Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems

Yingwu Zhu, *Student Member*, *IEEE*, and Yiming Hu, *Senior Member*, *IEEE*

Abstract—Many solutions have been proposed to tackle the load balancing issue in DHT-based P2P systems. However, all these solutions either ignore the heterogeneity nature of the system, or reassign loads among nodes without considering proximity relationships, or both. In this paper, we present an efficient, proximity-aware load balancing scheme by using the concept of virtual servers. To the best of our knowledge, this is the first work to use proximity information in load balancing. In particular, our main contributions are: 1) Relying on a self-organized, fully distributed k-ary tree structure constructed on top of a DHT, load balance is achieved by aligning those two skews in load distribution and node capacity inherent in P2P systems—that is, have higher capacity nodes carry more loads; 2) proximity information is used to guide virtual server reassignments such that virtual servers are reassigned and transferred between physically close heavily loaded nodes and lightly loaded nodes, thereby minimizing the load movement cost and allowing load balancing to perform efficiently; and 3) our simulations show that our proximity-aware load balancing scheme reduces the load movement cost by 11-65 percent for all the combinations of two representative network topologies, two node capacity profiles, and two load distributions of virtual servers. Moreover, we achieve virtual server reassignments in $O(\log N)$ time.

Index Terms—Proximity-aware, peer-to-peer, virtual server, load balancing.

1 Introduction

DHT-based P2P systems such as Chord [2], Pastry [3], Tapestry [4], and CAN [5], offer a distributed hash table (DHT) abstraction for object storage and retrieval.[1] Due to the theoretical approach taken in these DHTs, they assume that nodes in the system are uniform in resources, e.g., network bandwidth and storage. By providing such a simple and homogeneous abstraction, while theoretically elegant, these DHTs have two main limitations. First, simply resorting to the uniformity of the hash function used to generate object IDs in DHTs does not produce perfect load balance. It could result in an $O(\log N)$ imbalance factor in the number of objects stored at a peer node, where N is the number of nodes in the system. Second, with a homogeneous structure overlay network, they ignore the heterogeneity nature of P2P systems. Recent measurement studies (e.g., [6]) have shown that node capabilities (in terms of bandwidth, storage, and CPU) are very skewed in deployed P2P systems such as Gnutella.

The primary goal of P2P systems is to harness all available resources (e.g., storage, bandwidth, and CPU) in the P2P network so that users can access all available objects *efficiently*. From the P2P system perspective, "efficiently" is interpreted as striving to ensure fair load distribution among all peer nodes. As a result, achieving load balance is of fundamental importance in a DHT, due to the assumption that nodes are supposed to be uniform in resources, the resulting $O(\log N)$ load imbalance by a random choice of object IDs, and the fact that heterogeneous capabilities prevail among the nodes.

Many solutions [2], [7], [8], [9] have been proposed to tackle the load balancing issue in DHT-based P2P systems (see Section 2 for more details). However, existing load balancing approaches have some limitations in our opinion. They either ignore the heterogeneity of node capabilities, or transfer loads between nodes without considering proximity relationships, or both. In this paper, we present a proximity-aware load balancing scheme by using the concept of *virtual servers* previously proposed in [7]. The goals of our scheme are not only to ensure fair load distribution over nodes proportional to their capacities, but also to minimize the load-balancing cost (e.g., bandwidth consumption due to load movement) by transferring virtual servers (or loads) between heavily loaded nodes and lightly loaded nodes in a *proximity-aware* fashion. We achieve the latter goal by using proximity information to guide virtual server reassignments, as will be discussed later in this paper. There are two main advantages of a proximity-aware load balancing scheme. First, from the system perspective, a load balancing scheme bearing network proximity in mind can reduce the bandwidth consumption (e.g., bisection backbone bandwidth) dedicated to load movement. Second, it can avoid transferring loads across high-latency wide-area links, thereby enabling fast convergence on load balance and quick response to load imbalance. To the best of our knowledge, this is the first work that approaches the load balancing issue in a proximity-aware manner.

In particular, we make the following contributions:

1. Relying on a self-organized, fully distributed k-ary tree structure constructed on top of a DHT, load balance is achieved by aligning those two skews in

1. They all provide two basic operations: $DHT\_put(key, object)$ stores an object into the DHT with a key and $DHT\_get(key)$ retrieves a corresponding object from the DHT given a key.

• *The authors are with the Department of Electrical & Computer Engineering and Computer Science, University of Cincinnati, PO Box 210030, Cincinnati, OH 45221-0030. E-mail: {zhuy, yhu}@ececs.uc.edu.*

Manuscript received 12 Feb. 2004; revised 8 July 2004; accepted 10 Sept. 2004; published online 23 Feb. 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0047-0204.

load distribution and node capacity inherent in P2P systems—that is, have higher capacity nodes carry more loads.
2. Proximity information is used to guide virtual server reassignments such that virtual servers are reassigned and transferred between physically close heavily loaded nodes and lightly loaded nodes, thereby minimizing the load movement cost and allowing load balancing to perform efficiently.
3. Our simulations show that our proximity-aware load balancing scheme reduces the load movement cost by 11-65 percent for all the combinations of two representative network topologies, two node capacity profiles, and two load distributions of virtual servers. Moreover, we achieve virtual server reassignments in $O(\log N)$ time.

The remainder of the paper is organized as follows: Section 2 provides a survey of related work. Section 3 describes the system design of our load balancing scheme. Section 4 presents the proximity-aware load balancing scheme. In Section 5, we evaluate the load balancing scheme using detailed simulations. Finally, we conclude in Section 6.

2 Related Work

DHT-based P2P systems [2], [3], [4], [5] address the load balancing issue in a rather naive way, by simply resorting to the uniformity of the hash function used to generate object IDs. Such a random choice of object IDs, however, does not produce perfect load balance. It could result in an $O(\log N)$ load imbalance. Moreover, such systems ignore the heterogeneous nature of the system while measurement studies (e.g., [6]) have shown the prevalence of heterogeneity in deployed P2P systems.

Many load balancing approaches have been proposed for DHT-based P2P systems. Chord [2] was the first to propose the concept of virtual servers to address the load balancing issue by having each node simulate a logarithmic number of virtual servers. While theoretically elegant, virtual servers do not completely solve the load balancing issue. Also, Chord assumes nodes are homogeneous. CFS [7] accounts for node heterogeneity by having each node host some number of virtual servers in proportion to its capacity. When a node becomes overloaded, the node simply sheds part of its loads by removing some of its virtual servers. However, simply deleting virtual servers by overloaded nodes may make other nodes become overloaded, thereby causing a load thrashing problem.

Rao et al. [8] propose three simple load balancing schemes for DHT-based P2P systems: *One-to-One*, *One-to-Many*, and *Many-to-Many*. The basic idea behind their schemes is that virtual servers are moved from heavy nodes to light nodes for load balancing. In particular, *One-to-Many* achieves load balance by allowing each heavy node to contact a randomly chosen *directory* node. The *directory* node stores the load information of a random set of light nodes and performs virtual server reassignments from the heavy node to some of its light nodes. *Many-to-Many* achieves load balance by storing load information of both heavy nodes and light nodes in a number of *directory* nodes. The *directory* nodes periodically schedule reassignments of virtual servers. Recent work by Godfrey et al. [9] extends *One-to-Many* (for emergency load balancing of one particularly overloaded node) and *Many-to-Many* (for periodic load balancing of all nodes) to *dynamic* structured P2P systems. This is the first work to provide dynamic load balancing in structured P2P systems. In particular, the proposed load balancing algorithm deals with a system where 1) data items are continuously inserted and deleted; 2) nodes join and leave the system continuously; and 3) the distribution of data item IDs and item sizes can be skewed.

Our work presented in this paper bears similarity to previous work [8], [9]. For example, we all use the concept of virtual servers to achieve load balance and the virtual server reassignments performed along the k-ary tree in our approach is similar to *Many-to-Many* scheme [8], [9] in the sense that the *rendezvous points* along the k-ary tree are similar to the *directory* nodes in *Many-to-Many*. However, two important features distinguish our approach from those proposed in [8], [9]. First, our approach uses proximity information to guide virtual server reassignments such that virtual servers are reassigned between physically close nodes, thereby reducing the load movement cost and allowing efficient load balancing. Second, our approach performs virtual server reassignments along the k-ary tree in a bottom-up fashion and it can bound virtual server reassignments in $O(\log N)$ time. In the *Many-to-Many* scheme, randomly rehashing heavy nodes and light nodes into the *directory* nodes may not be able to bound the virtual server reassignment time.

Byers et al. [10] address the load balancing issue in DHTs from a different viewpoint. They propose using the "power of two choices" paradigm to achieve load balance: Each data item (say t) is hashed to a small number d (≥ 2) of different IDs and then is stored in the least loaded node (say X) among the nodes which are responsible for those IDs. To avoid hurting lookup performance significantly, the other nodes maintain a redirection pointer to X for t. This work is complementary to our work. Recent work by Karger and Ruhl [11] proposes two load balancing protocols, namely, *address-space balancing* and *item balancing*, whose provable performance guarantees are within a constant factor of optimal. *Address-space balancing* balances the distribution of the address space (or DHT's identifier space) to nodes. It improves consistent hashing in that each node is responsible for $O(1/N)$ fraction of the whole address space with high probability by keeping one of $O(\log N)$ virtual servers at each node *active* at any time. On the other hand, *item balancing* aims to directly balance the distribution of data items among the nodes when the distribution of items in the address space cannot be randomized (e.g., range searches in database applications). It allows nodes to move to arbitrary positions along the address space. The basic idea behind *item balancing* is that underloaded nodes migrate to portions of the address space occupied by too many data items for load balance.

The k-ary tree structure presented in this paper is similar to the *metadata overlay* proposed in [12]. Both are built on top of a DHT using soft state and used for information

aggregation and dissemination. But, the k-nary tree in our work is built on top of a DHT where each DHT node hosts multiple virtual servers and it also serves as an infrastructure for performing virtual server reassignments.

Proximity information has been used in both topologically-aware DHT construction [13] and proximity neighbor selection in P2P routing tables [14], [15]. The primary purpose of using the proximity information in both cases is to improve the performance of DHT overlays. However, the proximity information used in our work is to make load balancing fast and efficient.

3 System Design

3.1 Background: Virtual Servers

The concept of virtual servers was first proposed in Chord [2] to improve load balance. Like a physical peer node, a virtual server is responsible for a contiguous portion of the DHT's identifier space. A physical peer node can host multiple virtual servers and, therefore, can own multiple noncontiguous portions of the DHT's identifier space. Each virtual server participates in the DHT as a single entity. For example, each virtual server has its own routing table and stores data items whose IDs fall into its responsible region of the DHT's identifier space.

From the perspective of load balancing, a virtual server represents certain amount of load (e.g., the load generated by serving the requests of the data items whose IDs fall into its responsible region). When a node becomes overloaded, it may move part of its loads to some lightly loaded nodes to become light in which the basic unit of load movement is virtual servers [8], [9]. Hence, load balance can be achieved by moving virtual servers from heavy nodes to light nodes. Note that the movement of a virtual server can be simulated as a *leave* operation followed by a *join* operation, both of which are supported by all DHTs. Therefore, using the concept of virtual servers could make our load balancing scheme simple and easily applied to all DHTs.

3.2 System Overview

The load balancing scheme we present in this paper is not restricted to a particular type of resource (e.g., storage, bandwidth, or CPU). However, we make two assumptions in our work. First, we assume that there is only one bottleneck resource in the system, leaving multiresource balancing to our future work. Second, we assume that the load on a virtual server is stable over the timescale it takes for the load balancing algorithm to perform. Basically, our load balancing scheme consists of four phases:

1. *Load balancing information (LBI) aggregation*. Aggregate load and capacity information in the whole system.
2. *Node classification*. Classify nodes into overloaded (heavy) nodes, underloaded (light) nodes, or neutral nodes according to their loads and capacities.
3. *Virtual server assignment (VSA)*. Determine virtual server assignment from heavy nodes to light nodes in order to have heavy nodes become light. The VSA process is a critical phase because it is in this phase that the proximity information is used to make our load balancing scheme proximity-aware.
4. *Virtual server transferring (VST)*. Transfer assigned virtual servers from heavy nodes to light nodes. We allow VSA and VST to partly overlap for fast load balancing.

```
struct KT_node {
  DHT_key key
  DHT_region region
  DHT_node host
  struct KT_node *child[1..k]
  struct KT_node *parent
  ...
  }
```

Fig. 1. KT node data structure.

It is worth pointing out that the load balancing information aggregation (phase 1) may not be needed under some circumstances, though the load balancing scheme we discuss in this paper relies on it to perform node classification (phase 2). As suggested in recent work [9], each node may depend solely on its own load and capacity to determine whether it is overloaded or underloaded, without requiring the system-wide load balancing information. Consider a node i with the load L_i and the capacity C_i. Node i's *utilization* U_i is the fraction of its capacity that is used: $U_i = L_i/C_i$. If $U_i > 1$, node i is identified as a heavy node. Otherwise, it is a light node or neutral node ($U_i = 1$).

As mentioned earlier, our load balancing scheme uses the concept of virtual servers—that is, we assume each node in the system hosts a set of virtual servers. Hence, in the rest of the paper, we restrict our discussion to a DHT where each node has multiple virtual servers.

Roadmap. In Section 3.3, we present a distributed k-ary tree which is used for load balancing information aggregation/dissemination and virtual server reassignments. We then describe the load balancing information aggregation (phase 1) and node classification (phase 2) in Section 3.4 and Section 3.5, respectively. In Section 3.6, we discuss virtual server assignment (phase 3) which is proximity oblivious. The virtual server transferring (phase 4) is described in Section 3.7.

3.3 Building a Distributed k-ary Tree

The aggregation of load balancing information (LBI) naturally leads to the construction of a tree-based structure on top of DHT overlays. In this section, we discuss how to construct a k-ary tree on top of a DHT for load balancing information aggregation/dissemination and virtual server reassignments. For the sake of clarity, in the rest of the paper we refer to the k-ary tree node as the KT node while the node in the DHT overlay is the DHT node or just node.

The distributed k-ary tree is constructed as follows: Each KT node is responsible for a portion of the DHT's identifier space, while the KT root node is responsible for the whole DHT's identifier space. Fig. 1 describes the basic structure of a KT node. The member *key* is a DHT key used for the storage (to plant a KT node X into a virtual server by using the DHT interface $DHT\_put(X.key, X)$) and retrieval (to

```
procedure plant_KT_node(KT_node X)
1: X.key = CenterRegion(X.region)
2: X.host = DHT_put(X.key, X)
```

Fig. 2. Plant a KT node.

locate the KT node X by using the DHT interface $DHT\_get(X.key)$). The member *region* is the portion of the DHT's identifier space the KT node is responsible for. The *key* is produced by taking the *center* point of its responsible *region*.

Each KT node is planted in a virtual server which is responsible for the KT node's *key*. Fig. 2 describes this operation. $CenterRegion(X.region)$ produces a *key* by taking the center point of $X.region$. $DHT\_put(X.key, X)$ then stores X into a virtual server (say V) which is responsible for $X.key$. Note that $DHT\_put(X.key, X)$ also returns the DHT node which owns V. Consider a KT node Y with a responsible region $(3, 5]$ and a virtual server S with a responsible region $(3, 6]$. The KT node Y will be planted in the virtual server S, because the DHT key for Y is 4 by taking the center point of its responsible region $(3, 5]$.

A KT node X's responsible region is further partitioned into k equal parts, each of which is taken by its k children. That is, X's ith child will be responsible for the ith fraction of $X.region$. Each of X's k children (say X_i, $i = 1 \ldots k$) then performs the routine $plant\_KT\_node()$ to reside on its own hosting virtual server. As shown in Fig. 1, X keeps track of its k children in *child*. Note that $X.child[i] \rightarrow host$ allows X to communicate its child X_i directly without resorting to the DHT lookup infrastructure, thereby achieving better performance. In addition, each of X's children X_i keeps track of X in $X_i.parent$. And, also, $X_i.parent \rightarrow host$ allows X_i to communicate its parent X directly.

In the beginning of building the k-ary tree, there is only the KT root node. The partitioning of the DHT's identifier space starts from the KT root node and is continued until a certain termination condition is satisfied (as will be shown below).

To deal with the dynamism of P2P systems such as node joins and departures, each KT node (say X) will periodically run the routine $check\_KT\_node()$, as outlined in Fig. 3. If X's responsible region is completely covered by that of X's hosting virtual server (i.e., the termination condition of the partitioning of X's region is met), then X is already a leaf node and there is no need to grow any more children. In addition, X needs to prune its children (if any) (e.g., due to node departures from the underlying DHT) by running the subroutine $delete\_KT\_children()$. If X's responsible region cannot be fully covered by that of X's hosting virtual server (e.g., due to node additions into the underlying DHT), X needs to grow its children by running the subroutine $add\_KT\_children()$.

Note that each virtual server owns a portion of the DHT's identifier space, it is therefore guaranteed that a KT leaf node will be planted in it. As a result, each DHT node will host multiple KT leaf nodes.

As mentioned earlier, the k-ary tree we are building is used for load balancing information aggregation/dissemination and virtual server reassignments. Such an infrastructure must have the following properties:

- **Self-repair and fault-tolerant**. This property is very important due to the churn in memberships as nodes join or leave the system. It is achieved due to the following facts: First, the k-ary tree is built atop a DHT which already has the self-organizing property. Although the crash of a DHT node (say D) will take away the KT nodes its virtual servers are hosting, the responsible regions of D's virtual servers will be taken over by other DHT nodes after repair. Hence, the periodical checking of all KT nodes (as described in Fig. 3) ensures that the k-ary tree can be completely reconstructed in $O(\log_k N)$ time in a top-down fashion. Note that the KT root node is hosted by a

```
procedure check_KT_node(KT_node X)
1: if (X.region⊆ the responsible region of X.host) then
2:    delete_KT_children(X)
3: else
4:    add_KT_children(X)
5: end if
procedure delete_KT_children(KT_node X)
1: for i = 1 to k do
2:    if (X.child[i]≠ NULL) then
3:       delete X.child[i]
4:       X.child[i] = NULL
5:    end if
6: end for
procedure add_KT_children(KT_node X)
1: for i = 1 to k do
2:    if (X.child[i] == NULL && X.child[i]→region ⊈ the responsible region of X.host) then
3:       c = new KT_node
4:       c→region = the i-th fraction of X.region
5:       X.child[i] = c
6:       c→parent = X
7:       plant_KT_node(c)
8:    end if
9: end for
```

Fig. 3. Check a KT node.

```
procedure KT_node_report_LBI(KT_node X)
1: if (X is a KT leaf node) then
2:    < L_x, C_x, L_{x,min} > ⟵ receive < L_i, C_i, L_{i,min} > from X.host
3: else
4:    Receive < L_i, C_i, L_{i,min} >s from k children /* i = 1, ..., k */
5:    L_x ⟵ Σ_{i=1}^{k} L_i
6:    C_x ⟵ Σ_{i=1}^{k} C_i
7:    L_{x,min} ⟵ the smallest L_{i,min}
8: end if
9: if (X is not a KT root node) then
10:   Report < L_x, C_x, L_{x,min} > to X.parent /* report to the parent node */
11: end if
```

Fig. 4. LBI aggregation algorithm.

virtual server which is responsible for the center point of the whole DHT's identifier space and, thus, it can be located deterministically. Second, all the states the k-ary tree relies on use the principle of *soft-state* and can be refreshed and reconstructed in the event of system changes. Note that each KT node monitors all k children KT nodes for faults using heartbeats sent periodically at certain time interval. Recent work [12] has already shown that such a tree structure built atop of DHTs is able to be self-repair and fault-tolerant upon any failure in the system.

- **Fully distributed**. This property is easily achieved due to the fact that each operation in the k-ary tree involves at most $k+1$ interactions (with a parent node and k children nodes).

3.4 Load Balancing Information (LBI) Aggregation

Based on the k-ary tree structure, LBI aggregation is quite straightforward. Each KT node periodically, at an interval T, requests LBI from its children, while each KT leaf node simply asks its hosting virtual server to report LBI. Recall that it is guaranteed that a KT leaf node will be planted in each virtual server. As such, having each KT leaf node ask its hosting virtual server to report LBI can gather the LBI of the underlying DHT. Note that a DHT node hosts multiple virtual servers. In order to avoid reporting redundant LBI of a DHT node, a DHT node (say i) randomly chooses one of its virtual servers to report LBI, in the form of $< L_i, C_i, L_{i,min} >$ (where L_i, C_i, and $L_{i,min}$ stand for the total load of virtual servers, the capacity, and the minimum load of virtual servers on the node i, respectively).

Each KT node (say X) runs the routine $KT\_node\_report\_LBI()$ as outlined in Fig. 4. This process is continued along the k-ary tree in a bottom-up fashion until the KT root node is reached. As a result, the KT root node produces a system-wide LBI $< L, C, L_{min} >$, where L, C, and L_{min} represent the total load, the total capacity, and the smallest load of virtual servers in the system, respectively.

Note that the LBI aggregation completion is in $O(\log_k N)$ time. In the event of the crashing of DHT nodes during the process of LBI aggregation, as discussed earlier, the k-ary tree can recover in $O(\log_k N)$ time. Hence, the LBI process can continue along the k-ary tree in a bottom-up sweep after the tree is reconstructed.

3.5 Node Classification

After the LBI aggregation, the KT root node disseminates $< L, C, L_{min} >$ along the k-ary tree in a top-down fashion to each KT leaf node, which in turn distributes the $< L, C, L_{min} >$ to its own hosting virtual server. As a result, all DHT nodes are guaranteed to have a copy of the $< L, C, L_{min} >$. Let L_i denote the sum of the loads of all virtual servers on a DHT node i, and C_i represent the capacity of a DHT node i. Note that one of the goals of our load balancing scheme is to ensure fair load distribution over DHT nodes by assigning the load to a DHT node in proportion to its capacity. Let T_i denote the target load of a DHT node i proportional to its capacity. We have $T_i = (\frac{L}{C} + \varepsilon)C_i$ (ε is a parameter for a trade off between the amount of load moved and the quality of balance achieved. Ideally, ε is 0).

Therefore, a DHT node i can be defined as:

- A *heavy node* if $L_i > T_i$.
- A *light node* if $(T_i - L_i) \geq L_{min}$.
- A *neutral node* if $0 \leq (T_i - L_i) < L_{min}$.

3.6 Virtual Server Assignment (VSA)

Similar to the LBI aggregation, the VSA process is performed along the k-ary tree in a bottom-up fashion. Initially, each heavy DHT node (say i) chooses a subset of its virtual servers $\{v_{i,1}, \ldots, v_{i,m}\}$ ($m \geq 1$) that minimizes $\sum_{k=1}^{m} L_{i,k}$ subject to the condition that $(L_i - \sum_{k=1}^{m} L_{i,k}) \leq T_i$ (where $L_{i,k}$ denotes the load of the virtual server $v_{i,k}$ on the DHT node i). This subset of virtual servers are expected to be moved to make the heavy node i to become light. Note that this choice of virtual servers on heavy nodes would minimize the total amount of load moved for load balancing throughout the system. Then, the heavy DHT node i *randomly* chooses one of its virtual servers to report the *VSA information*

$$< L_{i,1}, v_{i,1}, ip\_addr(i) >, \ldots, < L_{i,m}, v_{i,m}, ip\_addr(i) >$$

to its hosted KT leaf node, which in turn propagates the VSA information upward along the tree. For a light DHT node (say j), it *randomly* chooses one of its virtual servers to report the *VSA information* $< \Delta L_j = T_j - L_j, ip\_addr(j) >$ to its hosted KT leaf node, which in turn propagates the VSA information upwards along the tree.

During the propagation of VSA information along the k-ary tree, each KT node (say X) runs the routine $KT\_node\_VSA()$ as described in Fig. 5. It collects the VSA information into VSA_pool from either its hosting virtual server or its children (lines 2-6). If the number of VSA information reaches a predefined *paring threshold*, the node X would serve as a *rendezvous point* for virtual server assignments, by running the subroutine $KT\_node\_rendezvous\_point()$. This subroutine uses a *best-fit heuristic* approach[2] to perform virtual server assignments. The successfully assigned VSA information is sent back to corresponding DHT nodes for virtual server transferring, while the unpaired VSA information is propagated to X's parent. This VSA process is continued in a bottom-up fashion along the k-ary tree until it reaches the KT root node. Then, the root node serves as the last rendezvous point (without a pairing threshold constraint) for virtual server assignments.

2. Computing an optimal reassignment of virtual servers from heavy nodes to light nodes is **NP**-complete [8], [9]. We use this simple greedy algorithm to find an approximation solution to minimize the load to be reassigned and moved.

procedure KT_node_VSA(KT_node X)
1: $VSA\_pool \longleftarrow \emptyset$
2: **if** (X is a KT leaf node) **then**
3: $VSA\_pool \longleftarrow$ VSA information from $X.host$ /* receive the VSA information from its hosting virtual server */
4: **else**
5: $VSA\_pool \longleftarrow$ VSA information from k children
6: **end if**
7: **if** ($VSA\_pool.size \geq pairing\_threshold \parallel X$ is a KT root node) **then**
8: $KT\_node\_rendezvous\_point(X, VSA\_pool)$ /* X serves as a rendezvous point*/
9: **else**
10: Report VSA_pool to $X.parent$ /* propagate the VSA information to its parent */
11: **end if**

procedure KT_node_rendezvous_point(KT_node X, VSA information $pool$)
1: $light\_list \longleftarrow$ remove all $< \Delta L_j = T_j - L_j, ip_addr(j) >$s from $pool$ /* $light\_list$ maintains the VSA information of light nodes */
2: $heavy\_list \longleftarrow$ remove all $< L_{i,r}, v_{i,r}, ip_addr(i) >$s from $pool$ /* $heavy\_list$ maintains the VSA information of heavy nodes */
3: $pool \longleftarrow \emptyset$
4: **while** ($heavy\_list \neq \emptyset$) **do**
5: Remove the most loaded virtual server $v_{i,r}$ from $heavy\_list$, and assign it to a DHT node j in $light\_list$ such that ΔL_j is minimized and subject to the condition that $\Delta L_j \geq L_{i,r}$
6: **if** ($v_{i,r}$ can be assigned) **then**
7: Remove $< \Delta L_j, ip\_addr(j) >$ from $light\_list$
8: **if** ($\Delta L_j - L_{i,r} \geq L_{min}$) **then**
9: Insert $< \Delta L_j - L_{i,r}, ip\_addr(j) >$ into $light\_list$
10: **end if**
11: Send the assigned information $< v_{i,r}, ip\_addr(i), ip\_addr(j) >$ to DHT nodes i and j /* prepare for virtual server transferring */
12: **else**
13: $pool \longleftarrow pool \cup \{< L_{i,r}, v_{i,r}, ip\_addr(i) >\}$
14: **end if**
15: **end while**
16: $pool \longleftarrow pool \cup light\_list$
17: **if** ($pool.size > 0$ && X is not a KT root node) **then**
18: Report $pool$ to $X.parent$ /* report un-assigned VSA information to its parent */
19: **end if**

Fig. 5. Virtual server assignment algorithm.

In summary, as the VSA process proceeds along the k-ary tree in a bottom-up sweep, it *recursively assigns virtual servers among DHT nodes scattered in an increasingly larger contiguous portion of the DHT's identifier space*[3] *until the whole DHT's identifier space (for which the* k-ary *tree root node is responsible)*. In other words, the VSA process is *identifier space-based* in that the virtual server assignments are performed earlier among those DHT nodes which are closer to each other in the DHT's identifier space. Similar to the LBI aggregation process, the VSA process is also resilient to system failures due to the robustness of the k-ary tree it depends on. After the k-ary tree recovers from DHT node's failures, the VSA process can continue along the tree in a bottom-up fashion.

It is worth pointing out that the VSA process discussed above is *proximity-ignorant* because the logical closeness in the DHT's identifier space does not necessarily reflect physical closeness of DHT nodes. We name it *proximity-ignorant VSA*. In essence, the proximity-ignorant VSA is similar to the *Many-to-Many* scheme proposed in [8] in the sense that the rendezvous points along the k-ary tree can be viewed as the *directory nodes* in the *Many-to-Many* scheme. We believe the performance of the proximity-ignorant VSA is similar to that of the *Many-to-Many* scheme except that the proximity-ignorant VSA completes quickly in $O(\log N)$ time. Thus, the results of the proximity-ignorant load balancing scheme reported in Section 5.2.3 can represent those of the *Many-to-Many* scheme.

3. Note that here the location of a DHT node in the identifier space is represented by its randomly chosen virtual server.

As mentioned earlier, we assume that the load on a virtual server is stable over the timescale it takes for the load balancing algorithm to perform. However, if this assumption does not hold in some case, we need to address the issue of *stale VSA information*. If a node relies on the system-wide information (e.g., obtained by the load balancing information aggregation/dissemination process along the k-ary tree) to determine its status (i.e., heavy or light), the information may be up to $3 \cdot T$ ($T = O(\log N)$) time old. On the other hand, if a node could determine its status solely dependent on its own information (as suggested in [9]), the information may be up to T time old. But, most information is expected to be less than T time old since the internal k-ary tree nodes can serve as the rendezvous points for virtual server assignments. Moreover, inspired by [9], a rendezvous point could schedule *emergency balancing* for some *urgent* heavy nodes.

3.7 Virtual Server Transferring (VST)

The VST process is quite simple and straightforward. Upon receiving the paired VSA information (say $< v_{i,r}, ip\_addr(i), ip\_addr(j) >$), the heavy DHT node i will transfer the virtual server $v_{i,r}$ to the light DHT node j. Note that the VST process can be performed partly overlapping with the VSA process for fast load balancing.

The transferring of a virtual server unavoidably causes the k-ary tree to restructure because the KT node which is planted in a virtual server has to migrate with the virtual server. In order to keep the k-ary tree relatively stable, we could adopt a lazy migration protocol for the KT node.

Only after the transferring of a virtual server is fully completed or after the whole VSA process is fully completed will the KT node migrate. Note that the restructuring of the k-ary is fully distributed and inexpensive because each KT node migration only involves at most $k+1$ messages.

4 Proximity-Aware Load Balancing

The load balancing scheme we have discussed so far is proximity-ignorant. In this section, we present the proximity-aware load balancing scheme. The basic idea behind the proximity-aware load balancing is to make virtual server assignments (i.e., the VSA process) proximity-aware by using proximity information. We first describe how to generate proximity information using landmark clustering techniques in Section 4.1, and then discuss the proximity-aware VSA process which uses the generated proximity information in Section 4.2.

4.1 Generating Proximity Information

Landmark clustering has been widely used to generate proximity information (e.g., [13], [14], [15]). It is based on an intuition that nodes physically close to each other are likely to have similar distances to a few selected nodes.

In a DHT overlay network, the landmark nodes can be chosen from either the overlay itself or the Internet. For a DHT node D, it measures the distances to a set of m landmark nodes (e.g., $m = 15$) and obtains a *landmark vector* $< d_1, d_2, \ldots, d_m >$. Node D is then mapped into a point in a m-dimensional Cartesian space by having the landmark vector as its coordinates. We call this Cartesian space the *landmark space*. As such, two physically close DHT nodes (say, D and E) would have similar/close landmark vectors and be close to each other in the landmark space. Note that a sufficient number of landmark nodes need to be used to reduce the probability of false clustering where nodes that are physically far away have similar/close landmark vectors. In our study, 15 landmark nodes are used in the landmark clustering.

Landmark clustering is a coarse-grained approximation and is not very effective in differentiating nodes within close distance [14], [15]. However, our experimental results (as will be shown in Section 5.2.3) show it works well for our load balancing scheme. This is largely because our load balancing scheme does not need very precise measurements. The pairing threshold constraint at rendezvous points during the VSA process naturally provides a trade off between load balance (a bigger pairing threshold means more DHT nodes' VSA information and is expected to provide a better load balance as will be shown in Section 5.2.4) and landmark clustering. Moreover, our experimental results as well as recent work [14], [15] have shown that 15 landmark nodes are pretty good in landmark clustering.

4.2 Proximity-Aware VSA Using Proximity Information

After generating proximity information, a big challenge we now face is how to effectively use it to guide virtual server assignments such that they are assigned between *physically* close heavy nodes and light nodes. As discussed in Section 3.6, the bottom-up VSA process along the k-ary tree is *identifier space-based* in that virtual servers are assigned *earlier* among DHT nodes if they are *logically closer* to each other in the DHT's identifier space. Therefore, the basic idea behind the proximity-aware VSA is to *use proximity information to map the VSA information of heavy nodes and light nodes into the underlying DHT such that physically close nodes' VSA information is also close to each other in the identifier space—that is, preserve the proximity relationships in the identifier space*. As such, the bottom-up VSA process along the k-ary tree naturally guarantees that virtual servers are assigned *earlier* among *physically closer* heavy nodes and light nodes.

It is worth pointing out that, unlike [13], we do not alter the DHT overlay structure such that physically close nodes are also close to each other in the identifier space and, instead, we just simply map the *VSA information* of heavy nodes and light nodes into the underlying DHT. For example, consider two physically close nodes i and j (i is a heavy node which needs to shed a virtual server $v_{i,k}$ to become light and j is a light node). We just map their VSA information $< L_{i,k}, v_{i,k}, ip\_addr(i) >$ and $< T_j - L_j,\ ip\_addr(j) >$ into the underlying DHT according to the information derived from two nodes' landmark vectors (as will be shown later) such that their VSA information $< L_{i,k}, v_{i,k}, ip\_addr(i) >$ and $< T_j - L_j, ip\_addr(j) >$ is also stored close to each other in the DHT's identifier space.

Recall that, in a DHT, an *object* is mapped into the identifier space with a DHT *key* by the interface of $DHT\_put(key, object)$. If two objects have similar/close DHT keys (close to each other in the identifier space), then these two objects will be stored close to each other in the DHT overlay (or the identifier space). Hence, the key issue (in mapping the VSA information of physically close nodes into the underlying DHT so that their VSA information is also stored close to each other in the identifier space) is how to derive similar/close DHT keys for physically close nodes from their similar/close landmark vectors. In other words, how do we preserve the closeness when deriving DHT keys from similar landmark vectors?

4.2.1 Deriving DHT Keys from Landmark Vectors

In this section, we address the issue of how to preserve the closeness when deriving DHT keys from similar landmark vectors. The first solution is that we can simply use the landmark vector of a node as a DHT key. However, due to the fact that the landmark space is usually of relatively high dimension compared to the DHT's identifier space, we cannot adopt this simple solution. To solve this problem, we could follow the approach suggested in [14], [15] by using *space-filling curves* [16].

Space filling curves such as the Hilbert curve [16] are a class of continuous, proximity preserving mappings from a m-dimensional space to a one-dimensional space, i.e., $N^m \longmapsto N^1$, such that each point in N^m is mapped to a unique point or index in N^1. The mapping can thus be thought of as laying out a string within the m-dimensional space so that it completely fills the space. The one-dimensional mapping generated by the space-filling curve serves as an ordered indexing into the m-dimensional space. One property of space filling curves is that points that are close together in the m-dimensional space will be mapped to

points that are close together in the one-dimensional space, i.e., proximity is preserved by the mapping.

Therefore, we divide the m-dimensional landmark space into 2^{mn} grids of equal size (where n controls the number of grids used to divide the landmark space) and fill a Hilbert curve within the landmark space to number each grid. We then number each DHT heavy/light node with the grid number of the grid in which its landmark vector falls. We call this grid number the *Hilbert number*, which will serve as a DHT key. Due to the proximity preserving property of the Hilbert curve, closeness in the Hilbert number reflects physical proximity. Moreover, a smaller n increases the likelihood that two physically close nodes have the same Hilbert number. It should be pointed out that using space-filling curves to reduce a high-dimension landmark vector could introduce inaccuracy. However, we believe our load balancing scheme can tolerate this inaccuracy. This is because our load balancing scheme does not need very precise information due to the paring threshold constraint at the rendezvous points during the VSA process.

4.2.2 Proximity-Aware VSA

Without loss of generality, we use Chord as the example, but the techniques discussed here are applicable or easily adapted to other DHTs such as Pastry and Tapestry.

Initially, each DHT heavy/light node independently determines its landmark vector $< d_1, d_2, \ldots, d_m >$ to a set of m landmark nodes, and derives a DHT key from the landmark vector as described above. Then, each DHT heavy/light node (say i) publishes its VSA information (e.g., $< L_{i,k}, v_{i,k}, ip\_addr(i) > / < T_i - L_i, ip\_addr(i) >$) into the DHT overlay with the DHT key. As such, the VSA information published by physically close nodes will be close together in the DHT's identifier space.

Given the published VSA information by all participating DHT nodes, the proximity-aware VSA differs from the proximity-ignorant VSA in that each individual virtual server *independently* reports the VSA information (if any) *which has been mapped into its responsible region of the identifier space*[4] to the KT leaf node which it hosts. In the case of multiple KT leaf nodes planted in a virtual server, the virtual server reports the VSA information to only one of its KT leaf nodes to avoid sending redundant information.

As a result, each KT leaf node has the VSA information of DHT heavy nodes and light nodes which are physically close together unless no VSA information is reported by its hosting virtual server. If the KT leaf node has the VSA information, it performs the following operations:

1. If the number of the VSA information reaches the predefined pairing threshold, it can immediately serve as a rendezvous point for the virtual server assignments by running the routine $KT\_node\_rendezvous\_point()$, as outlined in Fig. 5.
2. Otherwise, it propagates the VSA information to its KT parent node.

Then, the VSA process proceeds along the k-ary tree in a bottom-up sweep, as described in Section 3.6. Note that a k-ary subtree covers a contiguous portion of the DHT's identifier space and the subtree's root node may serve as a rendezvous point for virtual server assignments, so this bottom-up VSA process naturally guarantees that the virtual servers are assigned among DHT nodes, recursively in a decreasing physical closeness order as the rendezvous point moves up along the k-ary tree.

4. In the proximity-ignorant VSA, one of a DHT node's virtual servers instead is randomly chosen to report the DHT node's *own* VSA information.

5 Experimental Evaluation

5.1 Experiment Setup

We built two k-ary trees (with $k = 2$ and 8, respectively) on top of a Chord simulator (32-bit identifier space) for both load balancing information aggregation/dissemination and virtual server assignments.

We assumed the load of a virtual server is associated with the fraction of the DHT's identifier space it owns. And, we also assumed the fraction of the identifier space (let f denote the fraction) owned by a virtual server is exponentially distributed because it is true in Chord [2] and CAN [5]. To simulate the load of a virtual server, we used two distributions proposed by Rao et al. [8] (let μ and σ represent the mean and standard deviation of the total load in a DHT, respectively):

- *Gaussian distribution*. The load of a virtual server is generated using a Gaussian distribution with mean μf and standard deviation $\sigma\sqrt{f}$. This distribution would result if the load of a virtual server comes from the *independent* individual loads on a large number of small objects the virtual server stores [8].
- *Pareto distribution*. The load of a virtual server is generated using a Pareto distribution with the shape parameter $\alpha = 1.5$ and mean μf. The standard deviation is infinite. Due to the heavy-tailed nature of this distribution, it presents a particularly bad case for load balancing [8].

To account for heterogeneity in node capacity, we used two capacity profiles (which were used in [17]):

- *Gnutella-like*. We assigned capacity of 1, 10, 10^2, 10^3, and 10^4 to Chord nodes with probability of 20 percent, 45 percent, 30 percent, 4.9 percent, and 0.1 percent, respectively.
- *Zipf-like*. When sorted, the ith Chord node has capacity of $1,000 \cdot i^{-\beta}$ (where β is 1.2).

We evaluated the proximity-aware load balancing using two transit-stub topologies with approximately $5,000$ nodes each, produced by GT-ITM [18]. These two topologies have 10 graphs each and we ran all these graphs in our simulations:

- "ts5k-large" has five transit domains, three transit nodes per transit domain, five stub domains attached to each transit node, and 60 nodes in each stub domain on average. The average degree is 17.8.
- "ts5k-small" has 120 transit domains, five transit nodes per transit domain, four stub domains attached to each transit node, and two nodes in each stub domain on average. The average degree is 7.4.

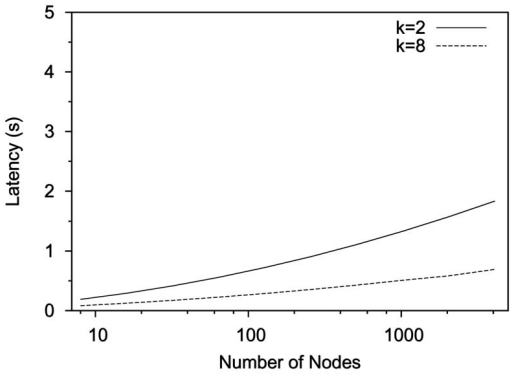

Fig. 6. Each DHT node hosts five virtual servers. k represents the degree of the tree. Each link in the physical network on top of which the DHT is layered has a uniform latency of 20 ms.

"ts5k-large" represents a situation in which the Chord overlay consists of nodes from several big stub domains, while "ts5k-small" represents a situation in which the Chord overlay consists of nodes scattered in the entire Internet. To account for the fact that interdomain routes have higher latency each interdomain hop counts as 3 hops of units of latency while each intradomain hop counts as 1 hop of unit of latency. We chose 15 nodes as landmark nodes to generate the landmark vectors from which we derived the DHT keys for each Chord node. Nodes in a stub domain have close DHT keys. We expect that the proximity-aware load balancing could perform very well in "ts5k-large."

It is worth pointing out that the focus of our experimental evaluation is to investigate the impact of our proximity-aware load balancing algorithm while considering the heterogeneity nature of P2P systems. We do not claim that our algorithm is bullet proof. Other aspects such as the robustness of the algorithm need further exploration in our future work.

5.2 Experimental Results

5.2.1 The Distributed k-ary Tree

The first set of experiments measured the overhead of k-ary tree construction. We used two metrics to measure the overhead:

- **Node stress**. It is defined as the number of messages a node receives and is used to quantify the load on nodes.
- **Link stress**. It is defined as the number of messages sent over a physical network link and is used to quantify the load on the network.

The experimental results showed that the mean node stresses for "ts5k-large" and "ts5k-small" are 11.6 and 13.9, respectively. The mean link stresses for "ts5k-large" and "ts5k-small" are 15.9 and 18.7, respectively. The results demonstrate that both node stress and link stress during the k-ary tree construction are low. In addition, we believe some future optimization methods could be applied to further reduce the overhead, e.g., by piggybacking the messages into DHT overlay maintenance messages.

Fig. 6 shows the latency as a function of network size we observed in the experiments for LBI aggregation as well as virtual server assignments (we assumed during the VSA process the cost of pairing process at each rendezvous point along the k-ary tree is negligible). Note that the latency is very small. For example, for a network of $4,096$ nodes, the latencies for $k = 2$ and $k = 8$ are about 1.8s and 0.7s, respectively. LBI dissemination is essentially the reverse: information is distributed along the tree down to the leaves and, therefore, the latency is similar to that of the LBI aggregation.

During the LBI aggregation/dissemination and VSA process, each KT node involves at most $k + 1$ interactions (one with the parent, and k with children). Thus, the overhead in a k-ary tree operation is a constant. The entities involved are actually the DHT nodes which host the k-ary tree. It seems that toward the KT root node the hosting DHT nodes need to have increasingly higher bandwidth and stability. However, stability is not a concern because, as discussed earlier, the k-ary tree hierarchy can be recovered in $O(\log N)$ time. As for bandwidth, it is not a problem because the message size is constant during the LBI aggregation and, during the VSA process the pairing operation at each rendezvous point reduces the VSA information (only the unpaired VSA information needs to be sent upward) and, thereby, brings down the bandwidth requirements (the pairing threshold gives a trade off). Moreover, the VSA information (i.e., $< L_{i,k}, v_{i,k}, ip\_addr(i) > / < T_i - L_i, ip\_addr(i) >$) is only several bytes.

5.2.2 Aligning Two Skews: Load and Capacity

In all experimental results we present in the rest of this paper, the Chord overlay consists of 4,096 nodes each with five virtual servers in the beginning and the degree of the k-ary tree is 2 (we observed similar results on the degree of 8). The pairing threshold at rendezvous points during the VSA process is 50 by default.

In this section, we present the results of load balancing in aligning those two skews in load distribution and node capacity. Due to space constraints, some results are omitted here. Fig. 7 shows the scatterplot of loads for the Gaussian distribution with Gnutella-like capacity profile and Fig. 8 shows the scatterplot of loads for the Pareto distribution with Zipf-like capacity profile. Note that our load balancing approach is able to align the two skews in load distribution and node capacity inherent in P2P systems—that is, have nodes carry loads proportional to their capacities by reassigning virtual servers among nodes.

5.2.3 Proximity-Aware Load Balancing

So far, we have evaluated one goal of our load balancing approach—that is, make heavy nodes to become light by transferring excess virtual servers to light nodes meanwhile having higher capacity nodes carry more loads. A question remaining is what is the benefit of the proximity-aware load balancing approach? Intuitively, that more loads are moved within shorter distances indicates less load movement cost (e.g., in terms of bandwidth usage) and faster convergence on load balance.

Fig. 9 shows cumulative distribution of moved load for "ts5k-large." The x-axis denotes the distance of virtual server transferring in terms of hops, while the y-axis represents the

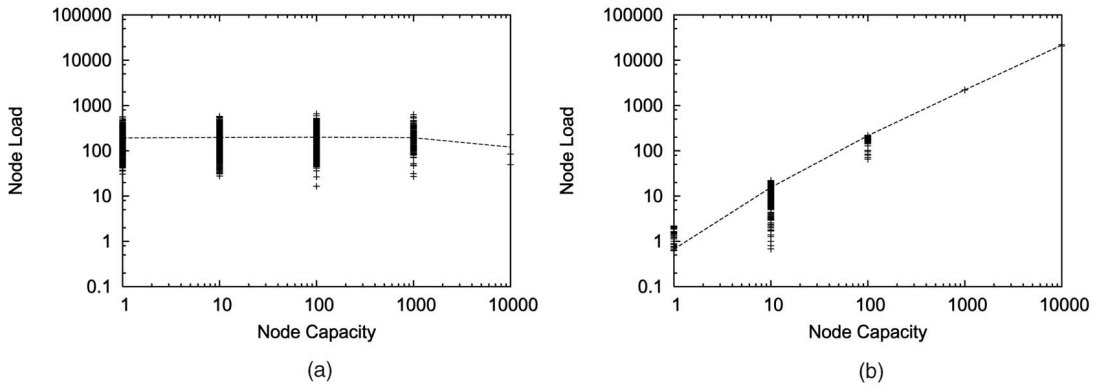

Fig. 7. Results for the Gaussian distribution, according to the Gnutella-like capacity profile. (a) Before load balancing and (b) after load balancing.

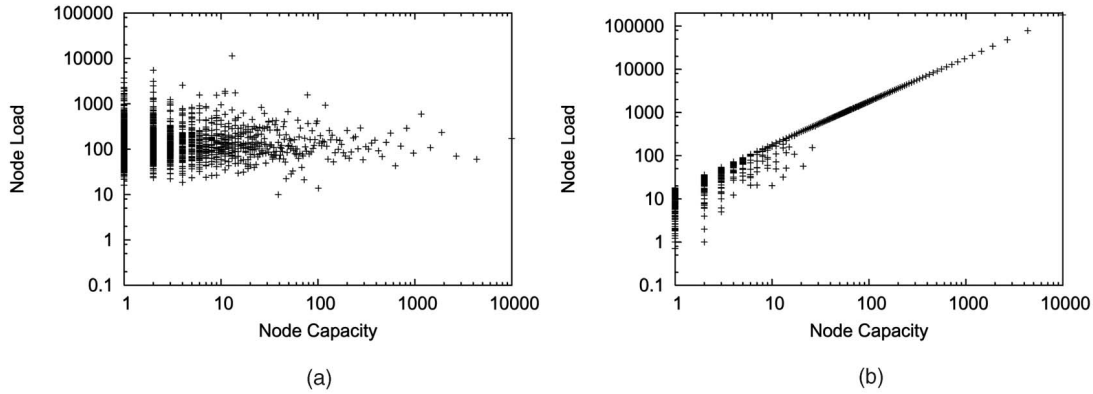

Fig. 8. Results for the Pareto distribution, according to the Zipf-like capacity profile. (a) Before load balancing and (b) after load balancing.

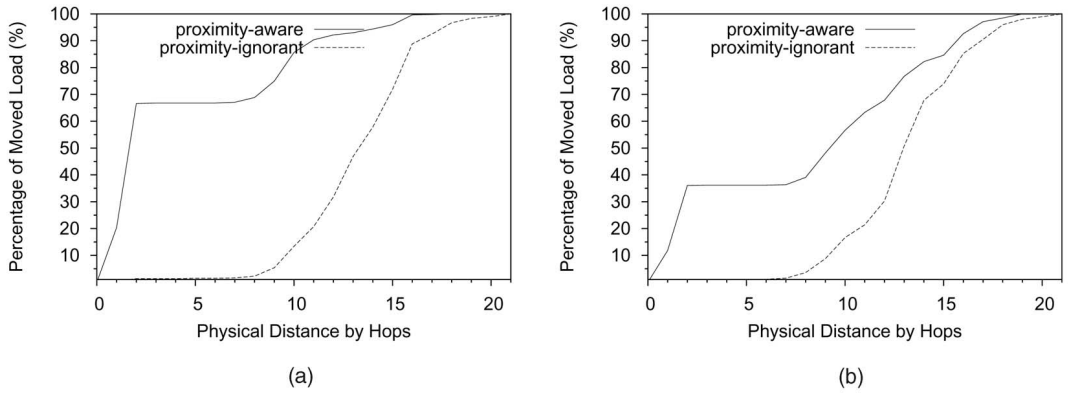

Fig. 9. Cumulative distribution of moved load for "ts5k-large." (a) Gaussian distribution and Gnutella-like capacity profile. (b) Pareto distribution and Zipf-like capacity profile.

percentage of total moved load. From Fig. 9a, we can see that the proximity-aware load balancing scheme transfers about 67 percent of total moved load within 2 hops and about 86 percent within 10 hops, while the proximity-ignorant load balancing scheme transfers only about 13 percent of total moved load within 10 hops. Fig. 9b shows that the proximity-aware load balancing scheme transfers about 36 percent of total moved load within 2 hops and about 57 percent within 10 hops, while the proximity-ignorant load balancing scheme transfers only about 17 percent of total moved load within 10 hops. Such a big difference implies that the proximity-aware load balancing scheme can effectively assign and transfer loads between physically close nodes—that is, a very large faction of loads are reassigned and transferred within the same stub domains, thereby reducing the load balancing cost (e.g., bandwidth consumption) and enabling fast and efficient load balancing.

Fig. 10 shows cumulative distribution of moved load for "ts5k-small." Note that, in this case, Chord nodes are randomly chosen from the nodes scattered in the entire Internet. The proximity-aware load balancing scheme still performs much better than the proximity-ignorant load balancing scheme. This is because the proximity-aware load balancing scheme can effectively guide heavy nodes to transfer loads to physically nearby light nodes by using the proximity information, in spite of the fact that most of the

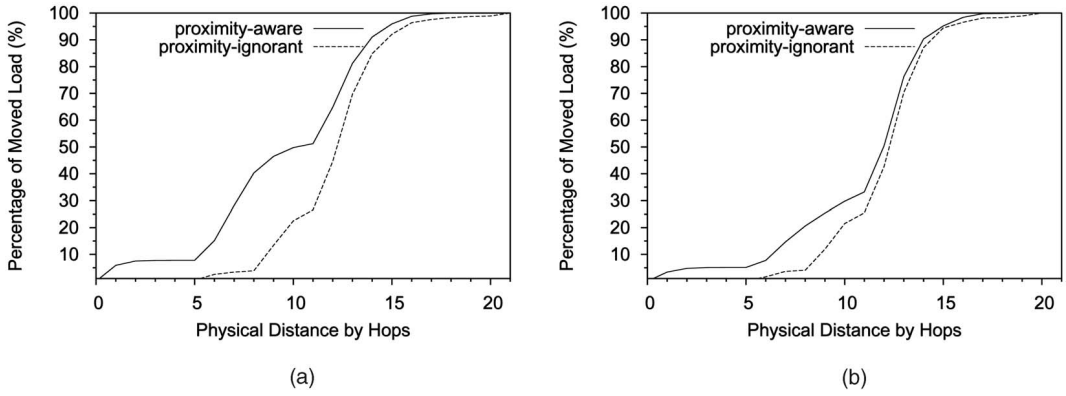

Fig. 10. Cumulative distribution of moved load for "ts5k-small." (a) Gaussian distribution and Gnutella-like capacity profile. (b) Pareto distribution and Zipf-like capacity profile.

nodes are scattered in the entire Internet. In other words, the proximity-aware approach is "greedy" in the sense that it tries, at each step, to reduce the load movement cost by making virtual server assignments among physically close nodes.

We further quantify the benefit of the proximity-aware load balancing scheme over the proximity-ignorant scheme in terms of bandwidth. The load has a *movement cost*, which we are charged each time we transfer the load between nodes. Let $LM(d)$ denote the loads moved at the distance of d hops. Then, the load movement cost incurred by a load balancing algorithm can be $C = \sum_{d=1}^{n} LM(d) \cdot d$.

Let C_{aware} and $C_{ignorant}$ be the load movement cost of the proximity-aware load balancing and the proximity-ignorant load balancing, respectively. Then, the benefit B achieved by the proximity-aware load balancing over the proximity-ignorant load balancing can be $B = \frac{C_{ignorant} - C_{aware}}{C_{ignorant}}$.

Our experimental results show the benefit B for "ts5k-large" is 37-65 percent (for all combinations of load distributions and node capacity profiles) and that for "ts5k-small" is 11-20 percent (for all combinations of load distributions and node capacity profiles). Note that the bandwidth savings by the proximity-aware load balancing are very significant in "ts5k-large." This is because, in "ts5k-large," heavy nodes probably can find light nodes within the same stub domains to move loads, thereby reducing the bandwidth consumption across different stub domains. Even in "ts5k-small," the bandwidth savings are nontrivial. Aside from bandwidth savings, another implication (that more loads are moved within shorter distances) is faster completion of load balancing throughout the system since less loads need to be transferred across high-latency wide-area links.

5.2.4 Impact of the Pairing Threshold

Recall that, during the VSA process, a KT node (except the KT root node) serves as a rendezvous point only if the number of the VSA information reaches a given pairing threshold θ. A rendezvous point then uses the best-fit heuristic approach to assign virtual servers. To explore the impact of the pairing threshold θ on load balancing, we define the quality of load balancing $Q = \frac{L_{actual}}{L_{ideal}} \times 100\%$, where L_{actual} denotes the loads actually reassigned and moved by a load balancing approach and L_{ideal} represents the loads needing to be reassigned and moved to achieve the perfect load balance (i.e., no node is overloaded).

For "ts5k-large," when θ was chosen to be 20, 30, and 50 (for all combinations of load distributions and node capacity profiles), the Q for the proximity-ignorant load balancing was 98.6+ percent, 99.2+ percent, and 100 percent, respectively. However, the Q for the proximity-aware load balancing was 100 percent for all these thresholds, immune to the changing of the threshold value. We also noted that the benefit B brought by the proximity-aware load balancing was unaffected by all these thresholds. This is mainly due to the synergy between the proximity-aware load balancing and "ts5k-large." For "ts5k-large," the VSA information of nodes in a stub domain is probably mapped into the same virtual server and, therefore, clustered together in the same KT leaf node. Due to the number of nodes in each stub domain (60 on average), the KT leaf node probably has sufficient VSA information[5] to serve as a rendezvous point anyway, given all these thresholds (note that a very large portion of loads were assigned here, about 36-67 percent within 2 hops as shown in Section 5.2.3). However, as θ becomes big, the benefit B of the proximity-aware load balancing decreases. For example, when θ was 200, the benefit B was reduced to 16-35 percent for all combinations of load distributions and node capacity profiles. When θ is infinite (only the KT root node serves as a rendezvous point), there is no benefit at all. In this case, the proximity-aware load balancing is completely degraded to the proximity-ignorant load balancing.

For "ts5k-small," like the proximity-ignorant load balancing, the proximity-aware load balancing was affected by small thresholds in terms of Q. This is attributed to the fact that, in "ts5k-small," nodes are scattered in the entire Internet. That a KT node is able to become a rendezvous point is therefore closely associated with the value of the pairing threshold. A relatively bigger threshold makes the best-fit strategy perform better (in terms of Q) from the

5. We found the number of VSA information in such KT leaf nodes was 50 or more upon various combinations of load distributions and node capacity profiles.

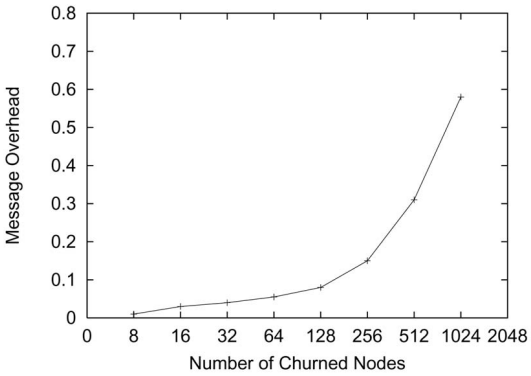

Fig. 11. Message overhead incurred by node churn.

system-wide perspective. However, the benefit B decreases as θ becomes big.

In summary, the proximity-aware load balancing is somehow insensitive to small thresholds in "ts5k-large"-like topologies, but the proximity-ignorant load balancing is relatively sensitive to small thresholds in both "ts5k-large"-like and "ts5k-small"-like topologies. Further, a big θ would reduce the benefit the proximity-aware load balancing strives for. Therefore, the pairing threshold provides a trade off for the proximity-aware load balancing. A too small pairing threshold may compromise the quality of load balancing, while a too big pairing threshold may hurt the benefit. An appropriate pairing threshold not only guarantees the quality of load balancing, but also maximizes the benefit. We leave the issue of how to choose an appropriate paring threshold upon various network topologies to our future work.

5.2.5 Impact of Node Churn

Previous experiments evaluated the load balancing scheme without considering node churn, i.e., node joins and leaves the system. Now, we quantify the message overhead incurred by node churn. The overhead metric is defined as $overhead = \frac{M_d - M_s}{M_s}$, where M_s is the number of messages during the VSA process without node churn and M_d is the number of messages during the VSA process when nodes continuously join and leave the system. It should be worth pointing out that the underlying DHT overlay maintenance messages incurred by node churn are not included in M_d because these messages are the overhead inherent in a DHT. Throughout the experiment, we kept the number of nodes in the system to be 4,096.

Fig. 11 plots this metric as a function of the number of nodes which join and leave the system during the VSA process. When the number of churned nodes is low, the overhead is low. When the number of churned nodes reaches 1,024 (25 percent of nodes), the overhead is about 58 percent. This overhead is mainly caused by the restructuring of the k-ary tree due to node joins and departures. Also, we believe some future optimization methods could be applied to further reduce the overhead, e.g., by piggybacking the restructuring messages into the DHT overlay maintenance messages.

6 Conclusions

In this paper, we present an efficient, proximity-aware load balancing scheme to tackle the issue of load balancing in DHT-based P2P systems. The first goal of our load balancing scheme is to align those two skews in load distribution and node capacity inherent in P2P systems to ensure fair load distribution among nodes—that is, have nodes carry loads proportional to their capacities. The second goal is to use the proximity information to guide load reassignment and transferring, thereby minimizing the cost of load balancing and making load balancing fast and efficient. We conducted a detailed simulation study using a Chord simulator, two representative load distributions, two node capacity profiles, and two representative Internet topologies. The results show that our proximity-aware load balancing scheme can not only ensure fair load distribution, but also minimize the load movement cost. We show the proximity-aware load balancing is very efficient, e.g., by 11-65 percent bandwidth savings.

Acknowledgments

This work is supported in part by the US National Science Foundation under Career Award CCR-9984852 and ACI-0232647, and the Ohio Board of Regents. The authors thank the anonymous reviewers for their valuable feedback. They are also grateful to Hung-Chang Hsiao for helpful comments on this paper. An early version of this work [1] was presented in the *Proceedings of IPDPS '04*.

References

[1] Y. Zhu and Y. Hu, "Towards Efficient Load Balancing in Structured P2P Systems," *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Apr. 2004.

[2] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. ACM SIGCOMM*, pp. 149-160, Aug. 2001.

[3] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *Proc. 18th IFIP/ACM Int'l Conf. Distributed System Platforms (Middleware)*, pp. 329-350, Nov. 2001.

[4] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph, "Tapestry: An Infrastructure for Fault-Tolerance Wide-Area Location and Routing," Technical Report UCB/CSD-01-1141, Computer Science Division, Univ. of California, Berkeley, Apr. 2001.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *Proc. ACM SIGCOMM*, pp. 161-172, Aug. 2001.

[6] S. Saroiu, P.K. Gummadi, and S.D. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems," *Proc. Multimedia Computing and Networking (MMCN)*, Jan. 2002.

[7] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP)*, pp. 202-215, Oct. 2001.

[8] 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, Feb. 2003.

[9] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Dynamic Structured P2P Systems," *Proc. IEEE INFOCOM*, Mar. 2004.

[10] J.W. Byers, J. Considine, and M. Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables," *Proc. Second Int'l Workshop Peer-to-Peer Systems (IPTPS)*, pp. 80-87, Feb. 2003.

[11] D.R. Karger and M. Ruhl, "Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems," *Proc. Third Int'l Workshop Peer-to-Peer Systems (IPTPS)*, Feb. 2004.

[12] Z. Zhang, S. Shi, and J. Zhu, "SOMO: Self-Organized Metadata Overlay for Resource Management in P2P DHT," *Proc. Second Int'l Workshop Peer-to-Peer Systems (IPTPS),* pp. 170-182, Feb. 2003.
[13] S. Ratnasamy, M. Handley, R.M. Karp, and S. Shenker, "Topologically-Aware Overlay Construction and Server Selection," *Proc. IEEE INFOCOM,* vol. 3, pp. 1190-1199, June 2002.
[14] Z. Xu, C. Tang, and Z. Zhang, "Building Topology-Aware Overlays Using Global Soft-State," *Proc. 23rd Int'l Conf. Distributed Computing Systems (ICDCS),* pp. 500-508, May 2003.
[15] Z. Xu, M. Mahalingam, and M. Karlsson, "Turning Heterogeneity into an Advantage in Overlay Routing," *Proc. IEEE INFOCOM,* vol. 2, pp. 1499-1509, Apr. 2003.
[16] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmaier, "Space Filling Curves and Their Use in Geometric Data Structure," *Theoretical Computer Science,* vol. 181, pp. 3-15, July 1997.
[17] Q. Lv, S. Ratnasamy, and S. Shenker, "Can Heterogeneity Make Gnutella Scalable?" *Proc. First Int'l Workshop Peer-to-Peer Systems (IPTPS),* pp. 94-103, Mar. 2002.
[18] E.W. Zegura, K.L. Calvert, and S. Bhattacharjee, "How to Model an Internetwork," *Proc. IEEE INFOCOM,* vol. 2, pp. 594-602, Mar. 1996.

Yingwu Zhu received the BS and MS degrees in computer science from Huazhong University of Science & Technology, at Wuhan, China, in 1994 and 1997, respectively. He is currently pursuing the PhD degree in computer science at the University of Cincinnati. His research interests include operating systems, storage systems, peer-to-peer computing, distributed systems, and sensor networks. He is a student member of the IEEE.

Yiming Hu received the PhD degree in electrical engineering from the University of Rhode Island in 1998. He received the BE degree in computer engineering from the Huazhong University of Science and Technology, China. He is an associate professor of computer science and engineering at the University of Cincinnati. His research interests include computer architecture, storage systems, peer-to-peer systems, operating systems, and performance evaluation. He is a recipient of a US National Science Foundation CAREER Award. He is a senior member of the IEEE and the IEEE Computer Society.