of 2-direction storage load transfers in the overload-oriented transfer mode and thereby induces a weak effect of the chosen mode on the stabilization time.

Our evaluation also examined the cost of balancing, that is, the volume of data for migration. We use the experiments in the above convergence-evaluation scenario for this evaluation. The metric is the cost-overload ratio Co_f being the ratio of the cumulative storage load sent (until the stable state) over the initial global overload. Figure 7.22 shows the average cost-overload ratio for the experiments. At the same Z, an experiment with cost-oriented transfers always has the cost-overload ratio under 1 and lower than the cost-overload ratio of an experiment with overload-oriented transfers. The reason is that a cost-oriented transfer reduces the combined overload the most while preventing the cost to exceed the gain. On the other hand, an overload-oriented transfer minimizes the combined overload before minimizing the migration cost. The graph also shows that the higher Z is, the less is the cost-overload ratio. This results from the fact that smaller available space yields smaller storage load movements. Therefore, migration costs for stabilization are smaller.

We also statistically compared the cost-overload ratio across the two transfer strategies. Let Co_{f_0} and Co_{f_1} denote the cost-overload ratio under the cost-oriented and the overload-oriented transfer modes, respectively. Table 7.III lists the confidence interval of $\Delta Co_f =$

$Co_{f_0} - Co_{f_1}$ at different utilization ratios Z. No interval includes 0. This shows a significant effect of the transfer trategies on the cost-overload ratio.

Z	Confidence interval of $\Delta Co_f = Co_{f_0} - Co_{f_1}$	Z	Confidence interval of $\Delta Co_f = Co_{f_0} - Co_{f_1}$
70%	$[-0.0695, -0.0660]$	100%	$[-0.0740, -0.0694]$
90%	$[-0.0690, -0.0652]$	105%	$[-0.0417, -0.0354]$
95%	$[-0.0791, -0.0731]$	110%	$[-0.0208, -0.0172]$

TAB. 7.III – Comparison of cost-overload ratio Co_f for cost-oriented (f_0) and overload-oriented (f_1) transfer strategies ($\alpha = 1\%$)

7.3.4 Integrated load balancing

To this point, index management load balancing and storage load balancing methods have been evaluated separately. One can question whether these load balancing methods preserve their performance when operating concurrently. In other words, are index management load balancing and storage load balancing affecting each other ? We designed experiments to assess this influence.

The experiments ran on a system of 2048 peers with peer dynamicity $\pi = 0\%$ and routing target dynamicity $\tau = 0\%$. The distribution of the desired storage capacity, the object size, the index management capacity, the probability of routing source, and the probability of routing target are the same as the experiments presented in Sections 7.3.2 and 7.3.3. The metrics for the load balancing performance are again the index management overload ratio Ω and the storage overload ratio Ψ. These metrics are verified in the following cases :

– (00) inactive index management load balancing, inactive storage load balancing,

– (01) inactive index management load balancing, active storage load balancing,

– (10) active index management load balancing, inactive storage load balancing,

– (11) active index management load balancing, active storage load balancing.

In those cases where the storage load balancing is active, we employ the cost-oriented

transfer strategy. To confirm the performance of the index management load balancing in a large range of utilization ratio, we realized experiments at two levels : $U \in [55\%, 65\%]$ and $U \in [100\%, 110\%]$. The graphs in this section plot the average results for at least 20 experiments within each case.

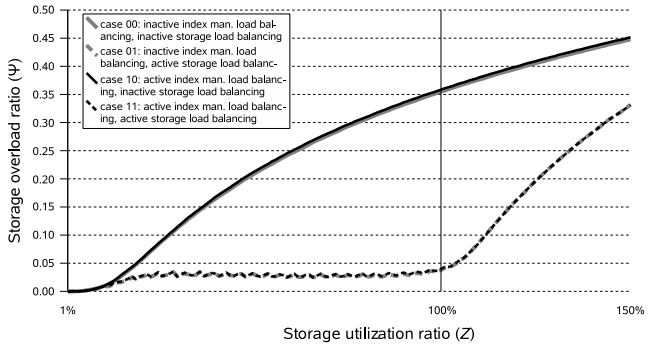

FIG. 7.23 – Storage overload ratio Ψ vs. storage utilization ratio Z for different balancing cases ($U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

Figure 7.23 compares the increase of the storage overload ratio Ψ (along with the increase of the storage utilization ratio Z from 1% to 150%) in the four cases. The index management utilization ratio U is set to the range $[55\%, 65\%]$. The increase of Ψ in the two cases 00 and 10 is similar. In the two other cases, the presence of the storage load balancing prevents the increase of Ψ and gives the same results regardless whether the index management load balancing is active or not.

We performed statistical comparisons (as described at the beginning of Sect. 7.3) between the graphs of cases 00 and 10, and between those of cases 01 and 11. The difference rate obtained from the former comparison is 78.00% and from the latter it is 24.67%. These rates show a statistical difference in each compared graph pair. However, the difference is very small. In each comparison between two graphs X and Y, the average difference $|X_i - Y_i|$ obtained never exceeds 1% of the graphs' scale (for all values i on the horizontal axis). The scale of a graph here denotes the graph's maximal value.

The above differences result from the measurement method. Index management load

balancing consists in transferring key intervals among peers. A transfer not only moves the key interval but also the replica diffusion processes of the objets involved in the key interval moved (see Sect 7.2.3). To ensure the integrity of the decentralized operation, the replica diffusion processes on the source peer are not stopped until they receive an acceptance message from the destination peer. Therefore, between the times of sending a proposal and receiving an acceptance, replica diffusion processes (for one object) may function concurrently on both peers. This situation accelerates object insertion. Because the storage utilization ratio considered is a real number, we cannot measure the overload corresponding to exactly one value point on the horizontal axis. However, we determined it by calculating the average of the measured results for each interval of length equal to 1% on the storage utilization ratio axis. Therefore, the acceleration of object insertion can affect the results measured with some probability. The small difference given by the statistical analyses reflects the effect of the object insertion acceleration on the measurement. We cannot, however, conclude that there is an impact of the index management load balancing on the effectiveness of the storage load balancing.

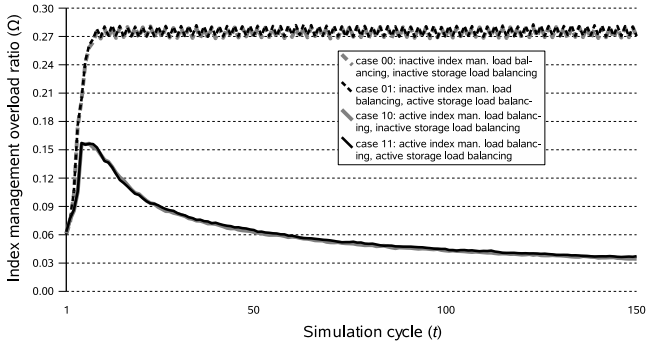

FIG. 7.24 – Index management overload ratio Ω vs. simulation cycle t for different balancing cases ($U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

In Figure 7.24, we compare the index management overload ratio Ω (for 150 simulation cycles) in the four cases. The utilization ratio U is chosen in the range $[55\%, 65\%]$. The variation of Ω is the same among the cases 00 and 01, and among the cases 10 and 11. We

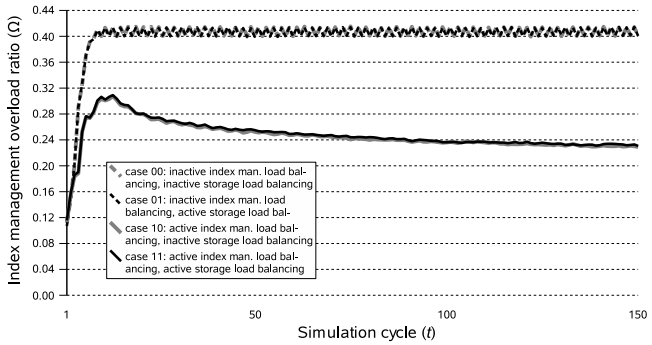

FIG. 7.25 – Index management overload ratio Ω vs. simulation cycle t for different balancing cases ($U \in [100\%, 110\%]$, $\pi = 0\%$, $\tau = 0\%$)

see that the presence of the storage load balancing does not degrade the result of the index management load balancing. Figure 7.25 shows similar experiments but at utilization ratio $U \in [100\%, 110\%]$. We observe similar results except that the Ω values obtained are higher. Statistical comparisons between the graphs of cases 00 and 01, and between the graphs of cases 10 and 11 (in both Figures 7.24 and 7.25) were also performed. The comparisons of graphs in Figure 7.24 give difference rates as low as 0.67% and 0.00%, respectively. The corresponding rates for the graphs in Figure 7.25 are 0.00% and 0.00%. The results demonstrate that the storage load balancing does not influence the effectiveness of the index management load balancing.

7.4 Validation

The performance of the proposed P2P topology and the load balancing methods have been evaluated. In this section, we validate the above results by demonstrating the advantages of the technical solutions chosen. We consider two points : the separation between the object location and the key, and index management load balancing based on key transfer. The demonstration compares our approach with some related work. We have implemented simulators inspired by these related models to run experiments for comparison purposes. These

simulators only implement the elements and functions required to perform comparisons with our approach. They therefore maximize simplicity to preserve efficiency and to guaranty conformance to the methods' specifications.

7.4.1 Separation vs. attachment of the storage location and the key

To validate the separation between the storage location and the key, we investigate how the dependency of the object location on the key influences the integrated index management load and storage load balancing. An example of the attachement of the object location to the key is the application of virtual servers in the storage load balancing. According to this approach, each physical peer maintains multiple virtual servers, which operate as individual peers in a normal system. The peers can achieve load balancing by arranging the virtual servers' residence. Because a virtual server represents a virtual peer, it is responsible for a set of keys. In the application of virtual servers for storage load balancing, the objects' location must be tied to their key.

In this discussion, we do not compare BALLS with the Virtual Servers system [RLS+03] in the general context as it was introduced. We simply evaluate the ability to apply the notion of virtual servers for simultaneously supporting the storage load balancing and the index management load balancing. Thus, we compare with the application of virtual servers for integrated load balancing. We developed a P2P simulator that creates virtual servers on top of physical peers. In simulations, it lets each virtual server manage one key, the smallest size of a virtual server. The simulator implements an index management load balancing similar to ours. The support of this index management load balancing adds a constraint to the virtual servers : each physical peer manages only numerically adjacent virtual servers, i.e., virtual servers on a peer compose a continuous key interval. Therefore, the transfer of virtual servers is limited among ring neighbours.

The storage load balancing is based on transferring virtual servers (the virtual servers carry all objects that belong to them). The simulator implements only the one-to-one transfer. That is, when a peer discovers the overload, it transfers an appropriate set of virtual servers to a ring neighbour for achieving the balancing goal. The use of this straightforward operation ensures the simulator's simplicity. The calculation of the global overloads, loads, and capacities simply aggregate the overloads, loads, and capacities, respectively, of all the peers, which are measured in a similar way as BALLS, thereby allowing to compare the simulation results.

The experiments ran in four cases (like the experiments presented in Sect. 7.3.4) :

– (00) inactive index management load balancing, inactive storage load balancing,

– (01) inactive index management load balancing, active storage load balancing,

– (10) active index management load balancing, inactive storage load balancing,

– (11) active index management load balancing, active storage load balancing.

The system comprises 2048 physical peers with dynamicity $\pi = 0\%$ and routing target dynamicity $\tau = 0\%$. The parameter settings for the peer's capacities, object size, and routing skewness are the same as those used for the experiments described in Section 7.3.4. The index management utilization ratio U is set in the range $[55\%, 65\%]$. The storage utilization ratio Z increases along with object insertions. To enable the comparison, we used the storage overload ratio Ψ and the index management overload ratio Ω as metrics to measure the effectiveness of the storage load balancing and the index management load balancing, respectively. The following graphs depict the average results of at least 20 experiments in each case.

Figure 7.26 compares the storage overload ratio Ψ (for the storage utilization ratio Z increasing from 1% to 150%) in the four cases. The storage load balancing still takes effect. However, the presence of the index management load balancing produces a higher Ψ. We have realized statistical comparisons between the graphs of cases 00 and 10, and between the

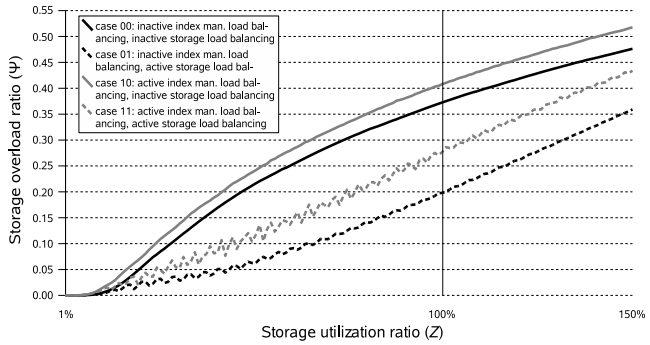

FIG. 7.26 – Storage overload ratio Ψ vs. storage utilization ratio Z for different balancing cases using the virtual servers approach ($U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

graphs of cases 01 and 11, which give difference rates as high as 96.67%. These results show a considerable effect of the index management load balancing on the storage load distribution. The transfer of key intervals in the index management load balancing involves transferring the corresponding objects as well. It therefore breaks the current distribution of the storage load. The index management load balancing considerably reduces the effectiveness of the storage load balancing if it is active.

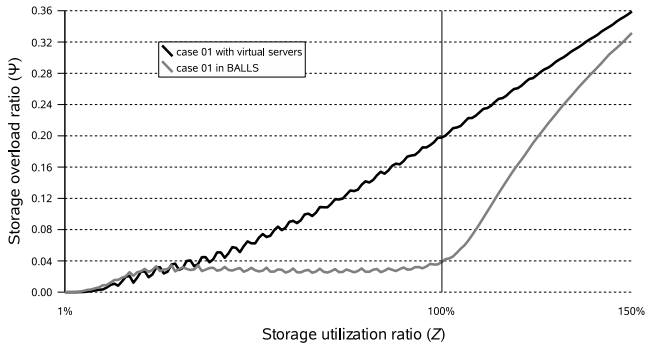

FIG. 7.27 – Storage overload ratio Ψ vs. storage utilization ratio Z for the virtual servers approach and BALLS (case 01, storage balancing only ; $U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

We also compared the result of the storage load balancing (without index management load balancing) to that of our system (Fig. 7.27). Our approach produces a much better result even when using the cost-oriented strategy. This difference comes from the fact that we balance the storage load at the object level while the arragement of virtual servers does it at

the key level. The manipulation at the key level is less flexible because all objects belonging to a key must move together. However, if we arrange the storage load at the object level, we can select individual objects to move. Moreover, better performances are obtained because our peers can freely choose objects and destination peers to transfer storage load without worrying about the maintenance of the peer connection structure (as in moving keys).

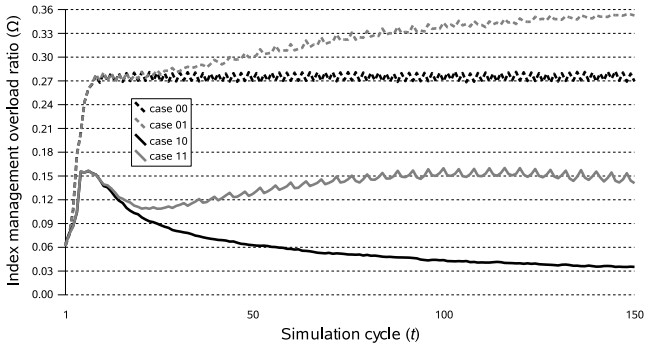

FIG. 7.28 – Index management overload ratio Ω vs. simulation cycle t for different balancing cases using the virtual server approach ($U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

We also investigated the impact of the storage load balancing on the index management load balancing. Figure 7.28 compares the index management overload ratio Ω (for 150 simulation cycles) in the four cases. Again, the index management load balancing takes effect. However, it is considerably affected by the simultaneous storage load balancing. Statistical comparisons between the graphs of cases 00 and 01, and between the graphs of cases 10 and 11, reinforce this observation. We indeed obtain high difference rates at 83.33% and 89.33%, respectively. Moving virtual servers in the storage load balancing breaks the current distribution of the index management load on the peers and often produces higher Ω. If the index management load balancing operates, its effectiveness is greatly reduced.

The comparison of the above experiments with the corresponding experiments on our system (Sect. 7.3.4) exposes two major disadvantages of associating the object and key locations :

– the index management load balancing and the storage load balancing are no longer in-

dependent. They reduce the performance of each other when being executed concurrently ;

– constraining object location to the key location prevents the choice of objects and destination peers in storage load migration. This constraint greatly reduces the effectiveness of the storage load balancing even without the intervention of the index management load balancing.

7.4.2 Disadvantage of restricting object location

As discussed in Section 7.1.2, some systems that break the tie between object storage and root still maintain a restriction on object placement. This introduces considerable network overhead when they change. We validate this discussion by evaluating this type of overhead in PAST [RD01b], which is a representative example of such systems. PAST uses a replica diversion technique so as to balance the storage load. This technique allows a file (PAST uses files as storage objects) to stay on one of the peers in the leaf set of its root instead of being attached to the root. Because of this relaxation, PAST reduces file insertion failures and thus improves storage capability. The replica diversion of PAST does not really separate the file location from the file's root. PAST maintains the following invariant : a file is stored inside the leaf set of the root. Suppose that l is the leaf set size. The arrival of a peer changes l peers' leaf set boundaries. It thus pushes some files out of their roots' leaf sets. To enforce the above invariant, PAST moves these files to the corresponding leaf sets. This invariant thus introduces additional maintenance costs.

We evaluated the file migration costs caused by peer arrival in such a system through experiments. To that end, we developed a simulator that constructs a simple P2P system, in which each peer p maintains a leaf set containing the contacts of l peers having ids numerically closest to p's id. Each file is replicated and assigned to κ roots being the peers with ids

numerically closest to the m-bit prefix of the file's id (m is the length of the peer id). The location of a replica is limited to its root and the leaf set of the root. The maintenance of this invariant against the system's evolution is straightforward. It discovers the files that are out of their limit and moves them to a valid host. The measurement of the migration costs incurred by a peer arrival consists in adding the sizes of all replicas moved to maintain the above invariant. To ensure correctness and efficiency of the measurements, this simulator performs calculations in a centralized manner.

The simulation sets the peer population to 2250, the peer id length (m) to 128 bits, and the number of replicas of a file (κ) to 5. The distribution of the peers' storage capacity is the same as that in experiments described by [RD01b] : normal distribution with $\sigma = 10.8$, $\mu = 27$MB, lower bound = 2MB, and upper bound = 51MB. Because we cannot obtain the set of files used in [RD01b], we generated file sizes using a log-normal distribution with min = 1KB, max = 10MB, median = 500KB, and mean = 1MB. In the context of experiments that verify the migration costs of PAST's peer arrivals, we do not use the experimental results presented by [RD01b]. Thus, the application of its exact settings is not necessary.

The experiments let peers arrive and depart with the same probability to maintain a relatively stable number of peers and storage capacity. We consider two metrics : (1) the load move ratio that is the ratio of the storage load moved (caused by one peer arrival) over the total storage load, and (2) the file move ratio that is the ratio of the number of moved files (caused by one peer arrival) over the total number of files. The experiments measure the cumulative load move ratio Λ (i.e., the sum of all load move ratios for all arrivals) and the cumulative file move ratio Φ (i.e., the sum of all file move ratios for all arrivals) for 500 arrivals. We obtained results at the following utilization ratio (i.e., the ratio of the total load over the total capacity) Z : 10%, 30%, 50%, 70%, and 90%. Two leaf set sizes (16 and 32) have also been tried. The graphs in Figures 7.29 and 7.30 plot the average results for at least 20 experiments in each case.

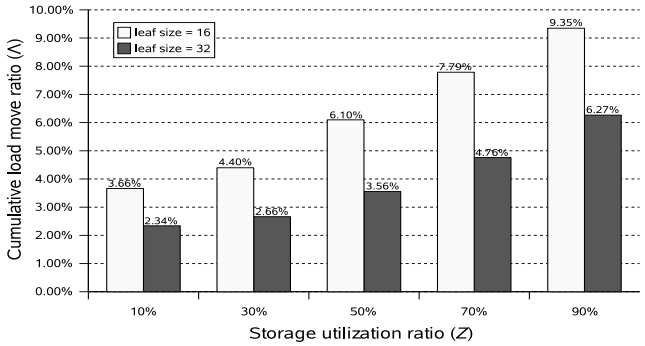

FIG. 7.29 – Cumulative load move ratio Λ vs. storage utilization ratio Z for 500 arrivals in PAST

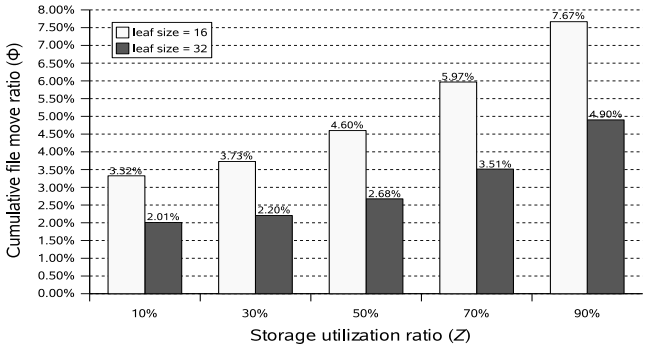

FIG. 7.30 – Cumulative file move ratio Φ vs. storage utilization ratio Z for 500 arrivals in PAST

A smaller leaf size leads to higher Λ and Φ. This is because the smaller leaf size increases the probability that a file is out of the root's leaf set when a peer arrives. We see that higher utilization ratios tend to produce higher migration costs. In all cases, the migration cost is considerable in comparing with the total load and the total number of files (Λ ranges from 2.34% to 9.35% and Φ ranges from 2.01% to 7.67%).

In order to reduce migration costs, PAST must use an additional caching mechanism. Note that in BALLS, peer arrival never incurs object migration costs as it completely separates object location and object key. In fact, it only requires a number of root notification messages equivalent to the number of objects whose keys fall within the key interval moved during peer arrival. In storage systems where object sizes are large enough, the communication costs incurred by the root notification messages are very small compared to the migration

costs.

In a highly loaded network, adding a peer (with low storage load) usually attracts objects from other peers to reduce their overload. We do not measure the cost of this tranfer as the migration cost for peer arrival since it does not serve any structure invariant maintenance. It is measured, however, in the migration cost for load balancing.

Note that the difference in the load balancing goal between PAST and BALLS does not allow us to compare them on their effectiveness. While PAST aims at maximizing the use of free space to decrease file insertion failures, BALLS minimizes the global overload based on the desired storage capacity of the peers.

7.4.3 Validation of the index management load balancing method

Since each peer in our system maintains an interval of de Bruijn nodes (i.e., keys) and the routing algorithm follows routing paths in the de Bruijn graph, transferring key intervals among peers allows to adjust the distribution of the index management load towards a balanced state. The use of the de Bruijn graph ensures a low peer degree (near 8) regardless of system size. The construction of the network makes the connection between peers symmetric. It means that if a peer p has a link to a peer q, then peer q is linked to peer p. These features enable low-cost maintenance procedures and thus efficient index management load balancing.

Expressways [ZSZ02] also considers index management load on the peers. The load balancing method of Expressways is based on the reorganization of links between the peers. As discussed in Section 7.1.2, this method yields a high peer degree. We evaluate this feature by experiments.

Expressways extends the idea of CAN [RFH$^+$01] by managing a hierarchy of zone spans

in which the leaves are basic CAN zones and each parent (internal) zone covers all its child zones. Because of the hierarchical management, the links of internal zones (also called expressway zones) tend to incur more routing traffic than those of their child zones. The Expressways load balancing method promotes peers with higher capacities to higher expressway levels so as to equalize the peers' load/capacity ratio.

The above load balancing goal is different from that of BALLS, which minimizes the global overload when the overload occurs. Again, this difference does not allow us to use the balancing effectiveness as the comparison criterion. However, we can compare BALLS and Expressways on the costs of the load balancing methods. In particular, these costs are function of the peer degree in each system.

The index management load balancing method of Expressways can only be applied in a P2P structure that manages a hierarchy of zone spans. In a basic CAN system, the peer degree is constant. In Expressways, multiple zone levels require multiple link levels. If a system has n peers, it requires a hierarchy of O$(\log n)$ zone levels and thus a peer degree in O$(\log n)$. We evaluated this assessment through experiments.

The simulator implements a simple prototype of the Expressways system, which is based on hierarchical management of zone spans. Each peer maintains the links to its neighbouring zone spans at all levels of the hierarchy. The peer arrival and departure consist of, respectively, splitting and merging zone spans and updating the links of peers to maintain the Expressways topology. A straightforward measurement of peer degree that counts the number of links at all levels on each peer enables us to obtain comparable results.

Our Expressways simulation sets the number of dimensions at 2, the coverage at 4, and the largest zone span at the entire Cartesian space. An experiment starts with 1 peer with an arrival probability higher than the departure probability. It stops when the system reaches 2100 peers.

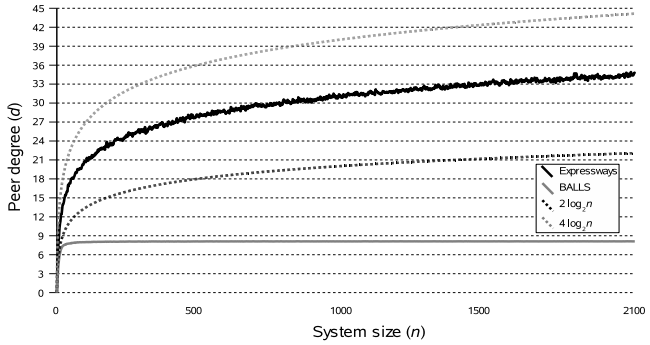

FIG. 7.31 – Peer degree d vs. system size n in Expressways

Figure 7.31 plots the average result for 40 experiments. It compares the average peer degree of Expresssways with that of BALLS. The experiments show that the measured average peer degree of Expressways falls within $2\log_2 n$ and $4\log_2 n$ (where n is the current system size). Confidence intervals were calculated for the average peer degree at all system sizes from 1 to 2100. The distance between the boundaries of these intervals and the measured average peer degree has max = 0.55, mean = 0.38, and median = 0.38. These numbers show very close confidence intervals. We see that the peer degree of Expressways is considerably higher than the constant peer degree (near 8) measured in BALLS. Thus the connection structure of Expressways tends to induce a higher complexity for maintenance.

In the current implementation of our system, we send a root notification message for every object with the same key. However, a simple improvement could greatly reduce this cost. We can notify all objects stored in the same peer about their new root using a single message, hence reducing the number of messages required.

7.5 Discussion

We have proposed and evaluated the performance of the BALLS topology and integrated load balancing methods. We have also shown the advantages of these methods by comparing

them with some related methods. From the results obtained, we have the following comments.

Recall that our work aims at two goals : (1) simultaneously supporting index management load balancing and storage load balancing, and (2) providing low maintenance costs. We now review the fulfilment of these requirements.

As described, simultaneous index management load balancing and storage load balancing is achieved by separating object location and key location. It allows the storage load balancing algorithm to arrange objects regardless of the key distribution. Therefore, the storage load balancing and the index management load balancing do not affect each other. The evaluation of integrated load balancing in BALLS and in the application of virtual servers (Sect. 7.3.4 and 7.4.1) confirmed this conclusion. The experiments using BALLS, where the key and object locations are independent, showed that the effectiveness of the index management load balancing is not affected by the storage load balancing and vice versa. However, if the system ties the object location to the key location (directly or indirectly), index management and storage load balancing are no longer independent, reducing the effectiveness of each other.

We also have seen that storage load balancing in a system having independent object and key locations is more effective than in a system without this independence. If the objects are tied to their key, each storage load move involves moving at least one key and all objects belonging to the key. Moreover, to maintain the P2P connection structure, only a certain set of keys on a peer can be selected for moving and only a certain set of neighbouring peers can receive the transferred keys. For example, in experiments with virtual servers in Section 7.4.1, a peer p can only move key intervals at two ends of $[p.b, p.e]$ to its ring neighbours. However, if we separate the object and key locations, the peers are free to choose objects and destination peers for the move. The peers have more possibilities to make storage load moves and thus induce a higher effectiveness of storage load balancing.

Let us now consider the cost of maintenance, which is usually defined as the bandwidth spent to maintain the P2P structure consistency against the network dynamicity. In BALLS, the dynamicity that affects the P2P structure is made of the peer arrivals/departures and the index management load balancing, which involve a key interval transfer.

In a traditional system where the object location is tied to the key, the transfer of a key always requires to move the objects belonging to it. The key interval transfer cost thus includes the object migration cost. In BALLS, a key interval transfer never needs to move objects. Instead, it only moves the associated storage pointers and launches the necessary root notification messages. In general, an object whose key falls within the transferred interval needs one pointer to move and one notification message to be sent. If we assume that the size of a pointer and the size of a notification message are much smaller than the size of an object, BALLS greatly saves on the transfer cost.

Although small, notification messages induce communication costs for object management in BALLS. These costs depend on the distribution of objects over the key space. The evaluation of these costs is beyond the scope of this paper.

We noticed in Section 7.4.2 that PAST relaxes the storage constraint by using redirection pointers and the replica diversion technique. Yet, its peer arrival produces a considerable migration cost. This follows from the fact that the file location and the key are not really separated.

When considering peer departure, the separation of object location and key location also have advantages. Since it allows object replication, the departure of a peer needs only to copy the objects that are not replicated elsewhere. Replicas of the other objects can be re-established later. This saves on the time of the departure procedure. Replication is also useful in storage load balancing. It distributes the access of an object on multiple peers so as to minimize bandwidth for “hot” content.

The maintenance of object availability is a common problem of P2P systems. Indeed, we should ensure that an object will still exist even when the peer storing it leaves the network. The difficulty in availability maintenance increases if more peers depart than peers arrive, i.e., the global storage capacity decreases. It also depends on the selection of storage capacity of the peers. This paper does not address this problem.

BALLS also achieves low maintenance costs by virtue of its low peer degree. A low peer degree simplifies the maintenance against the change of key responsibility. Experiments in Section 7.3.1 showed that BALLS's average peer degree is near 8, regardless of system size. This peer degree is considerably small in comparison to logarithmic peer degree systems such as Chord [SMK+01] and Tapestry [ZHS+04].

We have compared the peer degree of BALLS with another P2P system that also supports index management load balancing, Expressways, in Section 7.4.3. The experiments showed that the average peer degree of Expressways falls within $2\log_2 n$ and $4\log_2 n$ (at an n-peer system size). This degree is obviously higher than the constant peer degree of BALLS.

Finally, unlike other load balancing methods that aim at globally equalizing the load or the load/capacity ratio of the peers, our methods minimize the global overload (for both index management load and storage load). In practice, as long as the capacity of a peer covers the load, the peer can work properly and does not need any operation to reduce its load. Therefore, a load balancing action in this situation is not necessary. A peer provokes the load balancing operation only when its load exceeds the capacity and generates a positive overload.

The minimization of the global overload rearranges load from overloaded peers to other peers as long as sufficient capacity is available. It thus pays the least cost to reach the balanced state. So, relying on the minimization of the global overload allows us to save on the cost of rebalancing.

7.6 Conclusion

This paper has described BALLS, a structured P2P overlay that allows for simultaneous index management load balancing and storage load balancing. This system achieves integrated load balancing due to the separation among object location and object key location. This separation also saves on maintenance costs against peer dynamicity.

Experiments have evaluated and confirmed the effectiveness of the proposed index management load balancing and storage load balancing methods. We also compared the storage load balancing results that are obtained from two storage load transfer strategies : cost-oriented and overload-oriented. The former ensures that the transfer cost is limited by the overload reduction but reduces the effectiveness.

Our evaluation confirmed that the proposed index management load balancing and storage load balancing methods can simultaneously operate without reducing the effect of each other. Experiments of the load balancing methods on a P2P system that associates the objects in their key location showed that this association breaks the independence of the load balancing methods and degrades their effectiveness. We also compared BALLS with other structured P2P systems on the cost of maintenance. The comparison confirmed that BALLS reduces maintenance costs.

An evaluation of BALLS in a real network is planned. The fault-tolerance improvement and the evaluation of communication costs to maintain object consistency also need to be considered. These will allow us to produce and deploy an applicable P2P system.

7.A Proof of Theorem 1

Proof : Suppose that a peer p ($A_p < 0$) makes an 1-direction storage load transfer to a peer q ($A_q > 0$). Their overload before and after the transfer are, respectively :

$$\begin{aligned} W_p &= (|A_p| - A_p)/2 = -A_p \\ W_q &= (|A_q| - A_q)/2 = 0 \\ W'_p &= (|A_p + S_{pq}| - A_p - S_{pq})/2 \\ W'_q &= (|A_q - S_{pq}| - A_q + S_{pq})/2 \end{aligned}$$

The change to the combined overload of p and q is :

$$\begin{aligned} \Delta W &= W'_p + W'_q - W_p - W_q \\ &= (|A_p + S_{pq}| + |A_q - S_{pq}| + A_p - A_q)/2 \end{aligned} \tag{7.10}$$

We determine S_{pq} for the following cases :

1. if $S_{pq} \leq -A_p$,

   (a) if $S_{pq} \leq A_q$, (7.10)$\Rightarrow \Delta W = -S_{pq}$.

   Because $0 < S_{pq} \leq \min(-A_p, A_q)$, we have $0 > \Delta W \geq \max(A_p, -A_q)$.

   (b) if $S_{pq} > A_q$, (7.10)$\Rightarrow \Delta W = -A_q$.

   Because $A_q < S_{pq} \leq -A_p$, we have $-A_q > A_p$ and $\Delta W = \max(A_p, -A_q)$.

2. if $S_{pq} > -A_p$,

   (a) if $S_{pq} \leq A_q$, (7.10)$\Rightarrow \Delta W = A_p$.

   Because $A_q \geq S_{pq} > -A_p$, we have $-A_q < A_p$ and $\Delta W = \max(A_p, -A_q)$.

(b) if $S_{pq} > A_q$, (7.10)$\Rightarrow \Delta W = A_p - A_q + S_{pq}$.

$\Delta W < 0$ only if $S_{pq} < -A_p + A_q$.

The following graph shows the dependency of ΔW on S_{pq}.

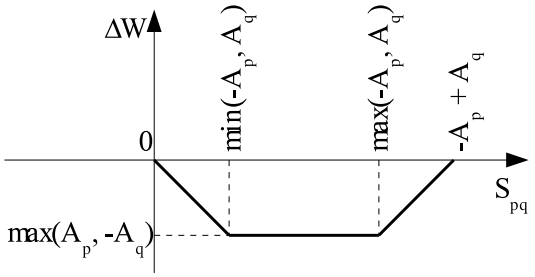

FIG. 7.32 – $\Delta W(S_{pq})$ vs. S_{pq} in the 1-direction storage load transfer mode

By definition, the optimal 1-direction storage load transfer requires :

$$\left.\begin{array}{l}\text{first, } \Delta W \text{ is the most negative} \\ \text{second, } S_{pq} \text{ is the smallest possible}\end{array}\right\} \tag{7.11}$$

The variation of ΔW shows that only S_{pq} chosen the closest to $\min(-A_p, A_q)$ such that $S_{pq} < -A_p + A_q$ satisfies (7.11).

Due to its definition, the bounded optimal 1-direction storage load transfer restricts S_{pq} not higher than the reduction of $W_p + W_q$. From the ΔW graph, S_{pq} does not exceed the combined overload reduction (being $-\Delta W$) only when $S_{pq} \leq \min(-A_p, A_q)$. In order to achieve the most negative ΔW while ensuring the above restriction, S_{pq} must be the greatest but not higher than $\min(-A_p, A_q)$.

□

7.B Proof of Theorem 2

Proof : Let W_p, W_q, W'_p, and W'_q denote the overload of p and q, before and after the transfer, respectively.

$$\begin{aligned} W_p &= (|A_p| - A_p)/2 = -A_p \\ W_q &= (|A_q| - A_q)/2 = 0 \\ W'_p &= (|A_p + S_{pq} - S_{qp}| - A_p - S_{pq} + S_{qp})/2 \\ W'_q &= (|A_q - S_{pq} + S_{qp}| - A_q + S_{pq} - S_{qp})/2 \end{aligned}$$

The change to the combined overload of p and q after the transfer is :

$$\begin{aligned} \Delta W &= W'_p + W'_q - W_p - W_q \\ &= (A_p - A_q + |A_p + S_{pq} - S_{qp}| + |A_q - S_{pq} + S_{qp}|)/2 \end{aligned} \tag{7.12}$$

Given A_p, A_q, and S_{pq}, ΔW is a function of S_{qp}. For achieving the optimal 2-direction storage load transfer, S_{qp} must satisfy (7.13).

$$\left.\begin{array}{l} \text{first, } \Delta W \text{ is the most negative} \\ \text{second, } S_{qp} \text{ is the smallest possible} \end{array}\right\} \tag{7.13}$$

We determine S_{qp} (respecting $0 \leq S_{qp} < S_{pq}$) for the following cases :

1. if $S_{pq} \leq A_q$,
   (a) if $S_{pq} \leq -A_p$, (7.12)$\Rightarrow \Delta W = -S_{pq} + S_{qp} < 0$. From (7.13), we choose $S_{qp} = 0$.
   (b) if $S_{pq} > -A_p$,
      i. if $S_{qp} \geq A_p + S_{pq} > 0$, (7.12)$\Rightarrow \Delta W = -S_{pq} + S_{qp} \Rightarrow A_p \leq \Delta W < 0$.

ii. if $0 \leq S_{qp} < A_p + S_{pq}$, (7.12)$\Rightarrow \Delta W = A_p < 0$.

In case 1(b)ii, both ΔW and S_{qp} are respectively less than those in case 1(b)i. Thus, from (7.13), we choose $S_{qp} = 0$, which falls within case 1(b)ii.

In case 1 ($S_{pq} \leq A_q$), $S_{qp} = 0$.

2. if $S_{pq} > A_q$,

   (a) if $S_{pq} \leq -A_p$,

      i. if $S_{qp} \geq -A_q + S_{pq} > 0$, (7.12)$\Rightarrow \Delta W = -S_{pq} + S_{qp} \Rightarrow -A_q \leq \Delta W < 0$.

      ii. if $0 \leq S_{qp} < -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = -A_q < 0$.

      In case 2(a)ii, both ΔW and S_{qp} are respectively less than those in case 2(a)i. Thus, from (7.13), we choose $S_{qp} = 0$, which falls within case 2(a)ii.

      In case 2a ($A_q < S_{pq} \leq -A_p$), $S_{qp} = 0$.

   (b) if $S_{pq} > -A_p$,

      i. if $S_{qp} \geq A_p + S_{pq}$,

         A. if $S_{qp} \geq -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = -S_{pq} + S_{qp} < 0$. Because $S_{qp} \geq \max(A_p, -A_q) + S_{pq}$, we have $\Delta W \geq \max(A_p, -A_q)$. If S_{qp} increases, ΔW increases as well.

         B. if $S_{qp} < -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = -A_q < 0$. This case occurs when $A_p < -A_q$. Then, $\Delta W = \max(A_p, -A_q)$.

      ii. if $S_{qp} < A_p + S_{pq}$,

         A. if $S_{qp} \geq -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = A_p < 0$. This case occurs when $A_p > -A_q$. Then, $\Delta W = \max(A_p, -A_q)$.

         B. if $S_{qp} < -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = A_p - A_q + S_{pq} - S_{qp}$. (7.13)$\Rightarrow S_{qp} > A_p - A_q + S_{pq}$. If S_{qp} increases, ΔW decreases. In addition, because $S_{qp} < \min(A_p, -A_q) + S_{pq}$, we have $\Delta W > A_p - A_q - \min(A_p, -A_q) = \max(A_p, -A_q)$.

In case 2b, S_{qp} affects ΔW in three phases as illustrated in Figure 7.33 :

– when S_{qp} increases from $\max(0, A_p - A_q + S_{pq})$ to $\min(A_p, -A_q) + S_{pq}$, ΔW decreases from $\min(0, A_p - A_q + S_{pq})$ to $\max(A_p, -A_q)$;

– when S_{qp} falls within the range $[\min(A_p, -A_q) + S_{pq}, \max(A_p, -A_q) + S_{pq}]$, then $\Delta W = \max(A_p, -A_q)$;

– when S_{qp} increases from $\max(A_p, -A_q) + S_{pq}$ to S_{pq}, then ΔW increases from $\max(A_p, -A_q)$ to 0.

To satisfy (7.13), we choose S_{qp} the closest to $\min(A_p, -A_q) + S_{pq}$.

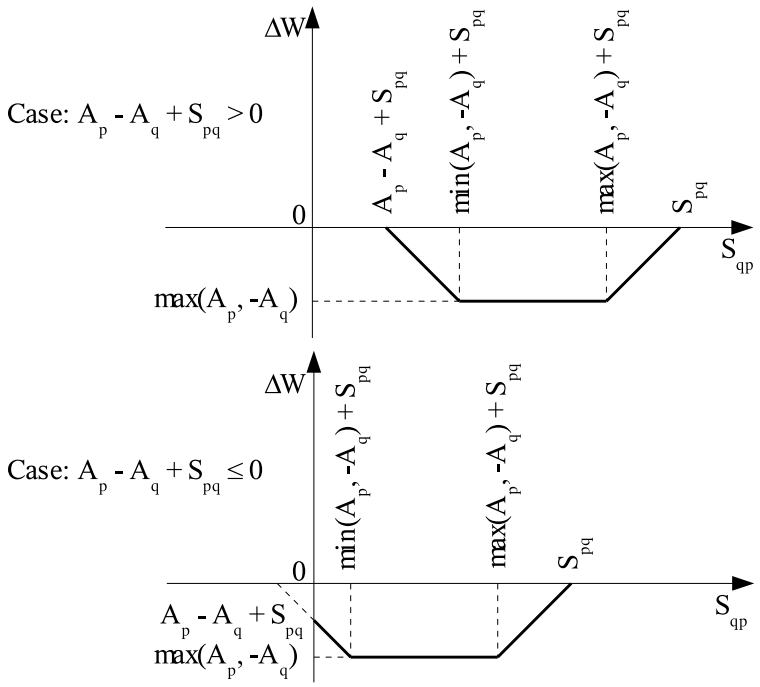

FIG. 7.33 – $\Delta W(S_{qp})$ vs. S_{qp} where $S_{pq} > \max(-A_p, A_q)$

In case 2b ($S_{pq} > \max(-A_p, A_q)$), S_{qp} is the closest to $\min(A_p, -A_q) + S_{pq}$ and respects the condition : $0 \leq S_{qp} < S_{pq}$ and $S_{qp} > A_p - A_q + S_{pq}$.

□