# Towards Efficient Load Balancing in Structured P2P Systems

Yingwu Zhu
Department of ECECS
University of Cincinnati
zhuy@ececs.uc.edu

Yiming Hu
Department of ECECS
University of Cincinnati
yhu@ececs.uc.edu

## Abstract

*Many solutions have been proposed to tackle the problem of load balancing in structured 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 propose an efficient scheme to ensure fair load distribution in such P2P systems by utilizing proximity information. To our knowledge, this is the first work to utilize the proximity information to guide load balancing. In particular, our main contributions are: (1) A self-organized, fully distributed $K$-nary tree structure is constructed on top of a distributed hash table (DHT) for load balancing information collection/dissemination and load reassignment. (2) Load balancing is achieved by aligning those two skews in load distribution and node capacity inhere in P2P systems — that is, have higher capacity nodes carry more loads. (3) Proximity information is utilized to guide load balancing such that loads are assigned and transferred between physically close heavy nodes and light nodes, thereby minimizing the load transferring overhead and allowing load balancing to perform more efficiently.*

## 1. Introduction

Structured P2P systems such as Chord [10], Pastry [8], Tapestry [16] and CAN [6], offer a DHT abstraction for object storage and retrieval. Due to the theoretical approach taken in these DHTs, they assume that nodes in the system are uniform in resources such as 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. Second, they build a homogeneous structure overlay network, ignoring the heterogeneity nature of P2P systems. Recent measurement studies (e.g., [9]) however have shown that node capabilities (in terms of bandwidth, storage and CPU) are very skewed in deployed P2P systems.

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 that it strives to ensure fair load distribution among all peer nodes. We therefore argue that achieving load balancing is of fundamental importance in a P2P system, due to the assumption that nodes are supposed to be uniform in resources, the resulting $O(\log N)$ ($N$ is the number of peer nodes in the system) load imbalance by a random choice of object IDs, and the fact that heterogeneous capabilities prevail among the nodes. If we could align those two skews in load distribution and node capabilities inherent in P2P networks — that is, have higher capacity nodes carry more loads — we might end up with a scalable and efficient P2P system [4].

### 1.1. Current Approaches

Current load balancing approaches have some limitations, to our opinion. They either ignore the heterogeneity of node capabilities, or transfer loads between nodes without considering proximity relationships, or both.

A solution proposed in [10] for load balancing is to have each DHT node host $O(\log N)$ virtual servers (more detail of virtual servers will be discussed in Section 2). While elegant from a theoretical perspective, virtual servers do not completely solve the load balancing issue. It still could result in $O(\frac{\log N}{\log \log N})$ imbalance, according to the standard balls and bins problem. CFS [3] accounts for node heterogeneity by having each node host some number of virtual servers in proportion to its capacity. If a node becomes overloaded, it sheds its excess loads by simply removing some of its virtual servers. Such a simple scheme however might result in load thrashing as removing some virtual servers from an overloaded node could make another node to be-

Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)



0-7695-2132-0/04/$17.00 (C) 2004 IEEE

come overloaded [5].

Recent work by Rao et al. [5] presents three simple load-balancing schemes by using the concept of virtual servers. Different form CFS, overloaded nodes split loads into virtual servers and transfer excess loads to light nodes in the unit of virtual servers. However, blindly transferring virtual servers without considering proximity relationships between heavy nodes and light nodes could incur big overhead of load transferring. For example, if virtual servers are transferred across wide-area links, it could consume huge amount of network bandwidth and introduce high latencies.

### 1.2. An Overview of Our Approach

In this paper, we propose a proximity-aware load balancing approach by using the concept of virtual servers. The goal of our approach is to not only ensure fair load distribution over nodes proportional to their capacity, but also minimize the load-balancing cost by transferring virtual servers between heavy nodes and light nodes in a proximity-aware fashion. That is, heavy nodes are trying to assign their virtual servers to physically close light nodes so that we might end up with an efficient and fast load balancing.

The load balancing approach we propose is not restricted to a particular type of resource (e.g., storage, bandwidth or CPU). We assume that the load on a virtual server is stable over the timescale it takes for the load balancing algorithm to perform. Our load balancing approach 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 capacity.
3. *Virtual server assignment (VSA)*. Determine virtual server assignment from heavy nodes to light nodes in order to have heavy nodes become light.
4. *Virtual server transferring (VST)*. Transfer assigned virtual servers from heavy nodes to light nodes. Our approach allows VSA and VST to partly overlap for fast load balancing.

### 1.3. The Contributions

To our knowledge this is the first work that approaches the load balancing issue in a proximity-aware fashion. In particular, our main contributions are:

1. A self-organized, fully distributed $K$-nary tree structure is constructed on top of a DHT for LBI collection and dissemination. Based on this structure, VSA is performed in a bottom-up sweep.
2. Load balancing is achieved by aligning those two skews in load distribution and node capacity inhere in P2P systems — that is, have higher capacity nodes carry more loads.
3. Proximity information is utilized to guide VSA such that virtual servers are assigned and transferred between physically close heavy nodes and light nodes, thereby minimizing the overhead of load transferring and allowing load balancing to perform more efficiently.

### 1.4. Paper Organization

The remainder of the paper is organized as follows. Section 2 gives a description of virtual servers. Section 3 describes the system design of our load balancing approach. Section 4 details the proximity-aware load balancing approach. In Section 5 we evaluate the load balancing approach using simulations. Section 6 provides related work. Finally we conclude in Section 7.

## 2. Virtual Servers

In this section we give a description of virtual servers. The concept of virtual servers was first introduced in Chord/CFS as a means of improving load balance. A virtual server looks like a single DHT node, responsible for a contiguous region of the DHT's identifier space. A physical DHT node can own multiple non-contiguous regions of the DHT's identifier space by hosting multiple virtual servers. Figure 1 illustrates the relationships between virtual servers and DHT nodes.

There are some advantages of using virtual servers for load balancing. First, it has flexibility in being able to move loads between DHT nodes in the unit of virtual servers. That is, the basic unit of load movement is a virtual server. We believe that such flexibility is important for any load-balancing approach over DHTs. Secondly, the movement of virtual servers can be visioned as a *leave* operation followed by a *join* operation, both of which are supported by all DHTs. As a result, the load-balancing scheme using the concept of virtual servers can be easily applied to all DHTs. In our work we address the load-balancing issue by taking advantage of virtual servers. But we also utilize proximity information to guide load balancing, thereby reducing the load-balancing overhead and making load balancing fast and efficient.





0-7695-2132-0/04/$17.00 (C) 2004 IEEE

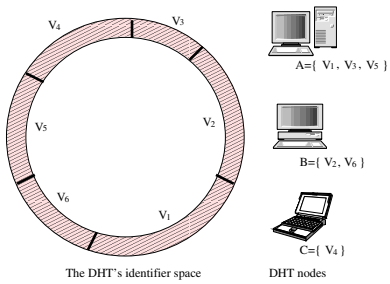


**Figure 1. Virtual servers vs. DHT nodes. $V_1$, $V_2$, $V_3$, $V_4$, $V_5$ and $V_6$ are virtual servers, each of which is responsible for a portion of the DHT's identifier space. $A$, $B$ and $C$ are DHT nodes. $A$ hosts $V_1$, $V_3$ and $V_5$. $B$ hosts $V_2$ and $V_6$, and $C$ hosts $V_4$.**

## 3 System Design

In the rest of the paper we restrict our discussion in a DHT where each node hosts multiple virtual servers.

### 3.1. Building a Distributed $K$-nary Tree

The aggregation of load balancing information (LBI) naturally leads to the construction of a tree-based structure on top of the DHT overlay. In this section we discuss how to construct a $K$-nary tree on top of DHTs for LBI aggregation/dissemination and VSA. For clarity, in the rest of the paper we refer to the $K$-nary tree node as $KT$ node while the node in the DHT overlay as DHT node or just node.

The distributed $K$-nary 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. Each $KT$ node is planted in a virtual server with a DHT key, which is produced by taking the *center* point of its responsible region. For example, suppose a $KT$ node $A$ has a responsible region $(3, 5]$ and a virtual server $S$ has a responsible region $(3, 6]$. Then the $KT$ node $A$ will be planted in the virtual server $S$, because the DHT key for $A$ is 4 by taking the center point of its responsible region $(3, 5]$. Each $KT$ node's responsible region is partitioned into $K$ equal parts, each of which is taken by its $K$ children. As a result, a $KT$ node's $i$-th child will be responsible for the $i$-th fraction of the $KT$ node's responsible region. This partitioning of the identifier space is repeated until a $KT$ node's responsible region is completely covered by that of a virtual server. In the above example, the $KT$ node $A$'s responsible region $(3, 5]$ will not be partitioned any more because its region is completely covered by that of its hosting virtual server $S$ (i.e., $(3, 6]$). So $A$ is a $KT$ leaf node. Note that it is guaranteed that a $KT$ leaf node will be planted in each virtual server. Therefore, each DHT node will host multiple $KT$ leaf nodes.

To deal with the dynamicity in P2P systems such as node addition and removal, each $KT$ node will periodically check if its responsible region is smaller or equal to that of the hosting virtual server in which it is planted. If this holds true, then this $KT$ node is already a leaf node and there is no need to grow any more children. Otherwise, it checks if each of its children's responsible region is covered by that of the hosting virtual server. If it is not true for the $i$-th child and currently this $i$-th child node does not exist, the current $KT$ node builds its $i$-th child node. This $i$-th child node will be planted into its own hosting virtual server by taking the center point of its responsible region (i.e., the $i$-th fraction of its parent's responsible region) as the DHT key. As the above procedure is performed periodically by all virtual servers, the $K$-nary tree will grow as the hosting DHT grows due to node addition. For node removal or crash, the $K$-nary tree will be able to prune itself accordingly by deleting redundant children.

As mentioned earlier, the $K$-nary tree is constructed on top of DHTs for LBI aggregation/dissemination and VSA. Such an infrastructure must have the following properties.

#### 3.1.1. Self-repair and fault-tolerant

This property is very important due to the churn in memberships as nodes join or leave at will. It is achieved due to the following facts. First, the $K$-nary tree is built atop a DHT which already has the self-organizing property. The crash of a DHT node will take away the $KT$ nodes its virtual servers are hosting. However, the responsible regions of the virtual servers of the crashing DHT node will be taken over by other virtual servers after repair. Hence, the periodical checking of all children $KT$ nodes (as described above) ensures that the $K$-nary 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 virtual server which is responsible for the center point of the whole DHT's identifier space, and it can be located deterministically. Secondly, all the states the $K$-nary tree relies on use the principle of *soft-state* and can be refreshed and reconstructed in the event of system change. Note that each $KT$ node monitors all $K$ children $KT$ nodes for faults using heartbeats sent periodically at certain time interval. Recent work [15] has shown that such a tree structure built atop of the DHT is able to be self-repair and fault-tolerant upon any failure in the system.

#### 3.1.2. Fully distributed

This property is easily achieved due to the fact that each operation in the $K$-nary tree involves at most $K + 1$ inter-

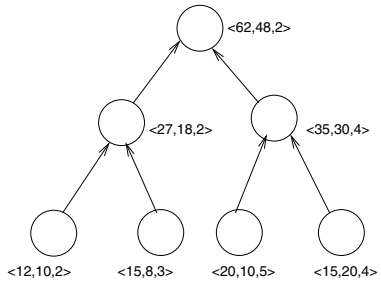


**Figure 2. LBI aggregation on a binary tree in a bottom-up sweep. The total load $L$, the total capacity $C$ and the minimum load of virtual servers $L_{min}$ in the system are 62, 48 and 2 respectively.**

actions (with a parent node and $K$ children nodes). More detail of this $K$-nary tree structure can be found in [17].

### 3.2. Load Balancing Information Aggregation

Based on the $K$-nary tree structure, LBI aggregation is quite straightforward. Each $KT$ node periodically at an interval $T$ requests LBI from their $K$ children $KT$ nodes. The $KT$ leaf node 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. So 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 $i$ 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 loads of all virtual servers, the capacity and the minimum load of virtual servers on the node $i$, respectively).

Each $KT$ leaf node reports the LBI (if any) to its parent $KT$ node. And each $KT$ node $j$ gathers all $< L_i, C_i, L_{i,min} >$s from its children nodes, by aggregating all $L_i$s and $C_i$s, and choosing the smallest load among all $L_{i,min}$s. It then reports the newly generated LBI $< L_j, C_j, L_{j,min} >$ to its parent node. This process is repeated until the $KT$ root node is reached. As a result, the $KT$ root node produces a new 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. Figure 2 exemplifies a process of LBI aggregation.

Note that the LBI aggregation is bound 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 $K$-nary tree can recover in $O(\log_K N)$ time. Hence, the LBI process can continue along the $K$-nary tree in a bottom-up sweep after the tree is reconstructed.

### 3.3. Node Classification

As described above, the $KT$ root node determines the $< L, C, L_{min} >$, which is disseminated along the $K$-nary tree in a top-down fashion to each $KT$ leaf node in $O(\log_K N)$ time. Each $KT$ leaf node then 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 $< L, C, L_{min} >$. Note that one of the goals of our load balancing approach 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 $T_i = (\frac{L}{C} + \varepsilon)C_i$ ($\varepsilon$ is a parameter for a trade-off between the amount of load moved and the quality of balance achieved. Ideally, $\varepsilon$ is 0).

Therefore, a DHT node $i$ can be defined as: (1) a *heavy node* if $L_i > T_i$. (2) a *light node* if $(T_i - L_i) \geq L_{min}$. or (3) a *neutral node* if $0 \leq (T_i - L_i) < L_{min}$.

### 3.4. Virtual Server Assignment (VSA)

Similar to the LBI aggregation, the VSA process is performed along the $K$-nary tree in a bottom-up fashion. Initially, a heavy DHT node $i$ chooses a subset of its virtual servers $\{v_{i,1}, ..., v_{i,m}\}$ ($m \geq 1$) 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 $k$-th virtual server on the DHT node $i$). This subset of virtual servers are expected to be moved to make the heavy node $i$ to become light. Therefore, *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 $< L_{i,1}, v_{i,1}, ip\_addr(i) >, ..., < L_{i,m}, v_{i,m}, ip\_addr(i) >$ to its hosted $KT$ leaf node. For a light DHT node $j$, it randomly chooses one of its virtual servers to report $< \Delta L_j = T_j - L_j, ip\_addr(j) >$ to its hosted $KT$ leaf node. All $KT$ leaf nodes in turn propagate the VSA information upwards along the tree.

Each $KT$ node collects all the VSA information from its $K$ children. It maintains two separated, sorted lists. One is composed of $< \Delta L_j = T_j - L_j, ip_addr(j) >$s sorted by $\Delta L_j$, the other is composed of $< L_{i,k}, v_{i,k}, ip_addr(i) >$s sorted by $L_{i,k}$. The former list maintains the VSA information about light nodes while the latter maintains the virtual servers expected to be assigned. If the total length of these two lists reaches a certain threshold (e.g.,30), this $KT$ node would serve as a *rendezvous point* for VSA. It chooses the virtual server $v_{i,k}$ with the heaviest load $L_{i,k}$

from the latter list, and picks a light node $j$ from the former list that minimizes $\Delta L_j$ subject to the condition that $\Delta L_j \geq L_{i,k}$. Then it removes $< \Delta L_j, ip\_addr(j) >$ and $< L_{i,k}, v_{i,k}, ip\_addr(i) >$ from the former list and the latter list, respectively. In addition, if $\Delta L_j{}' = \Delta L_j - L_{i,k} \geq L_{min}$, it re-inserts $< \Delta L_j{}', ip\_addr(j) >$ into the former list. It then sends the assigned VSA pair information directly to both DHT nodes $i$ and $j$ for virtual server transferring (VST). This VSA pairing process is repeated on this $KT$ node until the two lists become empty or no more appropriate VSA can be achieved. In the latter case, the current $KT$ node reports its unpaired VSA information to its parent node for further possible VSA. This VSA process is repeated in a bottom-up fashion along the $K$-nary tree until it reaches the $KT$ root node. Then the root node serves as the last rendezvous point (without the threshold constraint) for VSA.

In summary, the VSA process proceeds along the $K$-nary tree in a bottom-up sweep, by *recursively assigning virtual servers among DHT nodes scattered in a larger and larger contiguous DHT's identifier space* [1] *till the whole DHT's identifier space (for which the $K$-nary root node is responsible)*. In other words, the VSA process is *identifier space-based*, in that the VSA is 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 resilient to system failures as well due to the robustness of the $K$-nary tree it depends on. After the $K$-nary tree recovers, the VSA process can continue along the tree in a bottom-up fashion. Note that the VSA process is also bound in $O(\log_K N)$ time.

### 3.5. Virtual Server Transferring (VST)

The VST process is quite simple and straightforward. Upon receiving the paired VSA information, the heavy DHT node $i$ will transfer the corresponding virtual server 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 $K$-nary 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 $K$-nary 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 $K$-nary is fully distributed and inexpensive because each $KT$ node migration only involves at most $K + 1$ messages.

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

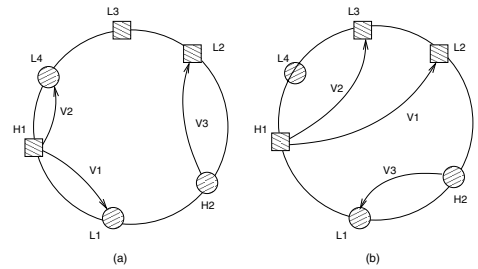


**Figure 3. Load balancing approach with/without proximity-aware mechanism. Letters "H" and "L" represents heavy nodes and light nodes respectively. Letter "v" denotes virtual servers expected to be moved to make heavy nodes to become light. Nodes with "rectangle" are physically close to each other, and nodes with "circle" are physically close to each other. Nodes with "rectangle" and nodes with "circle" are physically apart faraway.**

## 4 Proximity-Aware Load Balancing

So far we have discussed the load balancing approach without a proximity-aware mechanism, based on a $K$-nary tree built atop a DHT. In this section, we discuss in detail how to incorporate the proximity-aware mechanism into our load balancing approach. The primary goal of our approach is to utilize proximity information to guide virtual server assignment such that virtual servers are assigned and transferred between physically close heavy nodes and light nodes, thereby reducing the load balancing cost and enabling efficient and fast load balancing. As shown in Figure 3, in contrast to the proximity-ignorant load balancing approach shown in (a), the proximity-aware load balancing approach shown in (b) will assign and transfer virtual servers among physically close nodes.

### 4.1. Generating Proximity Information

Landmark clustering has been widely used to generate proximity information (e.g., [7, 13, 12]). It is based on an intuition that nodes physically close to each other are likely to have similar distances to a few selected (landmark) nodes.

In a DHT overlay network, the landmark nodes can be chosen from either the overlay itself or the Internet. For a DHT node $A$, suppose that the measured distance to a set of $m$ landmark nodes (e.g., $m = 15$) are $< d_1, d_2, ..., d_m >$, which is called the *landmark vector*. If node $A$ is 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 a result, two physically close DHT nodes $A$ and $B$ are supposed to have similar landmark vectors and be close to each other in the landmark space as well. 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.

## 4.2 Using Proximity Information

In our system, each DHT heavy/light node independently generates a landmark vector using the landmark clustering. So the proximity between heavy nodes and light nodes can be determined by their landmark vectors or their closeness in the landmark space. After generating proximity information, a big challenge we now face is how to effectively utilize this proximity information to guide load balancing to perform in a proximity-aware fashion — that is, how to guide heavy nodes to assign as many virtual servers as possible to those physically close light nodes (if any) during the VSA process until no further appropriate virtual server assignment can be achieved.

Note that the VSA process described in Section 3.4 starts from the smallest portion of the identifier space for which the $KT$ leaf node is responsible, and then proceeds in a bottom-up sweep along the $K$-nary tree, by recursively assigning virtual servers between DHT nodes scattered in a larger and larger contiguous identifier space till the whole identifier space for which the $KT$ root node is responsible. Obviously, this VSA process is *identifier space-based*, in that the VSA is performed earlier among those DHT nodes if they are closer to each other in the identifier space.

So the basic idea behind our proximity-aware load balancing approach is to *use proximity information to map physically close heavy nodes and light nodes into the identifier space so that they are also close to each other in the DHT's identifier space — that is, preserve the proximity relationships in the identifier space*. It should be pointed out that 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 map the *VSA information* of heavy nodes and light nodes into the DHT overlay [2]. For example, suppose 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). Then we map the $< L_{i,k}, v_{i,k}, ip\_addr(i) >$ and $< T_j - L_j, ip\_addr(j) >$ into the DHT overlay according to their landmark vectors such that they are also close to each other in the DHT's identifier space.

[2] In the rest of the paper, if we say "map the heavy/light nodes into the DHT" we mean "map the VSA information of the heavy/light nodes into the DHT".

Recall that in a DHT, an object is mapped into the DHT with a DHT *key* in the DHT's interface of *put(key,object)*. If two objects have similar DHT keys which are close to each other in the logical space, then these two objects are stored close to each other in the DHT overlay (or the identifier space). As discussed in Section 4.1, physically close nodes have similar landmark vectors and are close to each other in the landmark space. So the key issue is how to preserve the closeness when we map physically close nodes into the DHT. In other words, how to preserve the closeness when mapping similar landmark vectors into DHT keys?

### 4.2.1. Mapping Landmark Vectors into DHT Keys

In this subsection, we address the issue of how to preserve the closeness when mapping similar landmark vectors into DHT keys.

The first solution is that we can simply use the landmark vector of a node as a DHT key. So physically close heavy nodes and light nodes can be mapped into the DHT such that they are also close to each other in the DHT's identifier space.

However, due to the fact that the landmark space is usually of relatively high dimension compared to the DHT identifier space, we cannot adopt the solution described above. To solve this problem, we could follow the approach suggested in [13, 12] by using *space-filling curves* [1].

Space filling curves such as the Hilbert curve [1] are a class of "proximity preserving" mappings from a $m$-dimensional space to a 1-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 though of as laying out a string within the $m$-dimensional space so that it completely fills the space. The 1-dimensional mapping generated by the space-filling curve serves as an ordered indexing into multi-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 1-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*. 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.





0-7695-2132-0/04/$17.00 (C) 2004 IEEE

### 4.3. 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 $< d_1, d_2, ..., d_m >$ to a set of $m$ landmark nodes, and transforms it into a Hilbert number using the mapping described above. In the DHT of Chord, each DHT heavy/light node $i$ 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 its Hilbert number as the DHT key. Therefore, 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 described in Section 3.4 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 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 about 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 a certain threshold, it can immediately serve as a rendezvous point for the VSA. The VSA pairing process in this leaf node is repeated until no further appropriate VSA pair could be achieved. Then all paired VSA information are sent back to corresponding DHT heavy nodes and light nodes, and all unassigned VSA information (if any) are propagated up to its $KT$ parent node.

2. Otherwise, it propagates the VSA information to its $KT$ parent node.

Then the VSA process proceeds along the $K$-nary tree in a bottom-up sweep, as described in Section 3.4. Note that each $K$-nary sub-tree covers a contiguous portion of the DHT's identifier space and possibly serves as a rendezvous point for VSA, *so this bottom-up VSA process naturally guarantees that the VSA will be performed among DHT heavy nodes and light nodes, recursively in a decreasing physical closeness order as the rendezvous point for VSA moves up along the $K$-nary tree*.

In summary, the proximity-aware load balancing approach differs from the proximity-ignorant one in that it utilizes proximity information to guide VSA such that heavy nodes are able to assign and transfer their excess loads in the unit of virtual servers to physically close light nodes, thereby reducing the load balancing cost and allowing efficient and fast load balancing.

## 5. Experimental Evaluation

### 5.1. Experiment Setup

We built a $K$-nary tree (with $K$ = 2 and 8, respectively) on top of a Chord simulator (32-bit identifier space) for both LBI aggregation/dissemination and VSA.

To simulate the load of a virtual server, we considered two distributions to generate the load. Let $f$ denote the fraction of the DHT's identifier space owned by a virtual server. This fraction is exponentially distributed, which is considered to be true in both Chord and CAN. Moreover, we use $\mu$ and $\sigma$ to represent the mean and standard deviation of the the total load in the DHT system, respectively. These two distributions are: (1) *Gaussian distribution*. The load of a virtual server is generated using a Gaussian distribution with mean $\mu f$ and standard deviation $\sigma\sqrt{f}$. As suggested in [5], the Gaussian distribution would result if the load of a virtual server is attributed to a large number of small objects it stores and the individual loads on these objects are independent. (2) *Pareto distribution*. The load of a virtual server is generated using a Pareto distribution with the shape parameter $\alpha = 1.5$ and mean $\mu f$. The standard deviation is infinite.

To account for heterogeneity in node capacity, we used a *Gnutella-like* capacity profile. We assigned capacity of $1, 10, 10^2, 10^3$ and $10^4$ to DHT nodes with probability of $20\%, 45\%, 30\%, 4.9\%$ and $0.1\%$, respectively.

We evaluated the proximity-aware load balancing using two transit-stub topologies with approximately $5,000$ node each, produced by GT-ITM [14]. Both topologies have 10 graphs each and we ran all these graphs in our simulation. They are: (1) "ts5k-large" has 5 transit domains, 3 transit nodes per transit domain, 5 stub domains attached to each transit node, and 60 nodes in each stub domain on average. (2) "ts5k-small" has 120 transit domains, 5 transit nodes per transit domain, 4 stub domains attached to each transit node, and 2 nodes in each stub domain on average.

We chose "ts5k-large" to represent 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 vector and then generated the Hilbert number as the DHT key for each Chord node.





0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Nodes in a stub domain have close (or even same) Hilbert numbers. We expect that the proximity-aware load balancing could perform very well in the "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

In all experimental results we present in this paper, the Chord overlay consists of 4096 nodes each with 5 virtual servers in the beginning, and the degree of the $K$-nary tree is 2 (we observed similar results on the degree of 8).

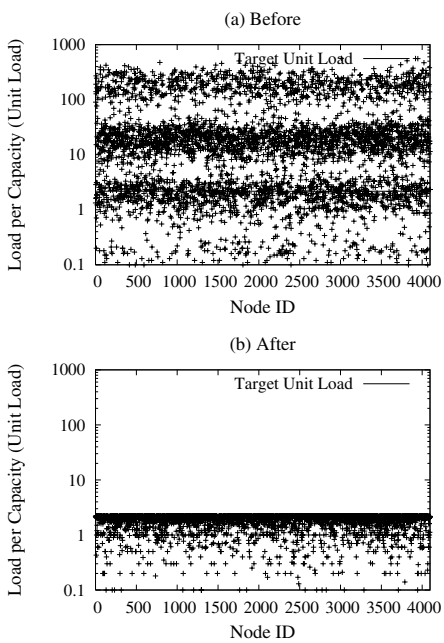


**Figure 4. Results for the Gaussian distribution.**

We first present the result of the load balancing in aligning those two skews in load distribution and node capacity. We ran our experiments based on two $K$-nary trees with $K = 2$ and $K = 8$, respectively. We found that, in both configurations, VSA completes quickly in $O(\log_K N)$ time.

Figure 4 shows the results for the Gaussian distribution. The x-axis represents the Chord nodes numbered from 1 to 4096, while the y-axis denotes the load per capacity (called *unit load*) in a DHT node. Figure 4 (a) shows a scatterplot of the unit load for each Chord node before load balancing. The percentage of heavy nodes are about *75%*. Figure 4 (b) shows a scatterplot of the unit load for each Chord node after load balancing. Note that all heavy nodes become light by transferring excess loads to light nodes.

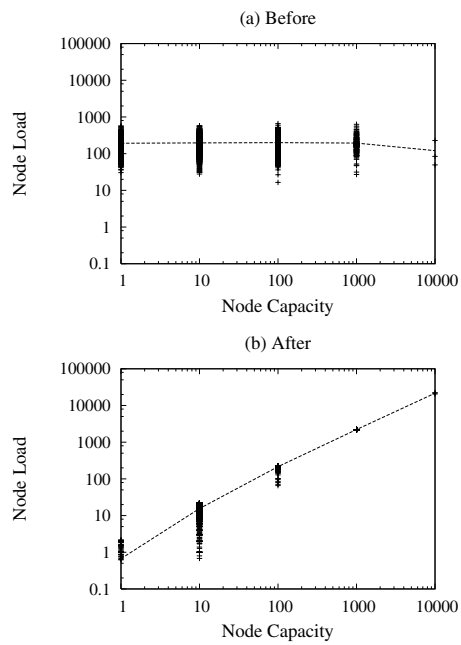


**Figure 5. Results for the Gaussian distribution, according to node capacity categories.**

Figure 5 and Figure 6 show the scatterplot of loads according to node capacity categories for the Gaussian distribution and Pareto distribution, respectively. Note that our load balancing approach is able to align those two skews in load distribution and node capacity inherent in P2P systems — that is, have higher capacity nodes take more loads.

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? To answer this question, we evaluated the benefit in terms of virtual server transferring cost — that is, the less transferring cost, the more benefit.

Figure 7 shows the results of the load balancing approach with/without the proximity-aware mechanism for "ts5k-large". The x-axis denotes the distance of virtual server transferring in terms of hops, while the y-axis represents the percentage of total moved load. Figure 7 (a) shows the distribution of moved load while Figure 7 (b) shows the CDF of the moved load distribution. We can see that the proximity-aware load balancing approach is able to transfer

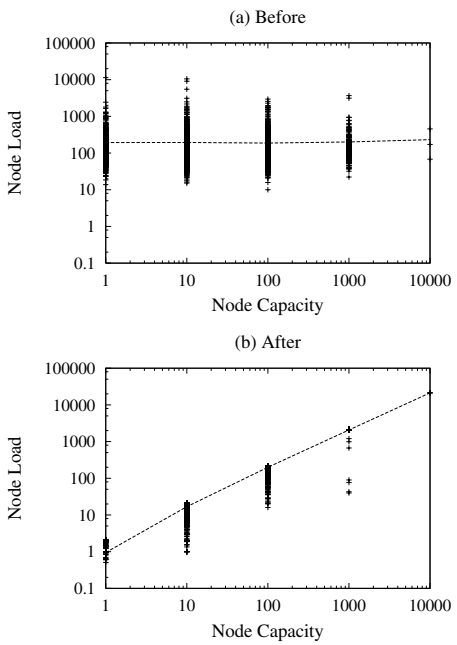


**Figure 6. Results for the Pareto distribution, according to node capacity categories**

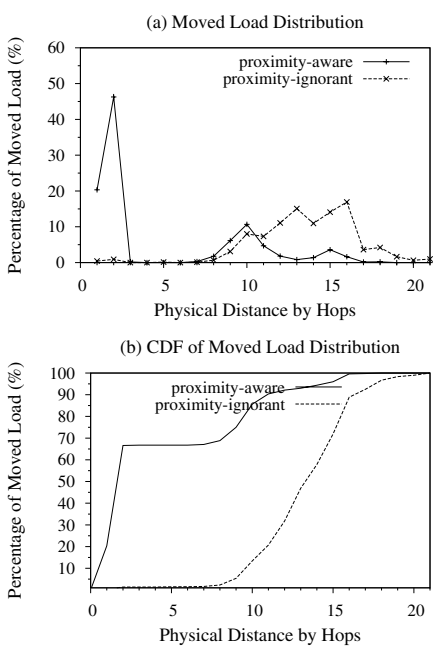


**Figure 7. Results for "ts5k-large".**

about 67% of total moved load within 2 hops and transfer about 86% within 10 hops. The proximity-ignorant load balancing instead transfers only about 13% of total moved load within 10 hops. Such a big difference implies that the proximity-aware load balancing approach can effectively assign and transfer loads between physically close nodes, thereby reducing the load balancing cost (e.g., bandwidth consumption) and enabling fast and efficient load balancing.

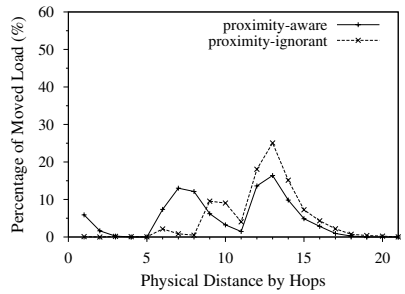


**Figure 8. Results for "ts5k-small".**

Figure 8 shows the moved load distribution 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 approach still performs much better than the proximity-ignorant load balancing approach. This is because the proximity-aware load balancing algorithm 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 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 transferring cost by making appropriate VSA among physically close nodes. Due to space constraints, more detailed experimental results can be found in [17].

## 6. Related Work

Load balancing in structured P2P systems [10, 8, 16, 6] is addressed in a rather naive way by simply resorting to the uniformity of the hash function. CFS [3] take into account node heterogeneity to tackle the load balancing problem by having each node host some number of virtual servers in proportion to its capacity. Rao et al. [5] propose three schemes to approach the load balancing issue by transferring load from heavy nodes to light nodes in the unit of virtual servers. Recent work [2] approaches the load balancing issue in DHTs from a different viewpoint. It proposes using the "power of two choices" paradigm to achieve load balancing. Triantafillou et al. [11] present a novel architecture

to ensure fair load distribution and efficient operations in the context of content and resource management. But their work is targeted for unstructured P2P file sharing systems.

The $K$-nary tree structure in our work is similar to the *metadata overlay* proposed in [15]. 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 assignment.

Proximity information has been utilized in both topologically aware DHT construction [7] and proximity neighbor selection in P2P routing tables [13, 12]. 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.

## 7. Conclusions

In this paper we propose an efficient load balancing scheme to tackle the issue of load balancing in structured 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 higher capacity nodes carry more loads. The second goal is to use proximity-aware information to guide load assignment 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, a Gnutella-like node capacity profile and two Internet topologies. The results show that our proximity-aware load balancing approach can not only ensure fair load distribution but also minimize the load transferring overhead. Further, virtual server assignment can be bound in $O(\log N)$ time.

## References


[1] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmaier. Space filling curves and their use in geometric data structure. *Theoretical Computer Science*, 181(1):3–15, July 1997.

[2] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2003.

[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Banff, Canada, Oct. 2001.

[4] Q. Lv, S. Ratnasamy, and S. Shenker. Can heterogeneity make gnutella scalable? In *Proceedings of the first International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002.

[5] A. Rao, K. Lakshminarayanan, R. K. Sonesh Surana, and I. Stoica. Load balancing in structured p2p systems. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2003.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, San Diego, CA, Aug. 2001.

[7] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologicallyaware overlay construction and server selection. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, New York, NY, June 2002.

[8] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed System Platforms (Middleware 2001)*, Heidelberg, Germany, Nov. 2001.

[9] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, San Jose, CA, Jan. 2002.

[10] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, CA, Aug. 2001.

[11] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Towards high performance peer-to-peer content and resource sharing systems. In *Proceedings of CIDR*, Jan. 2003.

[12] Z. Xu, M. Mahalingam, and M. Karlsson. Turning heterogeneity into an advantage in overlay routing. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Apr. 2003.

[13] Z. Xu, C. Tang, and Z. Zhang. Building topology-aware overlays using global soft-state. In *The 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, 2003.

[14] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM*, volume 2, pages 594–602, San Francisco, CA, Mar. 1996. IEEE.

[15] Z. Zhang, S. Shi, and J. Zhu. SOMO: self-organized metadata overlay for resource management in p2p DHT. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2003.

[16] 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, University of California, Berkeley, Apr. 2001.

[17] Y. Zhu and Y. Hu. Efficient, proximity-aware load balancing for structured peer-to-peer systems. Technical Report TR-270/04/03/ECECS, University of Cincinnati, 2003.