Fast and Cost-Effective Online Load-Balancing in Distributed Range-Queriable Systems

Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris, *Member*, *IEEE*

Abstract—Distributed systems such as Peer-to-Peer overlays have been shown to efficiently support the processing of range queries over large numbers of participating hosts. In such systems, uneven load allocation has to be effectively tackled in order to minimize overloaded peers and optimize their performance. In this work, we detect the two basic methodologies used to achieve load-balancing: Iterative key redistribution between neighbors and node migration. We identify these two key mechanisms and describe their relative advantages and disadvantages. Based on this analysis, we propose *NIXMIG*, a hybrid method that adaptively utilizes these two extremes to achieve both fast and cost-effective load-balancing in distributed systems that support range queries. We theoretically prove its convergence and as a case study, we offer an implementation on top of a Skip Graph, where we thoroughly validate our findings in a variety of static, dynamic and realistic workloads. We compare *NIXMIG* with an existing load-balancing algorithm proposed by Karger and Ruhl [1] and our experimental analysis shows that, *NIXMIG* can be as much as three times faster, requiring only one sixth and one third of message and item exchanges, respectively, to bring the system to a balanced state.

Index Terms—Peer-to-peer systems, load-balancing, range queries.

1 Introduction

Data skew is a well-documented concern for a variety of applications. It has been widely observed that most Internet-scale applications, including P2P ones, exhibit highly skewed workloads (e.g., [2], [3], etc). Failing or departing nodes further reduce the availability of various content. Consequently, resources become scarce, servers get overloaded and throughput can diminish due to high workloads that, in many cases, can by themselves cause denial of service [4].

One way to handle hotspots and balance load is by applying hash functions that transform skewed data access patterns to uniform distributions. Nevertheless, this transformation comes at the cost of destroying content locality, and thus cannot be used in situations where objects need to placed in an order-preserving way. Distributed data structures that support range queries is such an example: The keys are partitioned in the network nodes so that a natural order is preserved and each range query is efficiently handled by a small number of peers. The interest in such structures is increasing, as they can be very useful in a variety of situations: distributed databases [5], online games [6], web servers [7], data warehousing [8], etc.

Another orthogonal way to deal with data skew is the replication of popular items in numerous nodes. However, the content locality constraint minimizes available replica candidates (allowing, for instance, only few-hop away neighbors), something that makes balancing even more difficult. What is more, replication not only needs to change the underlying routing protocol to handle multiple replica locations during item searches and insertions, but it must also deal with consistency issues during object updates.

In such cases, load-balancing methods that redistribute items between nodes are an appealing solution. The highly dynamic and large-scale nature of these distributed data structures, where it is difficult for a single node to have a total network workload overview, poses two basic requirements: online functionality (i.e., the property to make correct decisions only with partial, local workload knowledge) and workload adaptivity (i.e., the ability to quickly respond to workload changes).

In current bibliography, a variety of methods exists focusing on achieving efficient load-balancing for such structures, whether they utilize the notion of "virtual servers" [9], [10], [11], [12], [13], [14] or not [1], [14], [15], [16], [17], [18], [19], [20], [21]. Yet, they can be categorized in two general strategies: *Node Migration* and *Neighbor Item Exchange*. These techniques represent two different approaches to handling the problem: *Node Migration* utilizes underloaded peers by placing them in overloaded areas of the network (see Fig. 1, where the height of the bars shows the load of each node, while their width reflects the number of keys served). The newly arriving peer takes up part of the load of its new neighbors. *Neighbor Item Exchange* balances load through iterative item exchanges between neighboring nodes (see Fig. 2). The majority of proposed approaches utilize a version of these two schemes in order to finally balance load among peers each responsible for a given range of the data. While they both achieve their goal, their speed and cost greatly vary, making a method that utilizes only one of them inefficient for all cases.

Our contribution[1] can be summarized in the following:

- We formally identify these two different methodologies that, iteratively applied, perform load-balancing

- The authors are with the Computing Systems Laboratory, Department of Computer Science, School of Electrical and Computer Engineering, National Technical University of Athens, Zografou Campus, 157 73 Zografou, Athens, Greece.
E-mail: {ikons, dtsouma, nkoziris}@cslab.ece.ntua.gr.

Manuscript received 6 Sept. 2009; revised 2 June 2010; accepted 23 Sept. 2010; published online 8 Nov. 2010.
Recommended for acceptance by J. Cao.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-09-0408.
Digital Object Identifier no. 10.1109/TPDS.2010.200.

[1] A preliminary version of this work was presented in the P2P'09 conference [22]. In this paper, we elaborate on *NIXMIG*, proving speed and convergence, we present new experimental results including a comparison with another algorithm [1] and a detailed literature overview.

1045-9219/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

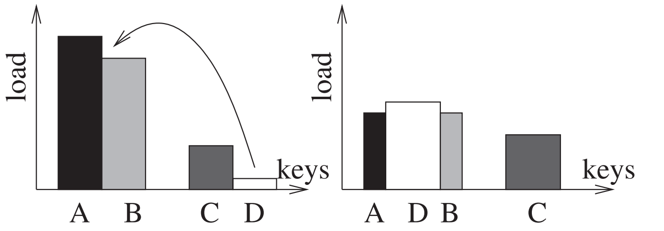

Fig. 1. Node Migration example. Node D is placed between nodes A and B and shares part of their load.

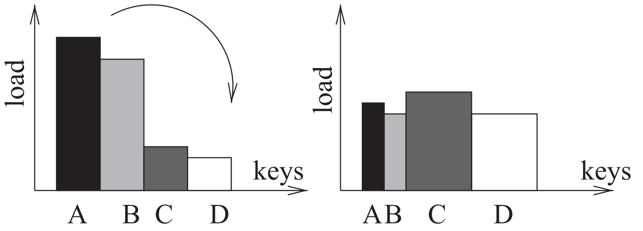

Fig. 2. Neighbor Item Exchange example. Iterative key exchanges between (A,B), (B,C), and (C,D) node pairs produce a balanced load.

on distributed range-partitioned data structures. We describe their mechanisms and analyze their performance in terms of completion time and communication cost. An important result of our work is the observation that, through mere key exchanges the achieved result can be highly delayed and the number of exchanged items can be very large, whereas using only node migrations the cost of updating the structure is considerably increased.

- Based on this analysis, we describe a hybrid method that utilizes both item exchange and node migration in order to minimize overloaded peers and balance the load distribution among them. This method manages to adjust the use of migrating nodes with the neighbor item exchange operations: Load moves in a "wave-like" fashion from more to less loaded regions of the structure adaptively, using our version of the Neighbor Item Exchange mechanism. When we locally identify highly overloaded regions, we activate Node Migration. We also present smart, "skew aware" remote underloaded node location and placement mechanisms that further decrease *NIXMIG*'s bandwidth consumption. We theoretically study the algorithm's convergence existence and speed along with the preconditions that need to hold for the system to reach an equilibrium.
- We present a Skip Graph [23] implementation on top of which we apply and compare the hybrid versus simple node migrations, neighbor item exchanges, and another load-balancing algorithm proposed by Karger and Ruhl [1]. We measure and compare their behavior in a variety of skewed, dynamic, and realistic workloads. Our results validate the analysis of the previous sections and show that our method balances at low cost (requires only one sixth and one third of message and item exchanges, respectively, compared to [1]) and high convergence rate (it is three times faster than [1]), adapts to changing workloads and is highly customizable.

The remainder of this paper is organized as follows: Section 2 gives the reader the basic notation and formulation of our problem. Section 3 describes and analyzes the two different primitive mechanisms for load-balancing, while in Section 4, we present and theoretically analyze our hybrid method. Our experimental results are detailed in Section 5, while Related Work and the Conclusions Section conclude our work.

2 Notation and Problem Setup

We consider the indexing and storing of M keys $(1, \dots, M)$ in N nodes $(1, \dots, N)$, where N ≪ M. We assume that a key represents an object or item, hence we shall use these terms interchangeably. We consider that M keys are divided along N partitions (ranges) with boundaries $r_1 <= r_2 <= \cdots <= r_N$ (obviously, $r_i \in [1, M], \forall i \in [1, N]$). Each node N_i stores and indexes keys for the partition $[r_i, r_{i+1})$. Nodes that manage adjacent ranges are said to be neighbors. We consider two different directions: *forward*, toward which indexed values are increasing and *backward*, where values are decreasing. Node N_i's forward and backward neighbors are Node N_{i+1} that is responsible for the adjacent range $[r_{i+1}, r_{i+2})$ and Node N_{i-1} that is responsible for the adjacent range $[r_{i-1}, r_i)$, respectively. As *item load* $l_j(t), j \in [1, M]$ at time t, we define the number of user requests for this specific item over a specific time interval (for instance, keys/sec). Item load can be viewed as a portion of bandwidth (kb/sec) consumed on queries for this key. The *server load* $L_i(t)$ of node N_i at time t is the sum of the loads of the items that it stores: $L_i(t) = \sum_{j=r_i}^{r_{i+1}} l_j(t)$.

We are interested in keeping the natural ordering of the indexed keys, so as to facilitate the routing and answering of range queries. Each stored item has a different popularity that is assumed not to be known beforehand and to change over time. Users perform both exact match and range queries. In the case of range queries, more than one node may be contacted in order for the correct answer to be computed.

We assume that each node N_i, according to its capabilities sets a local load threshold, $thres_i$. When the load exceeds this value $L_i(t) > thres_i$, the node wishes to shed some of its load according to the load-balancing algorithm that is implemented.

Our goal is to transform the set of partition boundaries through consecutive item exchanges or node migrations after some time so that $L_i(t') < thres_i, \forall i \in (0, N]$. In addition, our goal is to achieve a balanced load distribution.

3 Load Balancing Using Neighbor Item Exchange and Node Migration

Balancing is performed by transferring keys from overloaded peers to less loaded ones. The necessity for preserving order in a range-queriable data structure requires that any item exchange must be performed only between neighboring nodes in the structure. Nevertheless, there are situations where several neighboring nodes experience similar load stress. In that case, distant underloaded peers can gracefully depart from their place, join in the overloaded area, and take a portion of its keys. While this operation seems more efficient, a large number of message exchanges is required for the remote node location and the overlay structure maintenance.

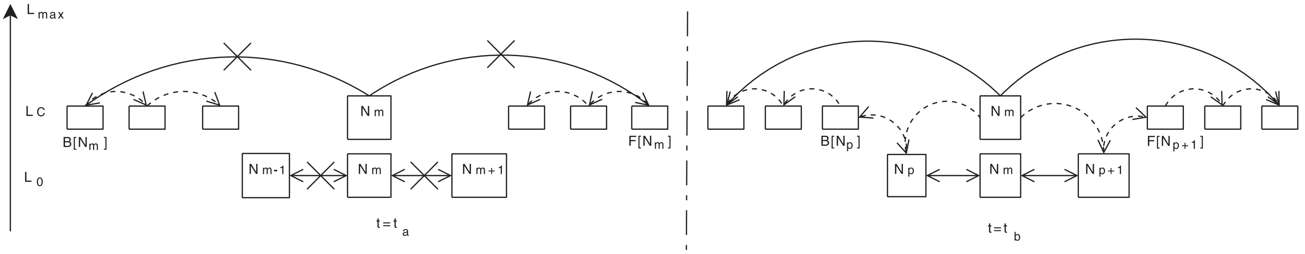

Fig. 3. Overlay maintenance communication cost for migration of node N_m next to node N_p.

Distributed structures that support range queries perform routing in logarithmic time by maintaining a routing table list of $logN$ increasingly distant nodes (for an overlay of size N). Without loss of generality, we consider these nodes to be placed in L_{max} levels (at the lowest level, L_0, each node holds the IDs of its immediate neighbors, etc). The maintenance cost of this overlay is a costly procedure in terms of communication exchange between the participating nodes: Fig. 3 depicts the message exchanges that occur when node N_m leaves its place at time t_a (left part of Fig. 3) and rejoins next to node N_p at time t_b (right part of Fig. 3). Solid lines represent node routing links, whereas dotted ones represent the messages required for overlay maintenance. In the described structure, every node contains $2L_{max}$ routing entries (backward and forward for every level). For simplicity, we describe the procedure for a random level, L_c: Before node N_m leaves its place, it removes every forward and backward link stored in tables $F[N_m]$ and $B[N_m]$, respectively (lines marked with an X). This triggers a number of message exchanges where nodes that were in $F[N_m]$ and $B[N_m]$ (i.e., N_m's old neighbors) contact a number of distant nodes in order to fill their routing table "hole" (dotted lines on the left side of Fig. 3). This operation is carried out for every old neighbor in $2L_{max}$ levels. When node N_m reenters between nodes N_p and N_{p+1} (right side of Fig. 3) at time t_b, it uses $F[N_{p+1}]$ as forward and $B[N_p]$ as backward links, scans the structure (dotted lines) and creates its own routing table (solid lines).

We now describe two different load-balancing algorithms: Neighbor Item Exchange (*NIX*), that transfers only keys between neighboring nodes and Node Migration (*MIG*) that transfers both keys and nodes from remote arbitrary locations.

3.1 NIX

The load exchange between neighboring nodes is described in Algorithm 1. For simplicity, we describe the situation where keys are transferred from Node N_i to its forward neighbor N_{i+1}. The transferring node (which we will refer to as the *splitter* peer) sets a pointer $j = r_{i+1}$ and scans its range backwards. The procedure stops when sufficient number of items have been found so as to fulfill its request. Moreover, helper nodes can alleviate their neighbors immediately: the helper can answer queries on behalf of the neighbor while the process is not completed, as they are both aware of the location of the pointer j. We note here that the splitter-helper node ID change caused by the range adjustment does not have to be reflected immediately to their remote neighbor's routing tables, as the overlay consistency is preserved (the node ordering remains unaltered). Therefore, the new IDs can be disseminated lazily with the first routing maintenance message exchange. Nevertheless, it is obvious that a major disadvantage of *NIX* is that possibly many iterative such operations may be needed in order to balance load inside large regions of loaded peers.

Algorithm 1. NIX($N_i \rightarrow N_{i+1}, load$)

```
{$N_i$ calculates key range to pass to $N_{i+1}$}
$j \leftarrow r_{i+1}$
while $j \geq r_i$ do
    if $\sum_{m=j}^{r_{i+1}} l_m \geq load$ then
        key range is $[j, r_{i+1}]$
        break
    else
        $j \leftarrow j - 1$
    end if
end while
$N_i\ transfers\ [j, r_{i+1}]\ to\ N_{i+1}$
$New\ N_i\ partition : [i, j]$
$New\ N_{i+1}\ partition : [j, r_{i+2}]$
```

3.2 MIG

In Algorithm 2, we describe the situation where Node N_m leaves its place to join next to overloaded Node N_p and take a portion of its keys. N_p locates N_m by issuing probing messages to its routing table neighbors until it locates an idle and underloaded peer that could migrate next to it. *MIG* is performed in two phases: In the first phase, Node N_m transfers its partition to its neighboring node N_{m-1}, clears its routing links, and informs them to search for a new entry (leave phase, lines 1-5 of Algorithm 2). In the second step of the procedure (the join phase), Node N_m places itself next to the overloaded peer, accepts a portion of its load and creates its new routing table (lines 6-7 of Algorithm 2). This process was thoroughly described in Fig. 3.

Algorithm 2. MIG $(Node\ N_m \rightarrow Node\ N_p, load)$

```
1: NIX($N_m \rightarrow N_{m-1}, L_m$)
2: for all $N_i$ in $N_m$'s routing table do
3:     $N_m$ removes link to $N_i$
4:     $N_i$ searches for new routing entry
5: end for
6: NIX($N_p \rightarrow N_m, load$)
7: $N_m$ creates new routing table
```

3.3 Analysis

In the following, we present an analysis to calculate the theoretical worst upper bounds for the completion time and amortized balancing costs (i.e., costs per balancing operation) of *NIX* and *MIG*. We consider three types of amortized balancing costs, with respect to bandwidth consumption

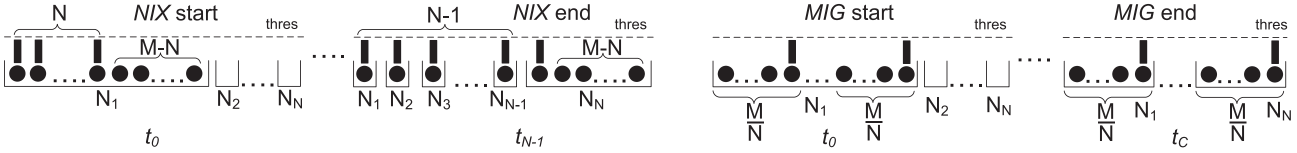

Fig. 4. Worst case of initial setup and converged balanced network for the *NIX* and *MIG* cases.

for: item exchanges between nodes (C_{itx}), overlay maintenance during migrations (C_{ovm}), and locating underloaded peers during probing (C_{prb}).

In Theorems 1 and 2, we use the aggregate method of amortized analysis to calculate the average cost of each balancing operation and completion time of *NIX* and *MIG* in the worst case of an initial setup (i.e., worst upper bound of amortized cost). We utilize the notations of Section 2.

Theorem 1. *In the worst case, the running time of* NIX *is $O(N)$ and the amortized cost per balancing operation is $O(M)$.*

Proof. In the first picture of Fig. 4, we present an initial setup of node and item combination that leads *NIX* to its worst behavior in terms of completion time and item exchanges. Buckets represent nodes and balls depict items. Bars above items represent the unit item load $l_j = 1$. For simplicity, we consider that for each node, $thres_i = 1$. At the beginning, N_1 contains all M objects, of which only the leftmost N are requested (i.e., they have $l_j = 1, j \in (0, N])$ and the rest M-N are not queried (i.e., $l_j = 0$, $j \in (M - N, M]$). All other nodes are empty. Since $L_1 = N > thres_1$, N_1 will perform a *NIX* operation with its neighbor N_2 at t_0 and it will transfer to it a total of $M - 1$ keys, keeping only the leftmost key, so that $L_1 = l_1 = 1 <= thres_1$. Likewise, at t_1 node N_2 will transfer $M - 2$ keys to its right neighbor N_3 keeping only its leftmost key. Finally, after $N - 1$ steps, in the second picture of Fig. 4 all nodes are balanced, since they will be responsible for a single item whose load is 1. N_N will also contain the remaining $M - N$ zero load keys. Given that $N - 1$ steps are needed, the running time of *NIX* is $O(N - 1) = O(N)$. By summing all moved items in every step, we have the total cost $\sum_{i=1}^{N-1}(M - i) = M(N - 1) - \frac{(N-1)N}{2}$. As no probing and overlay maintenance is necessary, the cost per operation then is $C_{itx} = \frac{M(N-1) - \frac{(N-1)N}{2}}{N-1} = M - \frac{N}{2} = O(M - N)$. Since $N \ll M$, the cost then is $O(M)$. □

Theorem 2. *In the worst case, the running time of* MIG *is constant $O(1)$ and the amortized cost per balancing operation is $O(\frac{M}{N} + logN)$.*

Proof. In the third picture of Fig. 4, we depict a worst initial network setup for the *MIG* case. Similar to *NIX*, N_1 contains all M objects, of which only N are requested and every node sets $thres_i = 1$. All other $N - 1$ nodes are empty at first. Requested items are evenly distributed in the ID space: for every $\frac{M}{N}$ objects, there is one with $l_j = 1$ (for instance, $l_j = 1$ if j mod N $= 0$, and 0 otherwise). In this setup, N_1 will initiate $N - 1$ migrations with the rest of the nodes, where in each migration $\frac{M}{N}$ keys are offloaded from N_1 to the helper. Finally (fourth picture of Fig. 4), a total of $\sum_{i=1}^{N-1}\frac{M}{N} = (N - 1)\frac{M}{N}$ keys are transferred. The cost for item exchanges then is $C_{itx} = \frac{(N-1)\frac{M}{N}}{N-1} = O(\frac{M}{N})$ which is basically the cost for a node insertion or deletion (see Theorem 3 of Karger's work [1]). The probing cost C_{prb} is $O(logN)$ since it involves contacting $logN$ neighbors. Moreover, in most DHT-like networks, overlay maintenance costs $C_{ovm} = O(logN)$ messages. Therefore, the total *MIG* cost is $C_{itx} + C_{ovm} + C_{prb} = O(\frac{M}{N} + logN)$. Migrations take a constant number of steps as, unlike *NIX* operations, they are executed in parallel: therefore, we consider *MIG* running time to be $O(1)$ (although overlay maintenance usually takes $O(logN)$ time, this can happen lazily after the key transfer phase). □

To gain insight into the behavior of the two algorithms in a more general case, let us consider a typical "balls into bins" setup, with N items being uniformly distributed among N nodes (we only consider N out of M items, since these items affect node loads). The fraction of underloaded nodes, i.e., nodes with a load less or equal to 1, is calculated by estimating the probability of a node to hold either one or no popular item. Utilizing the equation that calculates the probability of a particular bin to have exactly k balls we have: $P[N_j \; is \; underloaded] = \sum_{k=0}^{1} P[L_j = k] = \sum_{k=0}^{1}\binom{N}{k}(\frac{1}{N})^k(1 - \frac{1}{N})^{N-k}$. For large N, this is equal to $\frac{1}{e} + \frac{1}{e} = 0.74$. Moreover, the maximum load of a node is $\frac{logN}{loglogN}$. Thus, only 26 percent of the nodes is overloaded and the most loaded node(s) are well under the initial load of N in the worst case of *NIX* and *MIG*. Both algorithms benefit in this case: *NIX* will initiate small concurrent waves of item exchanges, finishing faster than $O(N)$ (as more waves are done in parallel) and less costly than $O(M)$ (as waves involve a smaller number of nodes and transfer a smaller amount of the id space). Similarly, *MIG* will transfer less items than $O(\frac{M}{N})$, since a fraction ($P[L_j = 1] = \frac{1}{e} = 37\%$) of the nodes will not participate in the balancing procedure, as their load is equal to their $thres$ value.

Although *MIG* performs better in terms of completion time and exchanged messages for a large number of N, it needs extra messages for overlay reorganization and probing. This can be avoided with the selective use of *NIX* operations.

In Fig. 5, a situation where a wave of *NIX* operations is more favorable compared to *MIG* is presented: Node A can shed its load toward its underloaded neighbors without the need for extra remote nodes, leading the neighborhood in a balanced state (right side of Fig. 5). In Fig. 6, we describe a situation where a *MIG* operation is more cost-effective than a number of *NIX* operations. Node A is located between nodes that their load is near their $thres$ value (left side of Fig. 6). In this situation, a chain of *NIX* operations would

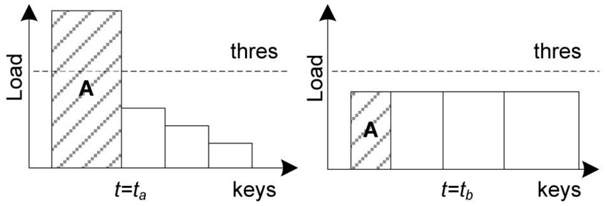

Fig. 5. Balancing effect of a chain of *NIX* operations.

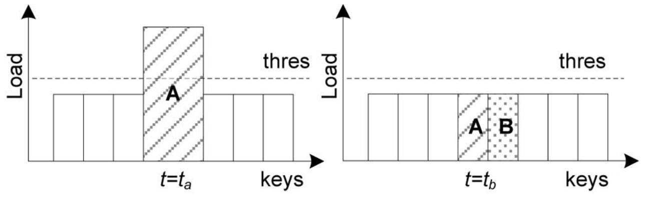

Fig. 6. Balancing effect of a single *MIG* operation.

simply forward the load from one node to another, as there is no nearby underloaded neighbor that could absorb it. On the other hand, the migration of a remote node B next to A (right side of Fig. 6) solves the problem in one step, justifying the extra number of required probing and maintenance messages needed to locate the underloaded peer and fix the topology, respectively. In any case, in order for A to decide the appropriate balancing action, a clear view of the neighborhood's load is required.

From this, it is obvious that fewer *MIG* operations can produce the same result to considerably more *NIX* ones, as helping nodes can be placed anywhere. Nevertheless, it is also evident that each node migration is costly, while the location of possible helpers and their exact new location has to be optimized. On the other hand, *NIX* avoids probing and routing maintenance messages, but it requires a large number of item exchanges between successive nodes, especially when it is applied in the "middle" of an overloaded neighborhood.

Procedure 3. REMOTEWAVE $(N_{p+lc+1}, exNodes_{lc+1}\ hops)$

```
Find$N_m$ such that
    $L_m < thres_m$ and $N_m$ is idle
if such $N_m$ $exists$ then
    $tmpL_m = L_m, j = 0$
    while $tmpL_m <= tmpL_{p+lc+1}$
    and $j <= exNodes_{lc+1}$ do
        if $N_{m+j+1}$ is idle then
            $tmpL_m += L_{m+j+1}$
            Node $N_{m+j+1}$ sends a LockRequest to $N_{m+j+2}$
        else {$N_{m+j+1}$ is locked}
            $N_{m+j}$ aborts lock
        end if
        $j = j + 1$
    end while
    $rNodes = j$
end if
return $rNodes$
```

4 NIXMIG

In this section, we describe *NIXMIG*, our proposed hybrid approach. The goal of *NIXMIG* is to balance load by adaptively choosing to utilize either *NIX* or *MIG*. The rationale behind our method is that *MIG* is fast but costly, whereas *NIX* is slow but cost-effective. Hence, we devise a scheme that, using only local knowledge, identifies conditions where *MIG* is necessary to speed up the balancing process but is not excessively utilized. In short, when *NIX* operations cannot alleviate an overloaded neighborhood, our method employs node migrations for faster load relief in that area.

TABLE 1
NIXMIG Variables

variable	definition
ttl	Maximum number of contacted nodes per wave
lc	Number of nodes reserved for the *NIX* wave
$rNodes$	Number of nodes reserved for the *MIG* wave
$tmpL_p$	Load of node N_p if balancing is performed
$movedLoad_{lc}$	Load that will be moved from N_{p+lc} to N_{p+lc+1} if balancing is performed
$exNodes_{lc}$	Number of extra remote nodes needed at step lc of Procedure 5
N_m	Remote node that will accept migration load
a	Load fraction accepted by the helper if the splitter's load is more than $overThres$

4.1 Algorithm

NIXMIG (Algorithm 4) is initiated when the load of a node N_p passes its self-imposed $thres_p$ value and it is performed in three phases: In the first phase (Exam phase), the overloaded node examines the load status of a number of neighboring nodes (Procedure 5) and, if necessary, an additional number of distant nodes is contacted (Procedure 3). In Table 1, we explain the variables used by the aforementioned methods. The node examination is performed in a wave-like manner toward one direction of the structure, where each node contacts its successor. When the first phase is successful, then the algorithm proceeds to the *NIX* phase (lines 5-12 of Algorithm 4) and portions of keys are iteratively transferred from one neighbor to another. Finally, the algorithm proceeds to the *MIG* phase (lines 14-17 of Algorithm 4), where the reserved underloaded nodes of the remote wave offload their keys to their neighbor and take a portion of the range of the final node of the *NIX* wave. We note here that the *MIG* phase is optional: it is triggered only if extra remote nodes are needed to absorb a neighborhood's load. Moreover, reserved (i.e., locked) nodes continue to answer user queries but they do not participate in or initiate other balancing actions until they are unlocked (lines 11 and 16 of Algorithm 4) or a time-out has occurred.

Algorithm 4. NIXMIG $(Node\ N_p, ttl\ hops)$

```
1: LOCALWAVE($Node\ N_p, ttl\ hops$)
2: if $exNodes_{lc+1} > 0$ then
3:     REMOTEWAVE($Node\ N_{p+lc+1}, exNodes_{lc+1}\ hops$)
4: end if
5: for $i = 0$ to $lc$ do
6:     if $L_{p+i} > overThres_{p+i}$ then
7:         $load = a(L_{p+i} - thres_{p+i})$
8:     else if $L_{p+i} > thres_{p+i}$ then
9:         $load = L_{p+i} - thres_{p+i}$
10:    end if
11:    NIX($N_{p+i} \rightarrow N_{p+i+1}, load$), unlock $N_{p+i}$
12: end for
```

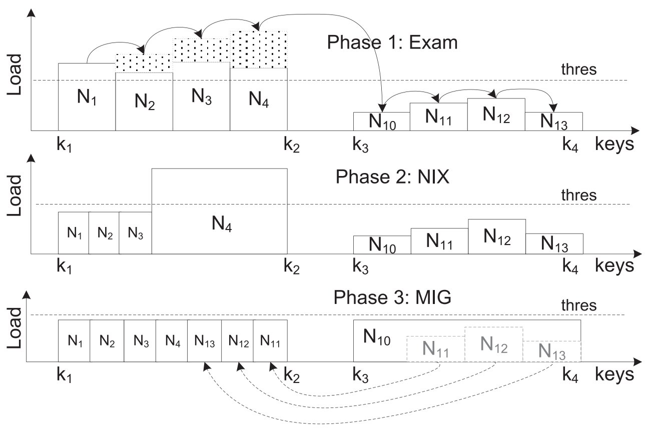

Fig. 7. A successful *NIXMIG* operation.

```
13: if $rNodes > 0$ then
14:     for $i = 0$ to $rNodes$ do
15:         MIG($N_{m+i+1} \rightarrow N_{p+lc}, \frac{tmpL_{p+lc}}{rNodes}$)
16:         unlock $N_{m+i+1}$
17:     end for
18: end if
```

In Fig. 7, we depict the phases of a successful *NIXMIG* operation initiated by node N_1. For clearer presentation, we assume that all nodes have equally set their $thres$ value (dotted horizontal line). In the Exam phase, N_1 issues a Lock Request that eventually reaches N_4 through N_2 and N_3. N_4 calculates the number of extra nodes that are needed to migrate to the neighborhood to absorb its load, and issues a new request for remote node reservation to node N_{10}. When N_{10} reserves nodes N_{11}, N_{12}, and N_{13}, the *NIX* Phase begins. In the *NIX* phase of Fig. 7, nodes N_1 to N_3 iteratively shrink their responsible range by adjusting their boundaries and drop their load under their required $thres$ value. At the end of Phase 2, most of the neighborhood's load ends up to N_4, but this will happen for a very small period of time, as N_4 has already reserved the requested number of remote nodes to share this load. Finally, in the *MIG* Phase, the remote underloaded reserved nodes N_{11}, N_{12}, and N_{13} sequentially offload their keys to N_{10}, place themselves next to N_4 and take a portion of its range. We notice that at the end of Phase 3, all participating nodes' loads are below their $thres$ value. We now give a more detailed presentation of the algorithm phases.

4.1.1 Exam Phase

The Exam phase of *NIXMIG* serves a dual purpose: it examines the load status of the contacting nodes to decide the appropriate balancing actions, while reserving them to participate in the balancing procedure. The load examination begins with the node's neighborhood (Procedure 5): after each node is successfully reserved (line 3 of Procedure 5), a *NIX* operation between Node N_{p+lc+1} (that acts as a helper) and its predecessor N_{p+lc} (that acts as a splitter) is simulated by N_{p+lc+1}. The splitter's load in this calculation is assumed to be $tmpL_{p+lc}$ and is equal to the load that would end up to it if a chain of lc *NIX* operations was initiated by N_p toward N_{p+lc}. Using this variable, node N_{p+lc+1} calculates the load that will be transferred toward it ($movedLoad_{lc}$ variable). This recursive calculation can be seen in Phase 1 of Fig. 7: The $movedLoad$ variable in steps 1, 2, and 3 is depicted with the dotted rectangle above nodes N_2, N_3, and N_4, respectively. In each step, the $tmpL$ variable is calculated by adding $movedLoad$ to the nodes' current load. $tmpL_{p+nc+1}$ is used by N_{p+lc+1} to estimate the number of extra remote nodes that are required to migrate next to it to absorb the neighborhood's load ($exNodes_{lc+1}$ variable in line 11). The examination of a node's neighborhood finishes when a number of ttl nodes have been successfully reserved, or when it is estimated that more than ttl remote nodes are needed to absorb the calculated extra load (line 2).

```
Procedure 5. LOCALWAVE ($Node\ N_p, ttl\ hops$)
1: $lc = 0$, $tmpL_p = L_p$, $N_p$ sends a LockRequest to $N_{p+1}$
2: while $lc <= ttl$ and $exNodes_{p+lc+1} <= ttl$ do
3:     if $N_{p+lc+1}$ is idle then
4:         if $tmpL_{p+lc} > overThres$ then
5:             $movedLoad_{lc} = a(tmpL_{p+lc} - thres_{p+lc})$
6:         else $\{0 < tmpL_{p+lc} < thres\}$
7:             $movedLoad_{lc} = tmpL_{p+lc} - thres_{p+lc}$
8:         end if
9:         $tmpL_{p+lc} = L_{p+lc} - movedLoad_{lc}$
10:        $tmpL_{p+lc+1} = L_{p+lc+1} + movedLoad_{lc}$
11:        $exNodes_{lc+1} = \lfloor \frac{tmpL_{p+lc+1}}{thres_{p+lc+1}} - 1 \rfloor$
12:        $N_{p+lc}$ locks $N_{p+lc+1}$
           Node $N_{p+lc+1}$ sends a LockRequest to $N_{p+lc+2}$
13:    else $\{N_{p+lc+1}$ is locked$\}$
14:        $N_{p+lc}$ aborts lock
15:    end if
16:    $lc = lc + 1$
17: end while
18: return <$lc, exNodes_{lc+1}, tmpL_{p+lc+1}$>
```

When the previous phase finishes, N_{p+lc+1} uses the $exNodes_{lc+1}$ variable to decide whether extra nodes are needed (line 2 of Algorithm 4). If this is the case, it uses the previously described underloaded node location mechanism to locate a remote peer N_m (line 1 of Procedure 3). N_m then tries to reserve $exNodes_{lc+1}$ adjacent nodes that are able to leave their place and help N_p's overloaded neighborhood. These nodes will offload their keys to N_m before they migrate. The reservation is performed in a similar wave-like manner for at most $exNodes_{lc+1}$ hops. During reservations, each contacted node estimates $tmpL_m$, and if this exceeds $tmpL_{p+lc+1}$, the algorithm moves on to the next phase (line 6 of Procedure 3), with only the so far reserved nodes participating in the migration procedure. Therefore, the goal of the remote locking procedure is to reserve the required $exNodes_{lc+1}$ without overloading N_m that will accept their load when they migrate. When this phase completes, the locked nodes are ready to begin balancing actions. Moreover, during the exam phase no item exchanges are performed. If the exam phase is not successful (e.g., not enough underloaded nodes are found, or a contacted node participates in another balancing procedure), nodes are unlocked and an exponential back-off mechanism is applied to the time N_p will wait before it initiates another *NIXMIG* operation.

4.1.2 NIX Phase

When the locking phase succeeds, the algorithm proceeds to the second phase and the initiator starts an iterative procedure where portions of ranges are transferred from one locked node to its neighbor for all the lc reserved nodes (lines 5-12 of Algorithm 4). In order to calculate the portion of load that a *splitter* will shed, we introduce the *overThres* threshold, where $overThres > thres$. If the *splitter*'s load is above the *overThres*, then only a fraction a of the extra load is accepted. Otherwise, the *splitter*'s excessive load is fully accepted. The purpose of *overThres* is to smooth out the key/load exchanges between sequential *NIX* executions. When *NIX* is performed consequently in a number of overloaded nodes, some nodes may end up with a large portion of the load that was shifted to them during recursive *NIX* procedures from *all* the nodes in the forwarding path. For this exact case, the helper peer does not alleviate the splitter from all of its excessive load, instead, it only accepts a portion of it.

4.1.3 MIG Phase

The final step of the algorithm is the *MIG* wave, where a number of $rNodes$ remote locked nodes offload their keys to node N_m, leave their place and join next to N_{p+lc+1} (lines 13-18). Placing remote nodes next to N_{p+lc+1} and not between nodes N_p and N_{p+lc+1} minimizes intranode communication, as nodes N_p to N_{p+lc} are unlocked after the *NIX* wave (line 11), and their routing tables are not significantly altered.

To sum up, *NIXMIG* first examines the neighborhood of an overloaded node: if its extra load can be absorbed by its neighbors it performs a cost-effective "wave-like" set of successive item exchanges. If this is not the case because, for instance, the entire neighborhood is overloaded, it selectively initiates a more expensive migration request to speed up the process.

4.2 Enhancements

In this section, we present enhancements to the original *NIXMIG* algorithm that further decrease the bandwidth utilization of the balancing procedure. Specifically, we present remote underloaded node location and placement mechanisms which minimize traffic during balancing operations.

Remote underloaded node location. *NIXMIG*'s performance depends on its ability to easily locate an underloaded node. To avoid random probing or the maintenance of a centralized registry, we utilize the query-induced traffic to piggyback information about underloaded nodes. As packets are routed, underloaded nodes add their ids and all participating nodes extract from the incoming packets this information to a local cache. Overloaded nodes use this cache to contact underloaded ones and if they fail to do so, then they resort to random probing.

Remote underloaded node placement. When a remote underloaded node has been successfully located and reserved, the splitter must decide which range to offload to it. In situations where the load is uniformly distributed in the key space, the same load movement results in the same (in terms of transferred items) key movement. Nevertheless, in skewed distributions, this property does not hold (e.g., a small range of items may experience the same load with a larger one), and the smallest possible range must be detected and transferred during load movement. In Fig. 8, we present this detection mechanism: Overloaded node A is responsible for the key range $[r_1, r_2]$ in which the load is not uniformly placed. On the left side of Fig. 8 at $t = t_a$, node A simulates two *NIX* operations by scanning its range in the forward direction starting from r_1 (arrows marked with an X) and in the backward direction starting from r_2. Finally, on the right side of Fig. 8 at $t = t_b$, node B is placed in the forward direction of A, as this minimizes the number of transferred keys ($|r_2 - r_b| < |r_1 - r_b|$).

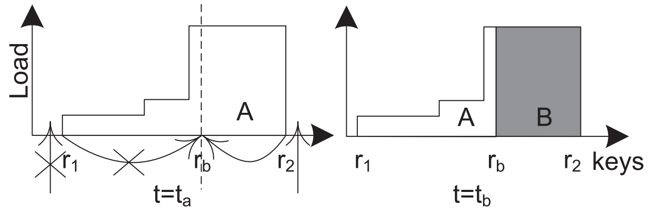

Fig. 8. Smart remote node placement: Node A scans its range and decides to place B on its forward direction, minimizing the number of transferred items $|r_2 - r_b|$.

4.3 Theoretical Analysis

Load-balancing between neighboring nodes can be classified in two general categories [24]: *diffusion* and *dimension exchange* methods. In the case of diffusion [25], every node balances its load concurrently with every other partner, whereas in the dimension exchange approach [26], every node is allowed to balance load only with one of its neighbors at a time (*NIXMIG* falls into this category). Diffusion methods are analyzed using linear algebra, whereas the analysis of dimension exchange methods is performed using a potential function argument. Potential functions map the load distribution vector at time t $\vec{w}(t) = (L_1(t), \ldots, L_N(t))^T$ into a single value that shows how "far" the system is from the balanced state. In the case of homogeneous peers, the balanced state is represented by the vector $\vec{w_{bl}} = (\bar{w}, \ldots, \bar{w})^T$ where $\bar{w} = \frac{\sum_{i=1}^{N} L_i(t)}{N}$ (every node gets an equal portion of the total load).

The goal of a balancing algorithm is to ensure that every load exchange between nodes will eventually decrease an initially large potential value and will lead the system to a more balanced (ideal) state. If this drop is ensured, the algorithm converges to an equilibrium. In the case of *NIXMIG*, we define the potential function of an arbitrary load distribution as $\phi(t) = \sum_{i=1}^{N} (L_i(t) - thres_i)^2$, where $\phi(t)$ is the square of the euclidean distance between $\vec{w}$ and the vector $\vec{w}_{thres} = (thres_1, \ldots, thres_N)^T$ in which every node's load is equal to its self-imposed *thres* value (ideal balanced state). Note that *NIXMIG* takes into account node heterogeneity and its balanced state is different from $\vec{w_{bl}}$. What is more, recall from Section 2 that *NIXMIG* terminates when $L_i(t) < thres_i \forall i \in [0, N]$, which means that a balanced state is every load distribution vector that satisfies this constraint. In Theorem 3, we prove the convergence of *NIXMIG* algorithm, along with the preconditions that need to hold for the system to reach an equilibrium.

Theorem 3. *Any load-balancing action using* NIXMIG *between a splitter node N_i and a helper node N_j leads the system to a balanced state, as long as the difference of the splitter's load from its thres value is by a constant of $1 - a$ bigger than the difference of the helper's load from its thres value, that is, $(L_i - thres_i)(1 - a) > L_j - thres_j$.*

Proof. In an atomic item exchange between two neighboring nodes, the load that will be moved from the splitter to the helper is $l = a(L_i - thres_i), 0 < a \leq 1$ (the case where $thres_i < L_i < overThres_i$ is covered by the general case for $a = 1$). The new loads are $L'_i = L_i - l$, $L'_j = L_j + l$.

Now, we have to show that the drop in the potential $\Delta\phi = \phi - \phi'$ caused by this load exchange is positive:

$$\begin{aligned}\Delta\phi &= (L_i - thres_i)^2 + (L_j - thres_j)^2 \\ &\quad - [(L_i - l) - thres_i]^2 - [(L_j + l) - thres_j]^2 \\ &= 2a(L_i - thres_i)[(L_i - thres_i)(1 - a) + (thres_j - L_j)].\end{aligned}$$

$\Delta\phi$ is the product of three terms. The first two are positive because $a \in (0, 1]$ (1) and L_i is overloaded ($L_i > thres_i$ (2)). So, the potential drop is positive if the third term is positive which happens if $(L_i - thres_i)(1 - a) > L_j - thres_j$. □

Corollary 1. *Any load-balancing action using* NIXMIG *between a splitter node N_i and a helper node N_j leads the system faster to an equilibrium as long as the helper is underloaded, that is, $L_j < thres_j$.*

Proof. The algorithm's convergence rate is faster as long as the selection of balancing partners ensures a larger drop in the $\Delta\phi$ value. If L_j is underloaded, then the third term of $\Delta\phi$ is larger (as a sum of two positive terms) compared to the case when L_j is overloaded. □

Corollary 1 is a special case of Theorem 3 that shows the importance for the algorithm's convergence of easily locating underloaded peers. In Corollary 2, we identify the moved load value l_{opt} that maximizes the algorithm's convergence rate leading the system quicker to an equilibrium. We define as $thdif_i = L_i - thres_i$ the difference of N_i's load from its $thres_i$ value.

Corollary 2. NIXMIG*'s optimum convergence rate is obtained when half of the difference of $thdif_i$ from $thdif_j$ is transferred from splitter Node N_i to helper Node N_j, that is, $l_{opt} = \frac{1}{2}(thdif_i - thdif_j)$.*

Proof. $\Delta\phi$ as a function of the moved load l is

$$\begin{aligned}\Delta\phi(l) &= (L_i - thres_i)^2 + (L_j - thres_j)^2 \\ &\quad - [(L_i - l) - thres_i]^2 - [(L_j + l) - thres_j]^2 \\ &= -2l^2 + 2(L_i - L_j + thres_j - thres_i)l.\end{aligned}$$

We notice that $\Delta\phi(l)$ is a quadratic function of l with coefficients $a = -2$, $b = 2(L_i - L_j + thres_j - thres_i)$, and $c = 0$. Because $a = -2 < 0$, $\Delta\phi(l)$ has a maximum point for

$$\begin{aligned}l_{opt} &= -\frac{b}{2a} = -\frac{-2(L_i - thres_i + thres_j - L_j)}{-4} \\ &= \frac{1}{2}[(L_i - thres_i) - (L_j - thres_j)] \\ &= \frac{1}{2}(thdif_i - thdif_j).\end{aligned}$$

□

In the case of a homogeneous splitter-helper pair ($thres_i = thres_j$) from Corollary 2, we notice that $l_{opt} = \frac{1}{2}(L_i - L_j)$, and thus $a_{opt} = \frac{1}{2}$.

5 Experimental Results

We now present a comprehensive simulation-based evaluation of our method on our own discrete event simulator written in Java. The time unit in our simulation is assumed to be equal to the time needed by a node to perform an operation with another node. Such operations include atomic item exchanges, lock requests, one-hop query routing messages, etc. For instance, a LOCALWAVE wave of $ttl = 5$ takes five time units to complete. For the remaining of the experimental section, we consider the time unit to be equal to one second. Starting off from a pure Skip Graph implementation, we incorporate our online balancing algorithms on top. By default, we assume a network size of 500 nodes, all of which are randomly chosen to initiate queries at any given time.

During the start-up phase, each node stores and indexes an equal portion of the data, $\frac{M}{N}$ keys. By default, we assume 50K keys exist in the system, thus each node is initially responsible for 100 keys.

Queries occur at rate $\lambda_r = 250$ queries/sec with exponentially distributed interarrival times in a 4,000 sec total simulation time. Each requester creates a range by choosing a starting value according to some distribution. The range of each created query is constant, and for the 50K setting it is equal to 100 keys (i.e., every range query requests 100 consecutive keys). The total network workload is a product of the query range with the query arrival rate, i.e., $w_{tot} = 250 \text{ queries/sec} \cdot 100 \text{ keys/query} = 25.000 \text{ keys/sec}$ (in every second, around 25K keys are requested in total). Recall from Section 4.3 that in the ideal balanced state of an homogeneous network, each node should get an equal load portion of $\bar{w} = \frac{w_{tot}}{N} = \frac{25.000}{500} = 50$ keys/sec.

In our experiments, we utilize several different distributions to simulate skew: A *zipfian* distribution, where the probability of a key i being asked is analogous to $i^{-\theta}$ and a *pulse* distribution, where a range of keys has a constant load and the rest of the keys are not requested. By altering the parameters of each distribution (e.g., the θ parameter, the width of the pulse, etc), we manage to create more or less skewed workloads to test our algorithms.

A node calculates its load using a simple moving average variable that stores the number of the keys it has served over a predefined time window. To minimize fluctuation caused by inadequate sampling, this time window is set to around 700 seconds. Since nodes in the beginning of the simulation do not have enough samples to estimate their load, we let the system stabilize on the input workload for 700 seconds without performing any balancing operation.

In the following, we plan to demonstrate the effectiveness of our protocol to minimize overloaded peers and create a load-balanced image of the system. As we mentioned before, we are interested in the resulting load distribution (in terms of overloaded servers and), the rate at which this is achieved (measured in seconds), as well as the cost measured in the number of exchanged messages and items.

TABLE 2
Exchanged Items and Messages and Completion Time for Various $overThres$ and ttl Values

ttl / overThres	1			3			**5**			10		
	keys	msgs	time	keys	msgs	time	keys	msgs	time	keys	msgs	time
160	62K	21K	92	74K	21K	93	85K	22K	170	134K	39K	554
280	68K	23K	114	79K	22K	120	80K	20K	117	80K	18K	147
400	66K	22K	116	71K	21K	108	**70K**	**19K**	**77**	80K	20K	173
500	69K	23K	147	71K	20K	105	71K	20K	105	80K	20K	166

During the experiments, *NIXMIG*'s parameters were set to the following values: $thres = 60$ keys/sec, $\alpha = \frac{1}{2}$, $ttl = 5$ nodes, and $overThres = 400$ keys/sec. The idea behind these parameters is the following: the $thres$ value is near the minimum theoretical value of $\bar{w} = 50$ keys/sec for which most of nodes eventually participate in the balancing procedure: the larger the $thres$ value, the easier (i.e., using less operations and bandwidth consumption) it is for *NIXMIG* to bring the system to its equilibrium making its comparison to other algorithms not fair. Furthermore, for homogeneous splitter-helper pairs, we have shown in Corollary 2 that $\alpha_{opt} = \frac{1}{2}$. With respect to ttl and $overThres$, in Table 2, we experimentally study *NIXMIG*'s behavior where in each column, we vary the ttl from 1 to 10 nodes and in each line, we vary the $overThres$ from 160 to 500 keys/sec. Table cells show the aggregated performance results for each ttl and $overThres$ combination. We notice that a combination of a ttl value of 5 nodes (third column) and an $overThres$ value of 400 keys/sec (third line) balances load quicker and cheaper compared to other $ttl - overThres$ combinations: smaller ttl values prohibit *NIXMIG* to examine a sufficient number of nodes, whereas a larger ttl value slows down the process due to more internode communication during locking procedures. The selected $overThres$ value enables *NIXMIG* to move the optimal amount of load during neighbor item exchanges: larger values lead to unnecessary load movement, whereas smaller values require more balancing operations.

5.1 Measuring the Effectiveness of *NIXMIG*

In the first set of experiments, we compare *NIXMIG*'s performance in a number of different input workloads against simple *MIG*, simple *NIX*, and the *Item Balancing* protocol (hence *IB*) proposed by Karger and Ruhl in [1]. *IB* was chosen as, in contrast with other systems, it applies the same minimal set of operations compared to *NIXMIG*: they both avoid the use of centralized load directories, item replication, and node virtualization (for a brief description of *IB* and a survey of similar systems refer to Section 6).

As input workload, we utilize *pulses* of variable width from 3 to 50 percent while keeping a constant surface (the pulse height is inversely proportional to its width) and constant surface zipfian workloads of variable θ from 1 to 4.5. In every case, nodes set their $thres$ value to 60 reqs/sec. This $thres$ value can also be seen as corresponding to 60 kb/sec bandwidth allocation, assuming that, for each request, 1 kb of data are transmitted. The simulation terminates when every node has dropped its load under its $thres$ value.

We have implemented the *IB* protocol setting $\varepsilon = \frac{1}{4}$ which provides the best balancing result. Moreover, probing messages occur with a rate of 0.1 msg/sec to keep the probing traffic low. In any case, we terminate the execution of *IB* when 50 seconds of simulation time have passed and no balancing action has occurred.

To apply *NIX*, we use Algorithm 4 and omit the REMOTEWAVE procedure: each overloaded node performs only a LOCALWAVE followed by a chain of *NIX* operations. For the wave direction selection, nodes use the following simple heuristic: new lock requests are sent toward the direction from which less lock requests were encountered. For *MIG*, nodes omit the LOCALWAVE of Algorithm 4 and directly proceed to the REMOTEWAVE procedure followed by a chain of *MIG* operations. In every situation, the load is balanced by moving most of the nodes inside the "hot" pulse area that is initially handled by a small number of overloaded nodes. In the *NIX* case, overloaded nodes iteratively shrink their range by offloading keys to their immediate neighbors, in the *MIG* case, remote nodes leave their place and rejoin inside the overloaded area and in the *NIXMIG* case, a combination of both these methods is adaptively utilized.

In Fig. 9, we compare *NIXMIG* against simple *NIX* simple *MIG* and the *IB* protocol. In the first graph, we present the completion time of each algorithm for the applied workloads. We notice that both *MIG* and *NIXMIG* balance load 4-8 times faster than *NIX*: for *NIX*, every node must accept and offload a large number of items for the balancing to succeed, whereas in the other two algorithms this is done in a more efficient way. Moreover, we notice that *NIXMIG* converges in almost half the time than *IB*.

Nevertheless, in the second graph, we notice that *MIG* is costly in terms of message exchanges, as it carelessly employs a large number of unnecessary node migrations. On the other hand, *NIXMIG* utilizes node migrations only when the load cannot be absorbed locally, thus keeping the number of required messages low compared to both *NIX*

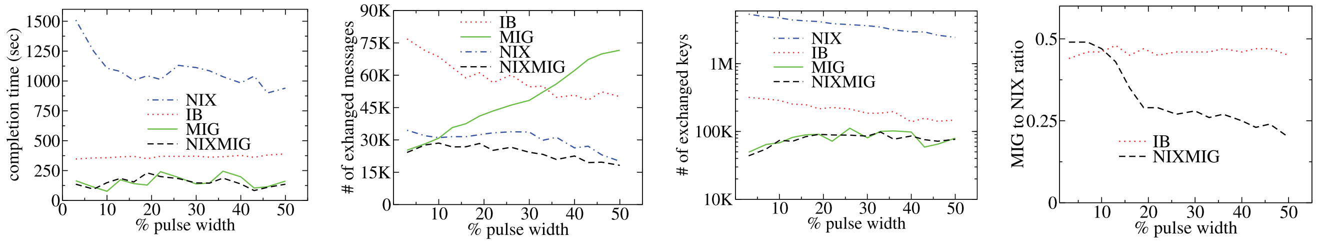

Fig. 9. Completion time, exchanged messages, and items and *MIG* to *NIX* ratio of *NIXMIG*, plain *NIX*, plain *MIG*, and *Item Balancing* for various pulse widths.

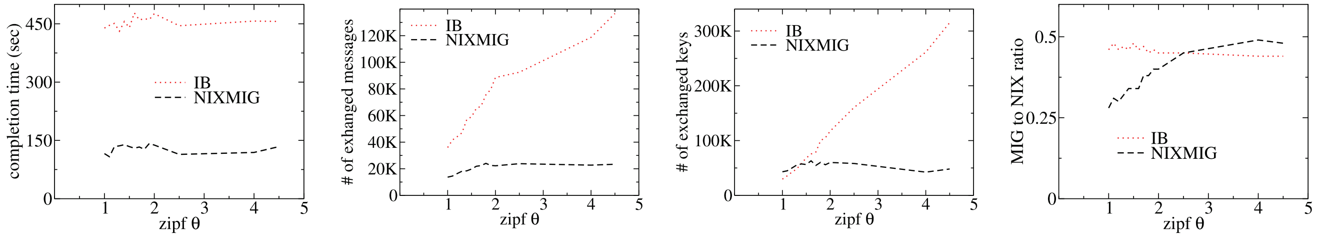

Fig. 10. Completion time, exchanged messages and items and *MIG* to *NIX* ratio of *NIXMIG* and *Item Balancing* for various zipfian θ values.

and *MIG*. In addition, *NIXMIG* requires less than half the messages compared to *IB*: *IB* requires a large number of probing messages, whereas *NIXMIG* uses the underloaded node location mechanism described in Section 4. Furthermore, the number of required messages in the *IB* algorithm increases more due to the fact that mostly node migrations are performed, as its *MIG* to *NIX* ratio is near 0.5 (see the fourth graph).

In the third graph, we notice that *NIX* requires two orders of magnitude more item exchanges than *MIG* and *NIXMIG* due to the iterative key transfer procedure. What is more, *NIXMIG* requires roughly the same number of item exchanges compared to *MIG*. *NIXMIG* outperforms *IB* whereas in skewed workloads *NIXMIG* exchanges one third of the items compared to *IB*: the cooperative nature of *NIXMIG* minimizes unnecessary load movement (thus item exchanges) back and forth, unlike *IB* where each node acts on its own. We observe that the *IB*'s number of exchanged messages and items drops when the workload is less skewed: *IB* performs less balancing actions, as it cannot easily locate nodes that their load differs by a fraction of ε.

Finally, in the fourth graph, we present *NIXMIG*'s and *IB*'s ratio of migrations to simple neighboring item exchange operations for various pulse widths. Here, we notice *NIXMIG*'s workload adaptivity: in extremely skewed workloads of 3-5 percent pulse widths mostly node migrations are used (recall from Algorithm 2 that each migration requires two neighboring item exchanges, thus the ratio in plain migrations is 0.5). When the pulse's width is increased, the ratio drops as load is absorbed using more neighboring item exchanges and costly remote migrations are avoided. On the contrary, *IB* most of the times carelessly employs node migrations.

These experiments confirm *NIXMIG*'s adaptivity to an arbitrary workload, as it identifies the most effective balancing action, combining the advantages and avoiding the disadvantages of both plain remote migrations and plain neighboring item exchanges. We continue our experimental analysis with a more thorough comparison of *NIXMIG* against *IB*.

In Fig. 11, we present a system's load snapshot after 100 seconds for the two algorithms for a 3 percent pulse. We notice that, unlike *IB* (dotted line), *NIXMIG* (solid line) has successfully dropped almost every node's load under its *thres* value (horizontal red line). Moreover, in Fig. 12, we present the variation of exchanged messages during time for the *NIXMIG* and the *IB* algorithm. We notice that *NIXMIG* constantly performs less message exchanges than *IB*. What is more, in the *IB* algorithm, we notice the constant traffic posed by the random probing messages.

In Fig. 10, we present the performance results of *NIXMIG* against *IB* for the zipfian setting. In this situation, the workload's skew increases as the θ parameter increases unlike the pulse setting where the skew decreases as the pulse width increases. In the first graph, we notice that *NIXMIG*'s completion time is similar to the one in the pulse setting. On the other hand, *IB*'s completion time increases compared to the respective completion time for the pulse setting: in the zipfian case, the load is spread more uniformly compared to the pulse setting, making it harder for *IB* to identify load imbalances. In any case, *NIXMIG* is three times faster than *IB*. In the second graph, we notice that *NIXMIG* requires a constant number of messages with a slight drop in the less skewed workload area, as more neighboring item exchanges are performed. On the other hand, *IB* requires constantly more messages due to the reasons mentioned in the previous paragraph. In the workloads with $\theta > 3$, *NIXMIG* requires one sixth of the messages that *IB* requires. In the third graph, we observe that *NIXMIG*'s and *IB*'s behavior in item exchanges is similar as in the pulse setting. *NIXMIG* performs more item exchanges than *IB* in the less skewed workloads of

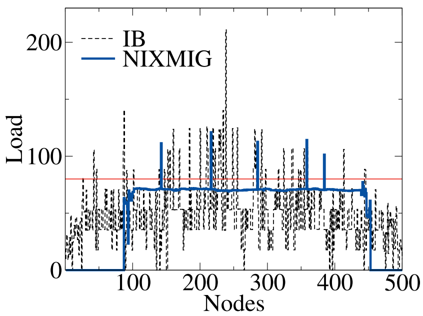

Fig. 11. Load snapshot at $t = 800$ sec for a 3 percent pulse.

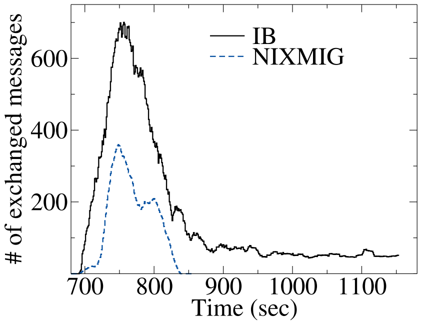

Fig. 12. Number of exchanged messages during time for a 3 percent pulse.

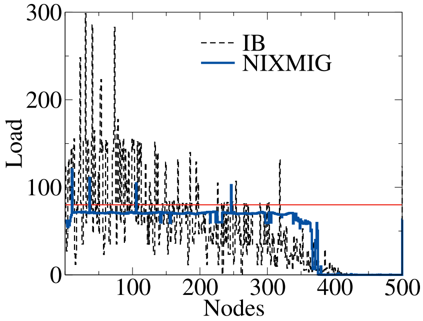

Fig. 13. Load snapshot at $t = 800$ sec for a zipfian of $\theta = 4.5$.

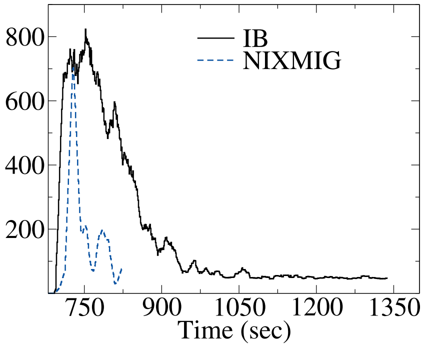

Fig. 14. Number of exchanged messages during time for a zipfian of $\theta = 4.5$.

$\theta < 1.6$, as it performs more neighboring item exchanges. In more skewed situations, *NIXMIG* performs one third less item exchanges compared to *IB*. The last graph shows the adaptivity of *NIXMIG* where more migrations are employed in more skewed workloads, whereas *IB* performs mostly migrations in any case. Finally, in Figs. 13 and 14, we present a load snapshot after 100 seconds and the variation of the message exchanges during time, respectively, for a zipfian workload of $\theta = 4.5$. The same behavior as in the pulse setting is observed: *NIXMIG* balances load faster and uses constantly less messages than *IB*.

5.2 *NIXMIG* Scalability

In the following experiment, we study *NIXMIG*'s behavior when the number of participating peers increases. Nodes share a total of 5 million keys and the applied workload is a 10 percent pulse. We vary the network size from 500 to 50,000 nodes. Table 3 presents our findings compared to the 500-node setting (i.e., we register the increase in measurements compared to the 500-node setting result). We notice that the number of messages increases linearly compared to the number of nodes. Moreover, we also notice a slow linear increase of the completion time: even for a 100 times larger network the algorithm terminates only three times slower. This happens because multiple *NIXMIG* executions are performed in parallel. Finally, the number of exchanged items remains constant: this shows that *NIXMIG* does not perform unnecessary item transfers when the network size increases.

5.3 *NIXMIG* Performance under Dynamic Workload

We now present results showing the performance of our *NIXMIG* method when the workload suddenly changes its skew. We assume an initial pulse load of width 12 percent and height 430 req/sec where items [10,000, 16,000] are requested. This pulse suddenly moves at time $t = 850$ sec to items [34,000, 40,000]. Note that this is an extreme scenario, since the skew changes completely and abruptly at this time.

TABLE 3
Ratio of Exchanged Messages, Completion Time, and Transferred Items for Various Network Sizes Compared to a 500-Node Setting

Nodes ratio	Messages ratio	Completion time ratio	Items ratio
10	13	1.2	1.05
50	102	2.2	0.96
100	208	2.8	0.98

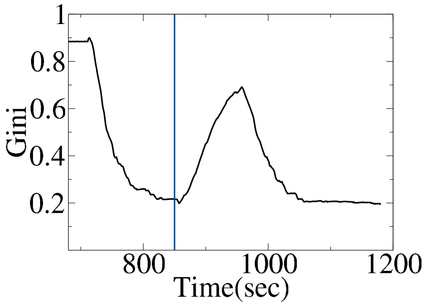

Fig. 15. The Gini variation for the dynamic setting.

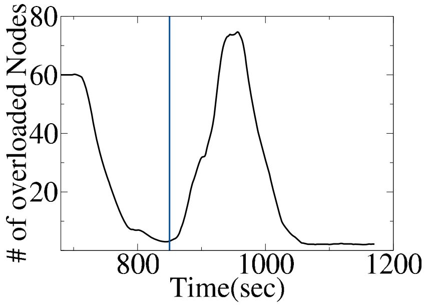

Fig. 16. Number of overloaded peers over time, dynamic setting

Fig. 15 shows the variation of the Gini [27] coefficient over time, respectively. Gini values range between 0 and 1, where 0 corresponds to perfect equality and 1 corresponds to the theoretic case of an infinite population with only one individual having a nonzero value. Recent work [28] proposed its use as a load-balancing metric. Assuming our population comprises of the number of received requests by each node, we calculate the value of G as an index of load distribution among servers. Note here that a low value of G is a strong indication that load is equally distributed among them, but does not necessarily imply that this load is low. Fig. 16 shows the number of overloaded peers during time. We notice that both metrics are affected immediately after the change in load occurs, nevertheless, *NIXMIG* works over this new situation and manages to reduce both quantities: The Gini coefficient increases when the pulse changes, but *NIXMIG* manages to keep it well under 0.9 (which is its initial value) until it is dropped near 0.2 in the balanced state. Moreover, the number of overloaded nodes slightly passes the initial value of 60 until it is minimized by *NIXMIG*. In Figs. 17 and 18, we present the number of exchanged messages and items during the simulation time, respectively, where we notice *NIXMIG*'s cost-effective balancing: the

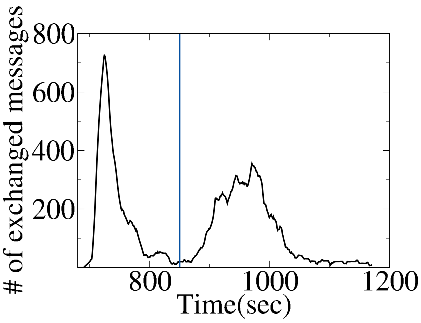

Fig. 17. Number of exchanged messages over time, dynamic setting.

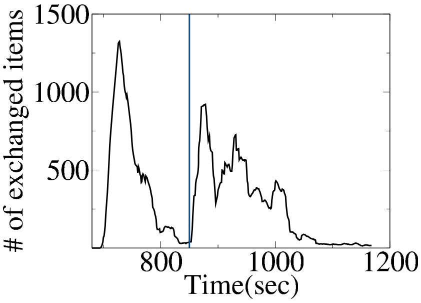

Fig. 18. Number of exchanged keys over time, dynamic setting.

number messages does not exceed 700 msg/sec (in a 500-node setting) whereas the number of exchanged keys stays under 1,200 keys/sec (in a 50K setting). Finally, the reason that the convergence time is documented to be larger than that of handling a single pulse is obvious: The very sudden change in skew forces the invalidation of many already performed balance operations and nodes with no load problem suddenly become very overloaded.

Fig. 19 shows the progress of the balancing process in time: First, at time $t = 800$ sec, after 100 sec of balancing time (recall that *NIXMIG* started at $t = 700$ sec), just before the query load changes, we show that *NIXMIG* is very close to balancing the load. This is obvious from the improvement shown at $t = 850$ sec, where the old pulse diminishes and the new one appears. After this point, the newly overloaded nodes start shedding load to their neighbors (hence the snapshot picture for time $t = 1{,}000$ sec). Finally, *NIXMIG* totally balances load (last image).

In the following experiment, we utilize the previously described 12 percent pulses and we modify the position in the ID-space of the second pulse along with the time we trigger the sudden pulse move. At every execution, the initial pulse is applied over items [10K,16K]. We present our results in Table 4. Each column represents an increase in the second pulse's distance from the initial one using a 10 percent step of 5K keys and each line an increase in the time we trigger the sudden change using a step of 20 seconds. We measure the total number of exchanged items along with the time it took for *NIXMIG* to balance both workloads. We notice that as the new pulse's distance increases in each column, *NIXMIG* performs more key exchanges and takes more time to complete. The same increase in both metrics is noticeable when, in each row we increase the time we trigger the second pulse. Nevertheless, even in the worst case where the second pulse is triggered at $t = 820$ sec (60 sec later compared to the 760 sec case) and in the [30K-36K] position (40 percent further than in the [15-21K] case), *NIXMIG*'s performance is not significantly degraded: only 30 percent more items are transferred and balancing is 2.2 times slower compared to the 760 sec and [15-21K] combination.

5.4 Smart Remote Node Placement Mechanism

Next, we study the effect of minimizing the number of exchanged items during load transfers caused by migrations by taking into account load skew, as presented in Section 4.2.

More specific, we compare our smart, skew-aware, remote node placement mechanism (hence *SmartRNP*) to the random case (hence *RandomRNP*) where nodes are randomly placed and to the situation where an "adversary" places remote nodes so as to maximize the number of transferred items (hence *AdversarialRNP*). As our input workload, we consider a number of (around 20) popular ranges in the ID space for which all keys are requested, whereas all other keys are not queried at all. We vary the width of each popular range so that all "hot" ranges occupy from 10 to 50 percent of the ID space. In Table 5, we present the effect of *SmartRNP* for various workloads (first column): in the second column, we depict the number of exchanged keys due to a *MIG* operation effectively minimized by *SmartRNP*, in the third column, we present the ratio of *SmartRNP* to *RandomRNP* key movement and in the fourth column the ratio of *SmartRNP* to *AdversarialRNP* key movement. We notice that for highly skewed distributions of 10 percent, *SmartRNP* exchanges only 47 percent items compared to *RandomRNP* and 16 percent compared to the adversarial case, while this ratio increases (i.e., *SmartRNP* number of transferred items gets closer to the number of *RandomRNP* and *AdversarialRNP*) for less skewed distributions. This is explained by Fig. 8: the larger the skew, the larger the difference of $|r_2 - r_b|$ from $|r_1 - r_b|$ making node A's decision more critical for the algorithm's performance. What is more, we notice that *RandomRNP* performs constantly better than *AdversarialRNP* (worst-case scenario) in terms of transferred items, as with high probability half of its decisions are "correct" (i.e., they minimize key transfer).

5.5 *NIXMIG* in Realistic Workloads

In the following experiment, we utilize a publicly available data set from AOL that contains 20 million queries from over 650,000 users over a three-month period.[2] The data set comprises around 3.7 million distinct keywords of varying popularity which are initially equally divided among 500 nodes. By measuring the number of occurrences of each keyword in the searches, we calculated the query frequency distribution: Clients are generating prefix queries of variable length (e.g., "goo*", "googl*", etc.) based on the calculated frequencies. Prefix queries, typically used by search engines to provide the autocomplete feature among others, are translated to range queries in our setup. Compared to our previous reported experiments, nodes now store almost 100 times more objects ,while the query workload follows a realistic distribution, with the selectivity of the range queries taking many possible values.

Table 6 presents the results for variable prefix lengths. In all cases, *NIXMIG* balances load fast and under 150 sec, a result that is well inline with our previous findings (see Section 5.1—first graphs of Figs. 9 and 10). *NIXMIG* adapts its operations to the applied workload: When the prefix length

[2] http://techcrunch.com/2006/08/06/aol-proudly-releases-massive-amounts-of-user-search-data/.

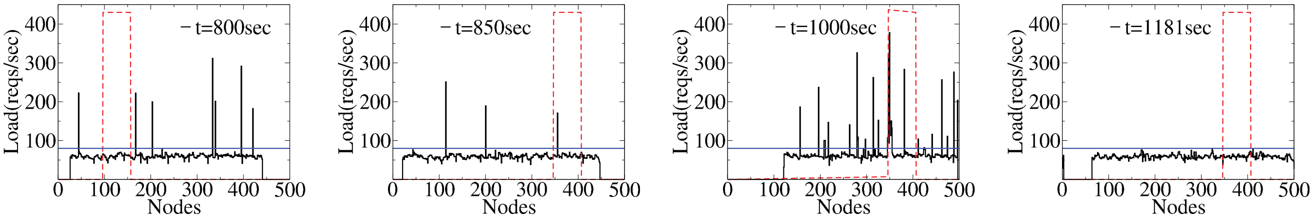

Fig. 19. Load distribution as it progresses in time: Snapshot taken at times $t = 800, 850, 1{,}000, 1{,}181$ sec.

increases, *NIXMIG* applies more migrations, increasing the number of exchanged messages, and the *MIG* to *NIX* ratio. This happens because the prefix length affects the number of matched objects and thus the range query size ("goog*" returns more results than "googl*"). Queries of larger prefix lengths are served by a smaller number of nodes. Consequently, these nodes are excessively overloaded and request more migrations for a faster load alleviation.

6 Related Work

DHTs such as Chord [29] tackle load-balancing issues by applying hash functions to the data, thus destroying their locality and allowing only exact match queries. The need to preserve the order of the indexed keys to achieve efficient lookups in range-queriable structures prevents us from using simple hash functions to uniformly balance load among nodes. Therefore, we can categorize the available approaches into two broad orthogonal groups: data replication and data migration. Data replication alleviates the overloaded nodes by providing extra targets for the incoming requests. Data migration requires actual data transfers between nodes in order to balance load. *NIXMIG* falls into the data migration category. Data replication, with its relative advantages and disadvantages, is applied in conjunction with data migration to further improve performance and fault tolerance. For instance, one sole replica of an overloaded node's items can effectively drop by half its load (provided that the routing protocol redirects half of the requests to the replica node) but on the other hand, both updates and query routing are more difficult to handle.

In the case of peer-to-peer systems, data migration can be further classified in the node virtualization [9], [10], [11], [12], [13], [14] and one ID per server [1], [14], [15], [16], [17], [18], [19], [20], [21] strategies. In the former, every actual server can host a number of virtual servers which are responsible for small disjoint ranges of its items, and balancing is performed by moving virtual servers between actual ones. It has been widely used because of its ease of use (virtual servers can be concurrent threads of a DHT implementation running on the same machine), but its main drawback is the increased bandwidth and memory consumption caused by the maintenance of numerous routing tables (the number of open network connections gets multiplied by a factor of $\Omega(logn)$ [30]). Godfrey and Stoica [31] tackled this by placing virtual servers owned by an actual one "near" themselves in the ID space but they make the assumption that the load is uniformly distributed in the identifier space, something that does not always hold in order-preserving structures. What is more, it has been shown that with only one ID per actual server balancing results are the same as in the case of node virtualization (see related work of [30]). *NIXMIG* uses one ID per server.

One ID per server approaches. In the work of Karger and Ruhl [1], a *work-stealing* technique is applied: peers randomly probe distant nodes and compare their loads. If the load of the less loaded node is smaller than a fraction of $0 < \varepsilon < \frac{1}{4}$ of the more loaded node's load then a migration or an neighboring item exchange is performed. The drawbacks of this method were shown in Section 5.1. Moreover, they do not present analytical results of their algorithm applied to a distributed system. Ganesan et al. [15] propose a balancing mechanism that works on top of a Skip Graph system [23]. Each node is aware of an ordered set of load thresholds and is responsible for periodically updating a shared directory with its current load. When its load crosses a boundary of this ordered set, it contacts the directory to locate the next more loaded node and performs load exchange with it. Its main drawback is the costly maintenance of this load directory. Aspnes et al. [16] propose a second layer on top of a simple Skip Graph, the buckets

TABLE 4
Exchanged Items and Completion Time for the Dynamic Setting for Various Trigger Times and New Pulse Positions

Position Time	[15K-21K]		[20K-26K]		[25K-31K]		[30K-36K]	
	keys	time	keys	time	keys	time	keys	time
760	107K	183	120K	238	125K	303	130K	308
780	109K	201	124K	258	129K	374	132K	315
800	128K	265	129K	268	132K	291	135K	321
820	130K	334	132K	352	138K	380	140K	406

TABLE 5
Ratio of Transferred Items Using *SmartRNP* versus *RandomRNP* and *AdversarialRNP* for Various "hot" Range Percentages

ID space % of "hot" range	*SmartRNP* # of transf. items	Ratio of transf. items compared to:	
		RandomRNP	*AdversarialRNP*
10	5.3K	0.47	0.16
20	6.8K	0.52	0.24
30	7.0K	0.59	0.36
40	7.2K	0.66	0.49
50	6.7K	0.71	0.53

TABLE 6
Completion Time, Number of Exchanged Messages, and *MIG* to *NIX* Ratio of *NIXMIG* for Various Prefix Lengths

Prefix length	time(sec)	messages	MIG to NIX ratio
4	134	16.4K	0.30
5	140	17.9K	0.32
10	150	18.5K	0.34

layer. Each bucket contains a number of ordered items and each server may have several buckets. During balancing procedures, overloaded nodes move buckets to their immediate neighbors (similar to a *NIX* operation). They address skewed data distributions with a list of free nodes which can migrate in an area to absorb excess load. The main drawback of this scheme is the requirement of a list of free nodes: This luxury cannot be considered trivial in actual deployments. In Mercury [17], probing and node migration is used to solve load-balancing problems. Nodes use a random sampling procedure with probe messages to calculate the average network load. When their load is above or below the average network load, they initiate balancing actions. The authors state that their load-balancing scheme is similar to *IB* [1]: their only difference is that they minimize flooding during probing as they perform a clever and selective way of disseminating load information. Similar to Mercury is the *HiGLOB* framework [18]: each node maintains a list of load information about nonoverlapping regions of the key space, and if it detects imbalances, it performs load exchanges following the *IB* paradigm. In Armada [14], load-balancing is performed with a hash function responsible for placing items into nodes that knows in advance the distribution of items in the ID space. Armada can handle only static workloads, unlike NIXMIG's ability to deal with dynamic workloads. Shen and Xu [19], [20] maintain matchings of overloaded to underloaded peers: balancing is performed by moving "hot" items and placing doubly linked pointers both to the source (overloaded peer) and the destination (underloaded peer) of the moved item. The drawback of this method is that during lookups, the overloaded peer will still be contacted, as it is still responsible for this "hot" item. In chordal graphs [21], balancing is performed by a process called "free drifting" which is actually a *NIX* operation that has the disadvantages described in Section 5.1.

Node virtualization approaches. The idea of virtual servers for load-balancing in peer-to-peer systems was initially proposed in CFS [9]. Based on this idea, Rao et al. [10] proposed three load-balancing algorithms (One to One, One to Many, and Many to Many) which were extended by Surana et al. [11] for heterogeneous peer-to-peer systems with churn. In the first case, an overloaded node contacts one node at random (as in the work of Karger and Ruhl [1]) while in the second case it contacts numerous nodes before it takes a balancing decision. The third case is similar to the approach used by Ganesan et al. [15]: a distributed directory with load information is maintained and contacted by overloaded peers before any balancing decision is taken. Zhu and Hu [12] also build and maintain a distributed load directory in the form of a k-ary tree structure that is stored in the overlay. This directory is used by nodes to detect load imbalances and to find suitable overloaded-underloaded node pairs. Chen and Tsai [13] use the general assignment problem (a particular case of a linear programming problem) to assign virtual to actual nodes: they make an initial estimation using the ant system heuristic which afterwards is refined using the descent local search algorithm. This procedure is iteratively applied until a solution is reached. In Armada [14], the authors use virtual servers for balancing purposes but they do not provide details about their specific implementation.

7 Conclusions

In this paper, we evaluate the performance in terms of bandwidth cost and convergence speed of balancing range-queriable data structures using successive item exchanges and node migrations. Our theoretical analysis and extensive experimental results show that none of these methods by itself is capable of efficiently balancing arbitrary workloads: Neighbor item exchanges are expensive in terms of item transfers and slow in terms of convergence speed, whereas node migrations are fast but costly in terms of message exchange. Our method, *NIXMIG*, is a hybrid approach that adaptively decides the appropriate balancing action and provably converges to a balanced state. Load moves in a "wave-like" fashion until it is absorbed by underloaded nodes, while node migration is triggered only when necessary. Our results show that *NIXMIG* can be three times faster, while requiring only one sixth and one third of message and item exchanges, respectively, compared to an existing load-balancing algorithm proposed by Karger and Ruhl [1] to bring the system in a balanced state under a variety of skewed, dynamic, and realistic workloads.

References

[1] D.R. Karger and M. Ruhl, "Simple Efficient Load-Balancing Algorithms for Peer-to-Peer Systems," *Theory of Computing Systems,* vol. 39, pp. 787-804, Nov. 2006.

[2] M. Cha, H. Kwak, P. Rodriguez, Y. Ahn, and S. Moon, "I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System," *Proc. Seventh ACM SIGCOMM Conf. Internet Measurement,* 2007.

[3] S. Sen and J. Wong, "Analyzing Peer-to-Peer Traffic Across Large Networks," *Proc. ACM SIGCOMM Internet Measurements Workshop,* 2002.

[4] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites," *Proc. Int'l Conf. World Wide Web (WWW),* 2002.

[5] Y. Vigfusson, A. Silberstein, B.F. Cooper, and R. Fonseca, "Adaptively Parallelizing Distributed Range Queries," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2009.

[6] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-Peer Support for Massively Multiplayer Games," *Proc. IEEE INFOCOM,* 2004.

[7] Q. Luo and J.F. Naughton, "Form-Based Proxy Caching for Database-Backed Web Sites," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 191-200, 2001.

[8] S.Y. Lee, T.W. Ling, and H.-G. Li, "Hierarchical Compact Cube for Range-Max Queries," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 232-241, 2000.

[9] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS," *ACM SIGOPS Operating Systems Rev.,* vol. 35, no. 5, pp. 202-215, 2001.

[10] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems," *Proc. Second Int'l Workshop Peer-to-Peer Systems (IPTPS),* pp. 68-79, 2003.

[11] S. Surana, B. Godfrey, K. Lakshminarayanan, R. Karp, and I. Stoica, "Load Balancing in Dynamic Structured Peer-to-Peer Systems," *Performance Evaluation,* vol. 63, no. 3, pp. 217-240, 2006.

[12] Y. Zhu and Y. Hu, "Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 16, no. 4, pp. 349-361, Apr. 2005.

[13] C. Chen and K. Tsai, "The Server Reassignment Problem for Load Balancing in Structured P2P Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 19, no. 2, pp. 234-246, Feb. 2008.

[14] D. Li, J. Cao, X. Lu, and K. Chen, "Efficient Range Query Processing in Peer-to-Peer Systems," *IEEE Trans. Knowledge and Data Eng.,* vol. 21, no. 1, pp. 78-91, Jan. 2009.

[15] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 444-455, 2004.

[16] J. Aspnes, J. Kirsch, and A. Krishnamurthy, "Load Balancing and Locality in Range-Queriable Data Structures," *Proc. Ann. ACM Symp. Principles of Distributed Computing (PODC),* pp. 115-124, 2004.

[17] A.R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries," *ACM SIGCOMM Computer Comm. Rev.,* vol. 34, no. 4, pp. 353-366, 2004.

[18] Q.H. Vu, B.C. Ooi, M. Rinard, and K.L. Tan, "Histogram-Based Global Load Balancing in Structured Peer-to-Peer Systems," *IEEE Trans. Knowledge and Data Eng.,* vol. 21, no. 4, pp. 595-608, Apr. 2009.

[19] H. Shen and C.Z. Xu, "Locality-Aware and Churn-Resilient Load-Balancing Algorithms in Structured Peer-to-Peer Networks," *IEEE Trans. Parallel and Distributed Systems,* vol. 18, no. 6, pp. 849-862, June 2007.

[20] H. Shen and C. Xu, "Hash-Based Proximity Clustering for Efficient Load Balancing in Heterogeneous DHT Networks," *J. Parallel and Distributed Computing,* vol. 68, no. 5, pp. 686-702, 2008.

[21] Y.J. Joung, "Approaching Neighbor Proximity and Load Balance for Range Query in P2P Networks," *Computer Networks,* vol. 52, no. 7, pp. 1451-1472, 2008.

[22] I. Konstantinou, D. Tsoumakos, and N. Koziris, "Measuring the Cost of Online Load-Balancing in Distributed Range-Queriable Systems," *Proc. IEEE Ninth Int'l Conf. Peer-to-Peer Computing (P2P),* pp. 135-138, 2009.

[23] J. Aspnes and G. Shah, "Skip Graphs," *ACM Trans. Algorithms,* vol. 3, p. 37, 2007.

[24] P. Berenbrink, T. Friedetzky, and Z. Hu, "A New Analytical Method for Parallel, Diffusion-Type Load Balancing," *J. Parallel and Distributed Computing,* vol. 69, no. 1, pp. 54-61, 2009.

[25] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel and Distributed Computing,* vol. 7, no. 2, pp. 279-301, 1989.

[26] B. Ghosh and S. Muthukrishnan, "Dynamic Load Balancing by Random Matchings," *J. Computer and System Sciences,* vol. 53, no. 3, pp. 357-370, Dec. 1996.

[27] C. Damgaard and J. Weiner, "Describing Inequality in Plant Size or Fecundity," *Ecology,* vol. 81, no. 4, pp. 1139-1142, 2000.

[28] T. Pitoura, N. Ntarmos, and P. Triantafillou, "Replication, Load Balancing and Efficient Range Query Processing in DHTs," *Proc. 10th Int'l Conf. Extending Database Technology (EDBT),* pp. 131-148, 2006.

[29] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *SIGCOMM Comput. Comm. Rev.,* vol. 31, pp. 149-160, 2001.

[30] G.S. Manku, "Balanced Binary Trees for ID Management and Load Balance in Distributed Hash Tables," *Proc. Ann. ACM Symp. Principles of Distributed Computing (PODC),* pp. 197-205, 2004.

[31] P.B. Godfrey and I. Stoica, "Heterogeneity and Load Balance in Distributed Hash Tables," *Proc. IEEE INFOCOM,* vol. 1, pp. 596-606, 2005.

Ioannis Konstantinou received the diploma in electrical and computer engineering from the National Technical University of Athens (NTUA) in 2004 and the MSc degree in techno-economic systems from NTUA in 2007. He currently works at the Computing Systems Laboratory of the School of Electrical and Computer Engineering of NTUA, where he is working toward the PhD degree in the field of distributed systems, peer-to-peer technologies, and cloud computing.

Dimitrios Tsoumakos received the diploma in electrical and computer engineering from the National Technical University of Athens (NTUA) in 1999, and the MSc and PhD degrees from the University of Maryland, in 2002 and 2006, respectively. He joined the graduate program in computer sciences at the University of Maryland in 2000. He currently holds a senior researcher position in the Computing Systems Laboratory of the School of Electrical and Computer Engineering of the NTUA.

Nectarios Koziris is an associate professor at the School of Electrical and Computer Engineering, National Technical University of Athens. His research interests include parallel architectures, scalable distributed systems, and data and resource management for large-scale Internet systems. He has published more than 90 papers in international journals and in the proceedings of international conferences. He is a recipient of the IEEE IPDPS 2001 best paper award. He served as a chair and Program Committee member in various IEEE/ACM conferences. He is a member of the IEEE and the IEEE Computer Society, a senior member of the ACM, and chairs the Greek IEEE CS Chapter. He also serves as the vice chairman for the Greek Research and Education Network.