Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 68 (2008) 686–702
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

# Hash-based proximity clustering for efficient load balancing in heterogeneous DHT networks

Haiying Shen[a,*], Cheng-Zhong Xu[b]

[a]*Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR 72701, USA*
[b]*Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202, USA*

Received 15 May 2007; received in revised form 5 November 2007; accepted 5 November 2007
Available online 12 November 2007

**Abstract**

Distributed hash table (DHT) networks based on consistent hashing functions have an inherent load uneven distribution problem. The objective of DHT load balancing is to balance the workload of the network nodes in proportion to their capacity so as to eliminate traffic bottleneck. It is challenging because of the dynamism, proximity and heterogeneity natures of DHT networks and time-varying load characteristics.

In this paper, we present a hash-based proximity clustering approach for load balancing in heterogeneous DHTs. In the approach, DHT nodes are classified as regular nodes and supernodes according to their computing and networking capacities. Regular nodes are grouped and associated with supernodes via consistent hashing of their physical proximity information on the Internet. The supernodes form a self-organized and churn-resilient auxiliary network for load balancing. The hierarchical structure facilitates the design and implementation of a locality-aware randomized (LAR) load balancing algorithm. The algorithm introduces a factor of randomness in the load balancing processes in a range of neighborhood so as to deal with both the proximity and dynamism. Simulation results show the superiority of the clustering approach with LAR, in comparison with a number of other DHT load balancing algorithms. The approach performs no worse than existing proximity-aware algorithms and exhibits strong resilience to the effect of churn. It also greatly reduces the overhead of resilient randomized load balancing due to the use of proximity information.


*Keywords:* Load balancing; Peer to Peer; Distributed hash table

## 1. Introduction

Distributed hash table (DHT) network is a content-addressable overlay network that maps files to each network node based on a consistent hashing function. Due to its salient feature of robustness, DHT network has received much attention in the past several years. Early studies have resulted in numerous DHT networks with various routing characteristics [22,24,27,28,35]. A downside of consistent hashing is uneven load distribution. In theory, consistent hashing produces a bound of $O(\log n)$ imbalance of file keys between nodes, where $n$ is network size [16]. In addition, factors like non-uniform file size, time-varying file popularity and node heterogeneity in capacity make the load balancing problem even more severe in practice.

The objective of DHT load balancing is to balance the workload of the nodes in proportion to their capacity so as to eliminate traffic bottleneck. The workload of a node can be measured in terms of metrics like file size and traffic volume incurred in the access to the files. Load balancing in DHT networks remains challenging because of their two unique features:

- *Dynamism*: A defining characteristic of DHT networks is dynamism/churn. A great number of nodes join, leave and fail continuously and rapidly, leading to unpredicted network size. A load balancing solution should be able to deal with the effect of churn. Popularity of the items may also change over time. A load balancing solution that works for static situations does not necessarily guarantee a good performance in dynamic scenarios. Skewed query patterns may also

* Corresponding author. Fax: +1 479 575 5339.
*E-mail addresses:* hshen@uark.edu (H. Shen), czxu@wayne.edu (C.-Z. Xu).

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.10.005

result in considerable number of visits at hot spots, hindering efficient item access.

- *Proximity*: A load balancing solution tends to utilize proximity information to reduce the load balancing overhead. However, logical proximity abstraction derived from DHTs does not necessarily match the physical proximity information in reality. This mismatch becomes a big obstacle for the deployment and performance optimization of peer-to-peer (P2P) applications.

In addition, DHT networks are often highly heterogeneous. With the increasing emergence of diversified end devices on the Internet equipped with various computing, networking and storage capabilities, the heterogeneity of participating peers of a practical P2P system is pervasive. This requires a load balancing solution not only to distribute the application load (e.g. file size, access volume), but also the load balancing overhead among the nodes in proportion to their capacities.

There are recent studies devoted to the DHT load balancing problem [14,17,26,28,36]. "Virtual nodes" [14,28,36] and "item movement" [17] are two general approaches for load balancing in heterogeneous DHTs. They focus on the distribution of application load between the network nodes in proportion to their capacities. Rao et al. [21] and Godfrey et al. [14] proposed randomized load balancing algorithms for load reassignment in DHTs with churn. The algorithms treat all nodes equally in random probing for lightly (or heavily) loaded nodes, without consideration of node proximity information in load balancing. Zhu and Hu presented a proximity-aware algorithm to take into account the node proximity information in load balancing [36]. The algorithm is based on an additional $k$-ary tree network constructed on top of Chord. Although the network is self-organized, it needs extra cost for reconstruction after every load transfer, and the load balancing algorithm is hardly applicable to DHT with churn. In [26], Shen and Xu proposed locality aware randomized (LAR) load balancing algorithms to deal with both of the proximity and dynamic features of Cycloid-structured DHTs. It introduces a factor of randomness in the probing process in a range of proximity to handle the effect of churn. Cycloid is a constant-degree DHT based on the network topology of cube connected cycle. Its hierarchical structure facilitates the implementation of the LAR algorithms.

In this paper, we present a hash-based proximity clustering approach to deal with both the proximity and dynamic features of DHTs, and take advantage of heterogeneity as well. The clustering approach distinguishes between supernodes and regular nodes according to the nodal capacities and constructs an auxiliary supernode network for load balancing. The novelty of the approach lies in the construction of the auxiliary network. Existing proximity clustering approaches often designate static gateways or routers of regular nodes as their supernodes [13,34]. In contrast, we cluster the nodes and associate them to supernodes by consistent hashing of their physical proximity information. Supernodes are designated dynamically according to their capacities and consistent hashing incurs little re-association of regular nodes to the supernodes as nodes join and leave the system. The auxiliary supernode network can be physical or virtual. It facilitates the design and implementation of efficient and churn-resilient LAR load balancing algorithm. The algorithm takes advantage of the proximity information of the DHTs in node probing and distributes application load among the nodes according to their capacities. We evaluated the performance of the clustering approach with the LAR load balancing algorithm via comprehensive simulations. Simulation results demonstrate the superiority of the approach in comparison with a number of other DHT load balancing algorithms.

The rest of this paper is structured as follows. Section 2 presents a concise review of representative load balancing approaches for DHT networks. Section 3 details hash-based proximity clustering to construct a auxiliary network to facilitate load balancing with churn, proximity and heterogeneity considerations. Section 4 presents LAR load balancing algorithm and how it is implemented on the auxiliary network. Section 5 shows the performance of the hash-based proximity clustering approach for load balancing in terms of a variety of metrics in Chord with and without churn. Finally, Section 6 concludes this paper with remarks on possible future work.

## 2. Related work

DHT networks have an inherent load balancing problem due to the use of consistent hashing functions for key ID range partitioning [16]. Node heterogeneity in P2P networks makes the load balancing problem even more severe. To alleviate the problem, Stoica et al. [28] proposed an abstraction of "virtual servers", in which each real node runs $\Omega(\log n)$ virtual servers, and the keys are mapped onto virtual servers so that each real node is responsible for $O(1/n)$ of the key ID space with high probability. The "virtual server"-based approach for load balancing is simple in concept. There is no need for the change of underlying DHTs. However, the abstraction incurs large space overhead and compromises lookup efficiency. The storage for each real server increases from $O(\log n)$ to $O(\log^2 n)$ and the network traffic increase considerably by a factor of $\Omega(\log n)$. Brighten et al. [15] addressed the problem by arranging a real server for virtual ID space of consecutive virtual IDs. This reduces the load imbalance from $O(\log n)$ to a constant factor. Karger and Ruhl [17] coped with the "virtual server" problem by arranging for each real node to activate only one of its $O(\log n)$ virtual servers at any given time. The real node occasionally checks its inactive virtual servers and may migrate to one of them if the distribution of load in the system has changed.

Another group of algorithms achieves load balance by initial location allocation for a item or a node. Byers et al. [8] uses "power of 2 choices" method to store an item in the least loaded node among more than 1 choices leading to $\log\log n/\log d + O(1)$ imbalance. On the other hand, other works [1,17,18,20] let a joining node have $\Theta(\log n)$ ID choices and choose the ID resulting in the best load balance. The approach achieves $O(1)$ imbalance.

Initial key ID space partitioning is insufficient to guarantee load balance, especially in DHTs with churn. It is often needed

to be complemented by dynamic approach. Most recently, Bienkowski et al. [7] proposed a node leave and rejoin strategy to balance the key ID intervals across the nodes. In the algorithm, lightly loaded nodes (with short intervals) may leave the system and rejoin to share the load of heavy ones. In a constant number of rounds, the algorithm achieves optimal balance with high probability in theory. Ganesan et al. [12] proposed schemes to let nodes with *short* ID space share to get more share from nodes with long ID space share. Other works proposed load information aggregation schemes, such as distributed approximative system information service [3], histograms [6] and partial tree [33], to help joining nodes to partition heavily loaded node's item set, or to move lightly loaded nodes to the location of the heavily loaded node. However, the above methods ignore the heterogeneity nature of nodes or items by assuming same capacity of each node or same load of each item.

With the consideration of the heterogeneity, Rao et al. [21] proposed three schemes to rearrange load based on different capacities of nodes. Their basic idea is to move load from heavy nodes to light nodes so that each node's load does not exceed its capacity. Their schemes are different primarily in the amount of information used to decide rearrangement. In a one-to-one rendezvous scheme, each light server randomly probes nodes for a match with a heavy one. In a many-to-many scheme, each heavy server sends its excess virtual nodes to a global directory, which executes rearrangement periodically. One-to-many scheme works in a way that each heavy server randomly chooses a directory which contains information about a number of light servers. Based on this work, Godfrey et al. [14] developed churn-resilient algorithm (CRA) for dynamic DHTs with rapid arrivals and departures of items and nodes. In this work, when a node's actual load divided by its capacity exceeds a predetermined threshold, its excess virtual nodes will be moved to light ones immediately without waiting for next periodic balancing. In the load reassignment schemes, a directory is stored in the node which is responsible for the directory ID hash value. The node itself may be heavily loaded and does not have sufficient capacity for directory management and load rearrangement.

An alternative to "virtual server" migration is "item movement". Karger and Ruhl [17] proved that the "virtual server" method could not be guaranteed to handle item distributions where a key ID interval of length $p$ has more than a $\omega(pl)$ fraction of the load ($l$ represents the maximum number of virtual locations of each node). As a remedy, they proposed an item moving scheme, in which every node occasionally contacts a random other node and moves items between the nodes for load balancing. In contrast to the "virtual server" approach, the item moving scheme keeps P2P network scalability and efficiency. Ahmad and Ghafoor [2] proposed a semi-distributed approach for load balancing in large parallel and distributed systems. This method uses a two-level hierarchical control by partitioning the interconnection structure of a distributed or multiprocessor system into independent symmetric spheres centered at schedulers. The schedulers optimally schedule tasks within their spheres and maintain state information within low overhead.

Note that the load reassignment schemes assumed a goal of minimizing the amount of load moved. It neglects the effect of load moving distance, a main attributing factor to the overhead requirement for load balancing. Virtual servers or items should be transferred between physically close heavy nodes and light nodes with proximity consideration. In addition, node communication in load balancing is another such attributing factor.

One of the early works to utilize the proximity information to guide load balancing is due to Zhu and Hu [36]. They suggested to build a $k$-ary tree structure on top of a DHT overlay. Each tree node is planted in a virtual server. A tree node reports its real server load information to its parent, until the tree root is reached. The root then disseminates final information to all the virtual nodes. Using this information, each real server can determine whether it is heavily loaded or not. Light and heavy nodes report their free capacity, excess virtual node information to their leaf nodes, respectively, and the information will be propagated upward along the tree. When the total length of information reaches a certain threshold, the tree node would execute load rearrangement. This approach uses proximity information to map physically close heavy and light nodes into the ID space. We note that the $k$-ary tree-based load balancing approach has three drawbacks. First, the tree construction and maintenance are costly, especially in DHTs with churn. In churn, without timely fixes, a tree will be destroyed, degrading load balancing efficiency. For example, when a parent fails or leaves, the load imbalance of its children in the subtree cannot be solved before its recovery. Besides, the tree needs to be reconstructed every time after virtual server transferring, which is imperative in load balancing. Second, a real server cannot start to determine its load condition until the tree root gets the accumulated information from all nodes. This centralized process is inefficient and hinders the scalability improvement of P2P systems. Third, it does not distribute load balancing overhead based on node capacity since all nodes are participating in load balancing, and some nodes may have not enough capacity for load information forwarding and load rearrangement.

Shen and Xu proposed LAR load balancing algorithms to take advantage of the hierarchical structure of Cycloid to cope with both dynamism and proximity [26]. This paper applies the concept of proximity-aware randomization for load balancing in general heterogeneous DHTs. A key component is proximity clustering that distinguishes between regular nodes from neighboring high-capacity supernodes and builds a self-organized churn-resilient hierarchical structure to take advantage of the network heterogeneity and make use of the proximity information in load balancing.

## 3. Hash-based proximity clustering

In general, supernodes are nodes with high capacity and fast connections and regular nodes are nodes with low capacity and slower connections. For simplicity, we define a node with capacity greater than a predefined threshold as supernode; otherwise a regular node.

Supernode network in DHTs is an auxiliary expressway for fast routing between the supernodes. Each supernode operates

as a server to its associated regular nodes. The supernode networks proposed in [13,34] take proximity into account by clustering physically close nodes into one group. They take static gateways (or routers) as supernodes. Network tools for finding gateway, such as traceroute, are too heavy weight and intrusive for use by large-scale applications because they generate excessive load on the network. Xu et al. [30] proposed to use landmark clustering to generate proximity information. The proximity information of physically close nodes is stored in the same or nearby nodes. Based on the proximity information, supernodes are connected in an auxiliary expressway for fast routing. Their expressway construction is constrained by the logical overlay topology. For a supernode, its direct neighbors are limited to those supernodes in the desired portion of its ID space. The resultant partially connected expressway does not make full use of heterogeneity and proximity. Propagating information in the expressway about node join and departure and the network condition changes may lead to high maintenance cost. Our proximity clustering approach bears resemblance to landmark clustering, in that the nodes are partitioned into groups according to landmark proximity information. But our hash-based proximity clustering approach constitutes all supernodes into a self-organized and churn-resilient DHT for load balancing.

A crucial step in proximity-aware load balancing is to gather load information of physically close nodes in a supernode and ensure that workload be transferred between physically close heavy nodes and light nodes. Hash-based proximity clustering generates a resilient auxiliary supernode network, in which supernodes process load balancing on behalf of their assigned regular nodes. The interconnections between the supernodes and their associated regular nodes can be defined by their routing tables. We distinguish the interconnections into two forms: physical and virtual. A physical cluster, denoted by *p*Cluster, is a structure in which each node is connected to its physical closest supernode and all supernodes form a DHT. A virtual cluster, denoted by *v*Cluster, is a structure in which each node is connected to logically closest supernode in their ID space.

Before we present the details of the auxiliary networks, let us introduce a landmarking method to represent node closeness on the Internet by indices. Landmark clustering has been widely adopted to generate proximity information [23,30,31]. It is based on the intuition that nodes close to each other are likely to have similar distances to a few selected landmark nodes, although details may vary from system to system. In DHTs, the landmark nodes can be selected by overlay itself or the Internet. We assume $m$ landmark nodes that are randomly scattered in the Internet. Each node measures its physical distances to the $m$ landmarks, and uses the vector of distances $\langle d_1, d_2, \ldots, d_m \rangle$ as its coordinate in Cartesian space. Two physically close nodes will have similar landmark vectors. Note that a sufficient number of landmark nodes are needed to reduce the probability of false clustering where nodes that are physically far away have similar or close landmark vectors.

We use space-filling curves [4], such as Hilbert curve as in [30,31], to map $m$-dimensional landmark vectors to real numbers. That is, $R^m \longmapsto R^1$, such that the closeness relationship among the points is preserved. This mapping can be regarded as filling a curve within the $m$-dimensional space till it completely fills the space. We partition the $m$-dimensional landmark space into $2^{mx}$ grids of equal size (where $m$ refers to the number of landmarks and $x$ controls the number of grids used to partition the landmark space), and number each node according to the grid into which it falls. We call this number *Hilbert number* of the node. The Hilbert number indicates the degree of physical closeness between nodes on the Internet. The smaller the $x$, the larger the likelihood that two nodes will have same Hilbert number, and the coarser grain the physical proximity information.

### 3.1. Physical clustering

*p*Cluster consists of clusters, and all nodes are physically close to each other within each cluster. Each cluster has a supernode, together with a group of regular nodes, and the supernode operates as a server to the others.

In *p*Cluster, a supernode DHT is constructed on top of the original DHT. We directly use a node's Hilbert number as its logical node ID and let supernodes act as the top-level supernode DHT nodes and regular nodes as top-level supernode DHT keys. The top-level supernode DHT can be any type of DHT such as Chord, Pastry, Tapestry, CAN or Cycloid, with a variant of consistent hashing key assignment protocol. By the protocol, a key is stored in a node whose ID is the closest to the key so that a regular node is assigned to a supernode whose ID is closest to the node's ID; that is, regular nodes are connected to their physically closest supernode since node ID represents node physical location closeness. As a result, the physically close nodes will be in the same cluster or in nearby clusters with supernodes. In the case when a number of supernodes have the same Hilbert numbers, one supernode is chosen and others become its backups. The consistent hashing for key assignment protocol requires relatively little re-association of regular nodes to dynamically designated supernodes as nodes join and leave the system.

**Algorithm 1.** Pseudocode for routing algorithm in *p*Cluster based on Chord lookup algorithm.

```
1: //Ask node n to find the supernode whose ID is closest to id
2: n.find_supernode(id) {
3: if supernode!=null then
4:    //n is a regular node
5:    supernode.find_supernode(id);
6: else
7:    //n is a supernode
8:    successor=find_successor(id);
9:    predecessor=successor.predecessor;
10:   //return the closer node to ID
11:   if predecessor's ID is closer to id than successor's ID then
12:      return predecessor;
13:   else
14:      return successor;
15:   end if
16: end if
17: }.
```

We use a "proximity-neighbor selection" technique as described in [9,29,31] to build each supernode's routing table in the supernode DHT. That is, it selects the routing table entries pointing to the physically nearest among all nodes with IDs in the desired portion of the ID space. Since Hilbert numbers represent node physical location closeness, the top-level supernode DHT in *p*Cluster preserves supernode physical proximity in logical ID space. As a result, nodes in one cluster are physically close to each other, close clusters/supernodes in logical ID space are also physically close to each other, and the application-level connectivity between the supernodes in the top-level supernode DHT is congruent with the underlying IP-level topology.

To find a supernode responsible for an ID, a regular node forwards a query to its supernode, and the routing algorithm on supernode DHT is the same as the DHT routing algorithm. Algorithm 1 shows pseudocode of *p*Cluster routing algorithm based on the Chord routing algorithm `find_successor(id)` in [28]. DHT protocols dealing with node and item joins and departures can be directly used to handle supernode and regular node joins and departures in supernode DHT. When a supernode or regular node joins the supernode DHT, it must know at least one node, and use *p*Cluster routing algorithm to find its place in *p*Cluster. To maintain the mapping between regular nodes and supernodes, when a supernode $s$ joins the *p*Cluster, certain regular nodes previously assigned to $s$'s successor or predecessor now become assigned to $s$ if $s$ is closer to them than their current supernodes. When supernode $s$ leaves the *p*Cluster, all of its assigned regular nodes are reassigned to $s$'s successor or predecessor based on their closeness to its regular nodes. No other changes in assignment of regular nodes to supernodes need occur. Algorithms 2 and 3 show the pseudocode of node join and departure in *p*Cluster, respectively.

**Algorithm 2.** Pseudocode for node joining in *p*Cluster containing node $n'$.

```
1: //For a new arrival node n joining pCluster containing node n'
2: n.join(n'){
3: ID=n.Hilbertnum;
4: //find the supernode closest to n
5: s = n'.find_supernode(n.ID);
6: if n's capacity<a predefined threshold then
7:    //n is a regular node, taking s as its supernode
8:    supernode=s;
9:    supernode.addto_clientlist(n);
10: else
11:    //n is a supernode
12:    if n.ID==s.ID then
13:       s.addto_backuplist(n);
14:    else
15:       //join in supernode DHT, initialize neighbors
16:       predecessor=nil;
17:       //find its successor
18:       if s.ID%2^d >n.ID%2^d then
19:          successor=s;
20:       else
21:          successor=s.successor;
22:       end if
23:    end if
24: end if
25: }.
```

**Algorithm 3.** Pseudocode for node leaving *p*Cluster.

```
1: //For a node n leaving pCluster
2: n.leave(){
3: if supernode!=nil then
4:    //n is a client
5:    supernode=nil;
6: else
7:    //n is a supernode
8:    if backuplist.size>0 then
9:       //choose one backup, transfer supernode information to it
10:      s=backuplist.getone();
11:      n moves routing table to s;
12:      n moves backup list to s;
13:      //notify its clients to change their supernode
14:      for i = 0 up to clientlist.size do
15:         client=clientlist[i];
16:         supernode_change_notify(client);
17:      end for
18:   else
19:      //no backup supernode, transfer regular nodes accordingly
20:      for i = 0 up to clientlist.size do
21:         client=clientlist[i];
22:         if predecessor is closer to client than successor then
23:            move client to predecessor;
24:         else
25:            move client to successor;
26:         end if
27:      end for
28:   end if
29: end if
30: }.
```

Fig. 1(a) shows an example of *p*Cluster in Chord. By taking Hilbert numbers as their ID and key assignment protocol, physically close nodes are grouped into a cluster with a supernode and all supernodes constitute a Chord. Each supernode functions as a node in a flat Chord. If $n40$ wants to join in the *p*Cluster, $n40$ asks its known node $n2$ to find the supernode with ID closest to 40 based on Algorithm 1, which is $n45$. If $n40$ is a supernode, $n45$ moves $n41$ to $n40$. The maintenance of supernode DHT is the same as that of Chord. The joining execution does not make the rest of the network aware of $n40$. It is the responsibility of stabilization to build routing table and other links for $n40$, and to update other supernode routing tables and other neighbors on supernode DHT. If $n40$ is a regular node, it becomes a client of $n45$. If a node, say $n45$, wants to leave the system, according to Algorithm 3, it moves $n41$ to $n34$ and $n50$ to $n63$. The routing tables which have $n45$ will be updated in stabilization. If $n41$ wants to leave the *p*Cluster, it only needs to disconnect its link to $n45$.

Node failure is an important problem in DHT since it leads to intact topology and degrades DHT performance. As in flat DHT, *p*Cluster uses stabilization to deal with supernode failures in the top-level supernode DHT. In Chord, each supernode refreshes its routing table entries and predecessor periodically to make sure they are correct. We use lazy update to handle the influence of a supernode failure on its regular nodes. Each regular node probes its supernode periodically. If a regular node $n$ does not

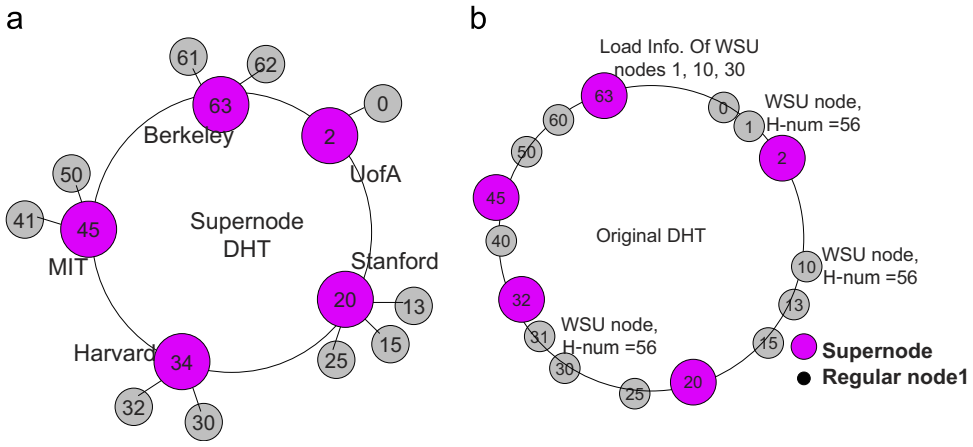



Fig. 1. Example of proximity-aware DHTs. (a) *p*Cluster and (b) *v*Cluster.

get a reply from its supernode $s$ after certain time period $T$, $n$ assumes $s$ fails, it uses *p*Cluster node join protocol to connect to another supernode again. For example, if $n41$ does not get reply from $n45$ after $T$, by joining algorithm, it will connect to $n32$.

To use *p*Cluster for load balancing, each node periodically reports its load information to its supernode. As a result, the load information of physically close nodes gathers together in the supernode. For example, nodes $n61$ and $n62$ report their load information to $n63$ periodically, which does load rearrangement, and notify heavy nodes to move excess load to light nodes.

### 3.2. Virtual clustering

Physical clustering constructs a top-level supernode DHT in the routing tables of the nodes. In contrast, virtual clustering constructs a perception of supernode DHT, *v*Cluster, by recording the proximity information in the original DHT network. That is, *v*Cluster assigns regular nodes to their *logically* closest supernodes in ID space as usual. Although nodes in the same cluster are not necessarily physically close, physically close nodes will report their load information to a same supernode or physically close supernodes in the load balancing process. Algorithms 4 and 5 show the pseudocode of node join and departure in *v*Cluster, respectively. They ensure that a regular node always connects to the supernode whose ID is closest to its ID. Like *p*Cluster, *v*Cluster also uses lazy update to handle supernode failure. Without an additional structure, *v*Cluster does not need any other extra construction and maintenance cost.

A question is how to gather load information of physically close nodes into a same supernode. Recall that, in a DHT, an object with a DHT key is allocated to a node by the interface of `put(key,object)`. In Chord, the object is assigned to the first node whose ID is equal to or follows the key in the ID space. If two objects have similar keys, then they are stored in close nodes in ID space. Because Hilbert numbers represent node physical proximity, if nodes put their load information to the DHT with their Hilbert number as the key by `put(HilbertNum,loadInfo)`, load information of physically close nodes with similar Hilbert numbers will reach the same node or nearby nodes. The nodes further forward the information to their supernodes. Fig. 1(b) shows an example of *v*Cluster in Chord. In the example, regular nodes $n1$, $n10$ and $n30$ send their load information to the DHT with their Hilbert number 56 as destination. The information will first arrive at $n60$ and then is forwarded to $n63$. The $n63$ does load rearrangement between physically close nodes $n1$, $n10$ and $n30$.

### 3.3. pCluster versus vCluster

Both *p*Cluster and *v*Cluster facilitate locality-aware load balancing. They achieve the goal in different ways. In *p*Cluster, the load information of the nodes with same Hilbert number $h$ is gathered in a supernode whose Hilbert number is closest to $h$. In *v*Cluster, the load information will be gathered in a supernode whose ID is closest to $h$. Therefore, in the case that node $n$ reports its load information to supernode $s$, $s$ is $n$'s physically closest supernode in *p*Cluster; in *v*Cluster, $s$ is not $n$'s physically closest node, and it may be even far away from $n$ because of the inconsistence between logical topology and underlying physical topology. Similarly, after a supernode completes load reassignment, its notification to transfer load may also need to travel a long distance in *v*Cluster. As a result, the communication cost of *p*Cluster load balancing would be less than *v*Cluster load balancing. This advantage of *p*Cluster load balancing is gained at the cost of supernode DHT construction and maintenance. In a conclusion, *p*Cluster load balancing can save communication cost in load balancing and speed up load balancing. In contrast, *v*Cluster can save storage space and cost for supernode DHT construction and maintenance.

Table 1 shows auxiliary network suitable for load balancing in each type of DHT networks. Geographic layout proximity-aware DHTs preserve physical closeness of nodes in logical topology. In this DHT type, the *p*Cluster load balancing can be used directly in supernode DHTs. On the other hand, in flat DHTs, the feature that the physically close nodes are also close in logical ID space can be taken advantage of to build *p*Cluster without additional proximity clustering cost. Because nodes report their load information to their supernodes at less communication cost than to the supernodes whose logical IDs are closest to their Hilbert numbers, *p*Cluster is more suitable to flat networks in geographic layout proximity-aware DHTs.

**Algorithm 4.** Pseudocode for node joining in $v$Cluster containing node $n'$.

```
1: //For a new arrival node n joining Chord containing node n'
2: n.joinChord(n'){
3: predecessor=nil;
4: successor=n'.find_successor(n);
5: }
6:
7: //For a new arrival node n joining vCluster containing node n'
8: n.join(n'){
9: n.joinChord(n')
10: if n's capacity<a predefined threshold then
11:    //n is a regular node
12:    suc_supernode=successor.supernode;
13:    pre_supernode=successor.predecessor.supernode;
14:    //find a closer supernode
15:    if pre_supernode is closer to n than suc_supernode then
16:       supernode=pre_supernode;
17:    else
18:       supernode=suc_supernode;
19:    end if
20:    supernode.addto_clientlist(n);
21: else
22:    //n is a supernode
23:    successor.supernodejoin_forwardnotify(n);
24:    successor.predecessor.supernodejoin_backwardnotify(n);
25: end if
26: }
27:
28: //n get supernode join notification
29: n.supernodejoin_forwardnotify(n'){
30: if n is a regular node then
31:    if n is closer to n' than supernode then
32:       supernode=n';
33:       supernode.addto_clientlist(n);
34:       successor.supernodejoin_forwardnotify(n');
35:    end if
36: end if
37: }
38:
39: n.supernodejoin_backwardnotify(n'){
40: if n is a regular node then
41:    if n is closer to n' than supernode then
42:       supernode= n';
43:       supernode.addto_clientlist(n);
44:       predecessor. supernodejoin_backwardnotify(n');
45:    end if
46: end if
47: }.
```

**Algorithm 5.** Pseudocode for node leaving $v$Cluster.

```
n.leave(){
1: //For a node n leaving vCluster
2: n.leave(){
3: if n is a supernode then
4:    //transfer regular nodes to their closest supernode
5:    suc_s=find_supernode_forward();
6:    pre_s=find_supernode_backward();
7:    for i = 0 up to clientlist.size do
8:       client=clientlist[i];
9:       if client is closer to suc_s than pre_s then
10:         client.supernode_change_notify(suc_s);
11:      else
12:         client.supernode_change_notify(pre_s);
13:      end if
14:   end for
15: end if
16: }
17:
18: //n is notified of supernode change
19: n.supernode_change_notify(s){
20: supernode=s;
21: supernode.addto_clientlist(n);
22: }
23:
24: n.find_supernode_forward(){
25: if n is a supernode then
26:    return n;
27: end if
28: successor.find_supernode_forward();
29: }
30:
31: n.find_supernode_backward(){
32: if n is a supernode then
33:    return n;
34: end if
35: predecessor.find_supernode_backward();
36: }.
```

Table 1
Auxiliary network for load balancing in each type of DHTs

| DHT type | Supernode network | Flat network |
|---|---|---|
| Geographic layout proximity-aware | $p$Cluster | $p$Cluster |
| Soft-state proximity-aware | $v$Cluster | $v$Cluster |
| Proximity-oblivious | $v$Cluster | $p$Cluster/$v$Cluster |

In a soft-state proximity-aware network, such as the topology-aware overlay proposed in [31], proximity information of nodes is stored on the system itself. Soft-state proximity-aware supernode network is already $v$Cluster since physically close nodes can report their information to a same rendezvous supernode. As to the flat networks in this type, $v$Cluster can be easily built on it by letting nodes report their load information to a rendezvous node, which then forwards the information to its supernode.

In a proximity-oblivious supernode network, all regular nodes are clients of supernodes, which constitute a DHT network, and its topology is constructed without proximity consideration. It is not necessary to construct another supernode DHT with close regular nodes in a cluster. The existing supernode network can be taken advantage of to route load information of close nodes to a rendezvous supernode so that $v$Cluster is preferable in this kind of DHTs. Proximity-oblivious flat DHTs can use either $p$Cluster or $v$Cluster load balancing approach considering the fact that the costs to build $p$Cluster and $v$Cluster auxiliary networks are almost the same on this type DHTs. The choice should depend on each approach's advantages based on actual requirements.

## 4. LAR load balancing

Proximity clustering facilitates the design and implementation of efficient and churn-resilient load balancing algorithms. A general method for load balancing is to gather node load

information in a number of rendezvous nodes, which arrange load movement from heavy nodes to light nodes based on their own load information firstly and then based on the load information combined with that of other rendezvous nodes by probing. To consider either of proximity or churn DHT feature in load balancing will degrade performance in the other feature. To take into account proximity, a node needs to contact its specific physically close nodes. It is not flexible enough to handle churn since physically close nodes are always changing. It is known that simple randomized load balancing scheme is a good method to deal with churn as it does not depend on DHT or auxiliary network maintenance, but it cannot ensure that the contacted nodes are physically close nodes. In [26], Shen and Xu proposed LAR algorithms in a Cycloid network, by taking advantage of Cycloid's inherent hierarchical structure. The basic idea of the paper is to let nodes to contact randomized nodes within a range of proximity and achieve a tradeoff between proximity and dynamism.

In the following, we present an implementation of the algorithm in general DHT networks, with the support of $p$Cluster and $v$Cluster from proximity clustering. Let $L_i$ represent the *actual load* of a real server $i$. It is the sum of the load of the items it stores: $L_i = \sum_{k=1}^{m_i} L_{i,k}$, assuming the node has $m_i$ items. Let $C_i$ be the capacity of node $i$; it is defined as a preset target load which the node is willing to hold. We refer to the node whose actual load is no larger than its target load (i.e. $L_i \leqslant C_i$) as a *light node*; otherwise a heavy node. We define *node utilization* as the fraction of its target capacity that is used: $L_i/C_i$. A *system utilization* is the ratio of the total actual load to the total node capacity.

Each node contains a list of data items, labelled as $ID_k$, $k = 1, 2, \ldots$ . To make full use of node capacity and to reduce the load balancing overhead, the items chosen to transfer should be with minimum load. We define these items as *excess items* of a heavy node. Each supernode has a pair of donating sorted list (DSL) and starving sorted list (SSL) which store the load information of all nodes in its cluster. The DSL is for light nodes and the SSL is for heavy nodes. The free capacity of light node $i$ is defined as $\delta L_i = C_i - L_i$. Load information of heavy node $i$ includes the information of its excess items in a set of 3-tuple representation: $\langle L_{i,1}, D_{i,1}, A_i\rangle, \langle L_{i,k}, D_{i,k}, A_i\rangle, \ldots, \langle L_{i,m'}, D_{i,m'}, A_i\rangle$, in which $A_i$ denotes the IP address of node $i$. Load information of light node $j$ is represented in the form of $\langle \delta L_j, A_j\rangle$. An SSL is sorted in a descending order of $L_{i,k}$, and a DSL is sorted in an ascending order of $\delta L_j$. Load rearrangement is executed between a pair of DSL and SSL, as shown in Algorithm 6. This scheme guarantees that heavier items have a higher priority to be reassigned to a light node, which means faster convergence to a system-wide load balance state. A heavy item $L_{i,k}$ is assigned to the most-fit light node with $\delta L_j$ which has minimum free capacity left after the heavy item $L_{i,k}$ is transferred to it. It makes full use of the available capacity.

LAR algorithms run in three phases. First, regular nodes report their load information to their supernodes. Recall that, with the help of the auxiliary network, the load information of physically close nodes gathers together in a supernode or close

**Algorithm 6.** Supernode performs load rearrangement periodically between a pair of DSL and SSL.

```
1: for each item k in SSL do
2:   for each item j in DSL do
3:     if L_{i,k} <= δL_j then
4:       item k is arranged to be transferred from i to j;
5:       if δL_j − L_{i,k} > 0 then
6:         put < (δL_i − L_{i,k}), ip_addr(i) > back to DSL;
7:       end if
8:     end if
9:   end for
10: end for.
```

supernodes. Second, the supernodes arrange load movement. A supernode with nonempty starving list firstly arranges load movement between its own DSL and SSL, which is called local load balancing. The supernode then probes another supernode and arranges load movement between their SSL and DSL until its SSL becomes empty, which is called global load balancing. This scheme can be extended to perform load rearrangement between one SSL and multiple DSLs for improvement.

In DHTs, each node has a routing table and neighbor list, such as successor list in Chord, and leaf sets in Pastry, Tapestry and Cycloid, for item query routing. Supernode $s$ in supernode $n$'s routing table is generally physically closer to $n$ in $p$Cluster, and logically closer to $n$ in $v$Cluster than a randomly chosen supernode in the entire network. Based on this principle, in global load balancing, in order to move load between relative closer nodes, randomized locality-aware probing is used. In LAR probing, each supernode contacts its supernode neighbors or supernodes of its neighbors. After all neighbors are probed, if the supernode's SSL is still nonempty, the supernode randomly contacts other supernodes in the entire ID space.

The third phase of load balancing process is load movement. The load balancing algorithm is based on item movement instead of "virtual servers" to save cost. When an item $D$ is transferred from heavy node $i$ to light node $j$, node $i$ will have a forward pointer in $D$ location pointing to the item $D$ in $j$'s place; item $D$ will have an backward pointer to node $i$ indicating its original host. When queries for item $D$ reach node $i$, they will be redirected to node $j$ with the help of forward pointer. As to each transferred item, two pointers should be maintained.

Cai et al. [10] indicated two aspects of the load balancing problem: evenly and skewed load distribution among nodes. Since LAR maps physically close heavy nodes and light nodes first and then maps nodes in the entire ID space by randomized probing, it can deal with both evenly and skewed load distributions in DHT networks.

In a randomized probing policy, each supernode probes other supernodes randomly for load rearrangement. A simple form is one-way probing, in which a supernode, say node $i$, probes other supernodes one by one to execute load rearrangement between $SSL_i$ and $DSL_j$, where $j$ is a probed node. The randomized probing in our load balancing framework is similar to load balancing problem in other contexts: *competitive online load balancing* and *supermarket model*. Competitive online load balancing is to assign each task to a server online with the objective of minimizing the maximum load on any server, given a set

of servers and a sequence of task arrivals and departures. Azar et al. [5] proved that, in competitive online load balancing, allowing each task to have 2 server choices to choose a less loaded server instead of just 1 choice can exponentially minimize the maximum server load and result in a more balanced load distribution. Supermarket model is to allocate each randomly incoming task modelled as a customer with service requirements to a processor (or server) with the objective of reducing the time each customer spends in the system. Mitzenmacher et al. [19] proved that allowing a task 2 server choices and to be served at the server with less workload instead of just 1 choice leads to exponential improvements in the expected execution time of each task. But a poll size larger than two gains much less substantial extra improvement. The randomized probing between the lists of SSLs and DSLs is similar to the above competitive load balancing and supermarket models if we regard SSLs as tasks and DSLs as servers. However, the randomized probing approach in P2P systems has a general workload and server models. Specifically, in P2P systems, servers are dynamically composed with new ones joining and existent ones leaving, and they are heterogeneous with respect to their capacities. Tasks are of different sizes and arrive in different rates. In [11], we proved that the random probing is equivalent to a generalized supermarket model and showed the following results.

**Theorem 4.1.** *Assume servers join in a Poisson distribution. For any fixed time interval* $[0, T]$, *the length of the longest queue in the supermarket model with* $d = 1$ *is* $\ln n / \ln \ln n\ (1 + O(1))$ *with high probability*; *the length of the longest queue in the model with* $d \geqslant 2$ *is* $\ln n / \ln d + O(1)$, *where n is the number of servers.*

The theorem implies that two-way probing could achieve a more balanced load distribution with faster speed even in churn, but $d$-way probing, $d > 2$, may not result in much additional improvement.

## 5. Performance evaluation

We designed and implemented a simulator in Java for evaluation of the LAR algorithm based on $p$Cluster ($p$LAR) and $v$Cluster ($v$LAR) on Chord DHT and compared their performance with CRA [14] and KTree method [36]. CRA can deal with DHT churn by randomized probing in load balancing and KTree is a proximity-aware load balancing method that maps physically close heavy nodes and light nodes for load transfer. We compared the different load balancing schemes in Chord without churn in terms of proximity-aware load balancing achievement, load balancing cost, heterogeneity consideration, and also compared the resilience of the schemes in Chord with churn. In CRA, we set 16 directories as in [14]. We set the load information size threshold for load balancing in each KTree node as 15; that is, when the total size of load information of a node reaches 15, it executes load rearrangement.

We use two transit-stub topologies generated by GT-ITM [32]: "ts5k-large" and "ts5k-small" with approximately 5,000 nodes each. "ts5k-large" has 5 transit domains, three transit nodes per transit domain, 5 stub domains attached to each transit node and 60 nodes in each stub domain on average. "ts5k-small" has 120 transit domains, five transit nodes per transit domain, 4 stub domains attached to each transit node and two nodes in each stub domain on average. "ts5k-large" has a larger backbone and sparser edge network (stub) than "ts5k-small". "ts5k-large" is used to represent a situation in which DHT overlay consists of nodes from several big stub domains, while "ts5k-small" represents a situation in which DHT overlay consists of nodes scattered in the entire Internet and only few nodes from the same edge network join the overlay. 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 assumed bounded Pareto distribution for the load of nodes and items. This distribution reflects real world where there are machines with capacities that vary by different orders of magnitude. Table 2 lists the parameters of the simulation and their default values.

Table 2
Simulated environment and algorithm parameters

| Environment parameter | Default value |
|---|---|
| System utilization | 0.5–1 |
| Object arrival location | Uniform over ID space |
| Number of nodes | 4096 |
| Node capacity | Bounded Pareto: shape:2 |
| | Lower bound: 25,000, upper bound: 25,000*10 |
| Supernode threshold | 50,000 |
| Number of items | 20,480 |
| Existing item load | Bounded Pareto: shape:2 |
| | Lower bound: mean item actual load/2 |
| | Upper bound: mean item actual load/2*10 |

### 5.1. Effectiveness of $p$LAR and $v$LAR algorithms

In this section, we will show the effectiveness of LAR load balancing algorithms on $p$Cluster and $v$Cluster. First, we present the impact of $p$LAR and $v$LAR on the alignment of the skews in load distribution and node capacity when the system is fully loaded. Because the results of $p$LAR and $v$LAR are almost the same, we use results of $p$LAR to represent both. From Fig. 2(a) and (b), we can see that many nodes are overloaded before load balancing and after load balancing they become light by transferring excess items to light nodes. Fig. 2(c) shows the scatterplot of loads according to node capacity. These figures show that the load balancing frame assigns load to nodes based on their capacity with the consideration of node heterogeneity.

Load movement factor is the total load transferred due to load balancing divided by the system actual load. We measured the load movement factors due to different load balancing schemes, $p$LAR, $v$LAR, KTree and CRA, on systems of utilization from 0.5 to 1 at a step size of 0.05. Fig. 3 plots the load movement factors. We can see that the schemes require the same amount of load movement in total for load balance. This is consistent with

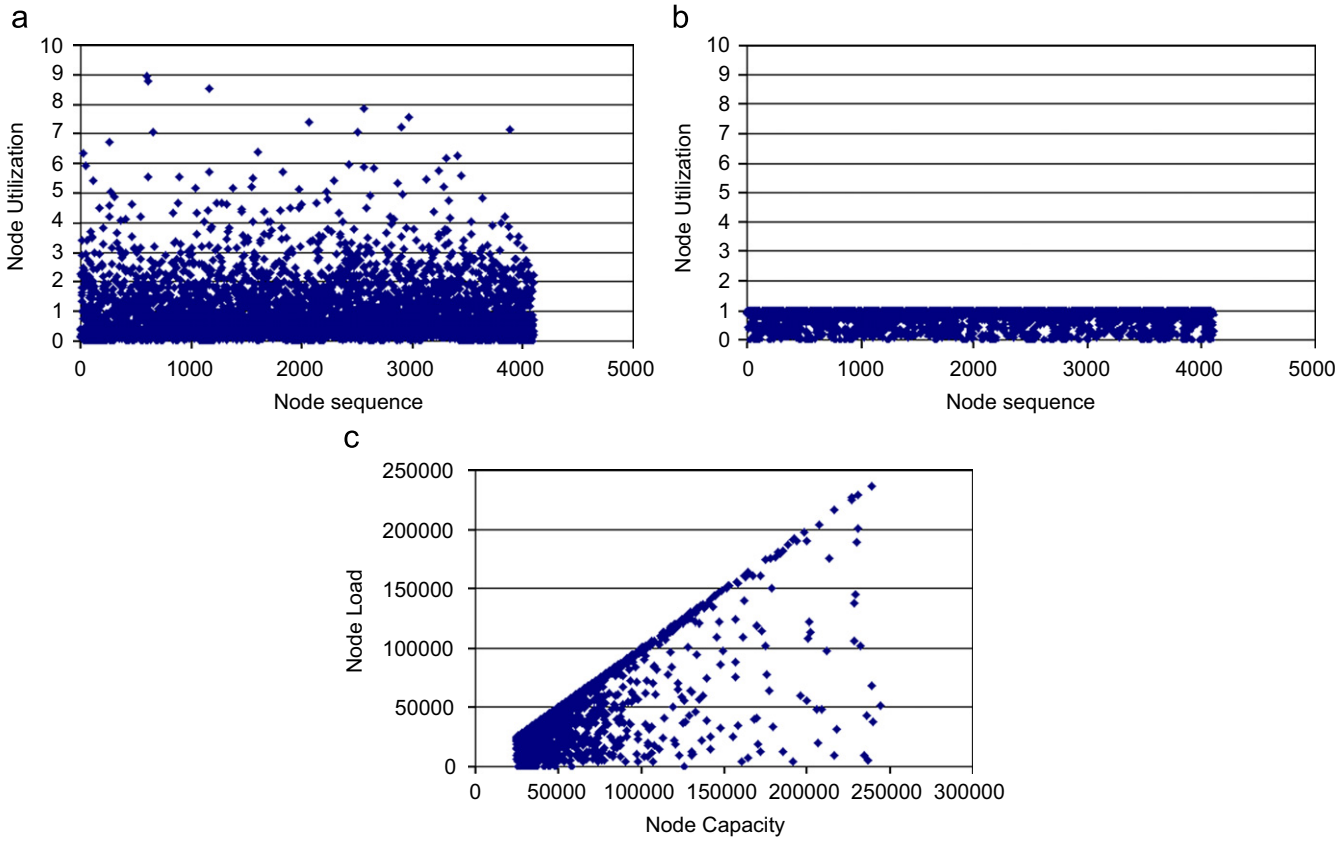



Fig. 2. Effect of *p*LAR and *v*LAR load balancing. (a) Before load balancing, (b) after load balancing and (c) utilization of nodes after load balancing.

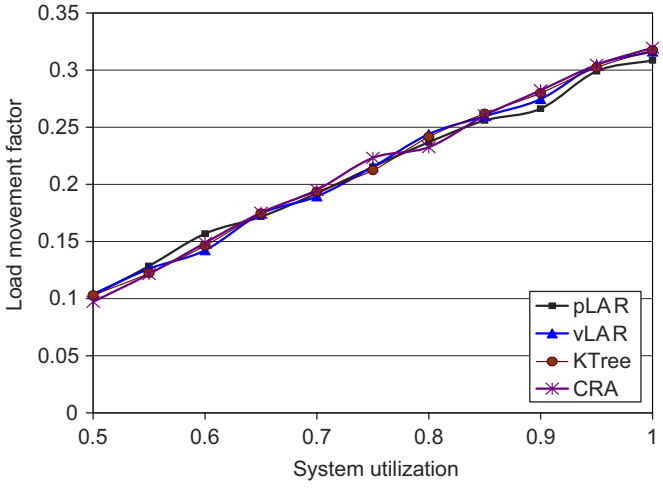



Fig. 3. Load movement factor in different load balancing schemes.

the observations by Rao et al. [21] that the load moved depends only on distribution of loads, and the target to be achieved, but not on load balancing schemes. This result suggests that better load balancing schemes should explore how to move the same amount of load along shorter distance to reduce item transfer cost; in other words, how to achieve locality-aware load balancing. In the following, we will examine the performance of various load balancing schemes in terms of other performance metrics.

### 5.2. Proximity-aware load balancing

In this section, we will show how *p*Cluster and *v*Cluster help LAR to achieve high proximity-aware performance. Fig. 4(a) and (b) shows the cumulative distribution function (CDF) of the percentage of total moved load versus moving distance in each load balancing scheme when system utilization approaches 1 in "ts5k-large" and "ts5k-small", respectively. This performance metric represents the load movement cost for load balance. The more the load moved along the shorter distances, the less the load balancing costs. We can see that, in "ts5k-large", *p*LAR, *v*LAR and KTree are able to transfer 95% of total moved load within 10 hops, while CRA moves only about 15% within 10 hops. Almost all load movements in *p*LAR, *v*LAR and KTree are within 15 hops, while CRA scheme moves only 75% within 15 hops. The results show that *p*LAR, *v*LAR and KTree move most load in short distances while CRA moves most load in long distances. From Fig. 4(b), we can have the same observations as in "ts5k-large", although the performance difference between the schemes is not so significant as in "ts5k-large". The more the load moved in the shorter distance, the higher the proximity-aware performance of a load balancing scheme with less load balancing cost. The results indicate that proximity-aware load balancing schemes *p*LAR, *v*LAR and KTree perform better than CRA with regard to proximity-aware performance. The results of *p*LAR and *v*LAR are comparable to KTree means that *p*LAR and *v*LAR are as efficient as KTree to guide heavy nodes to transfer load to physically close light nodes either

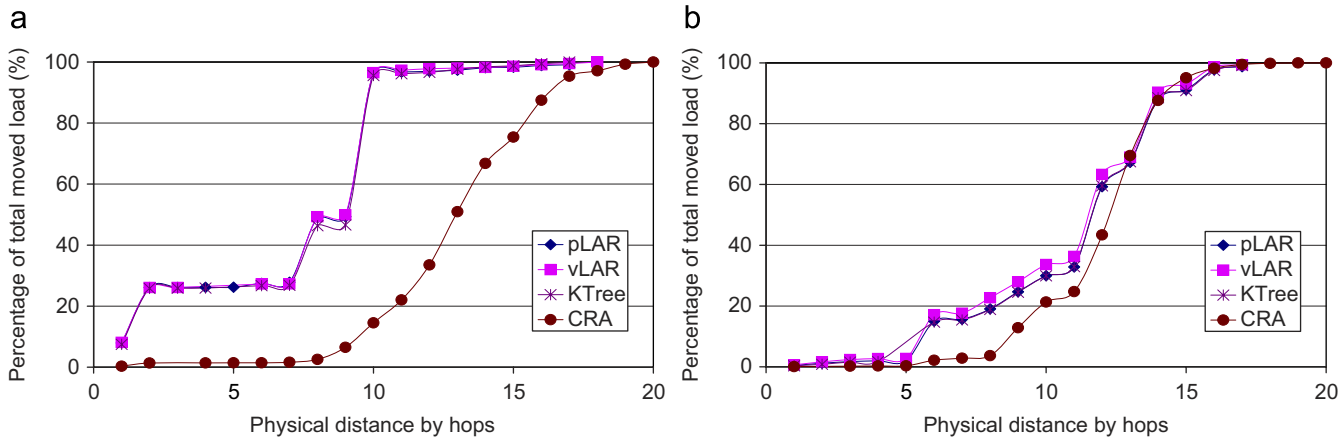



Fig. 4. CDF of total moved load distribution of different load balancing schemes. (a) ts5k-large and (b) ts5k-small.

when nodes are from several big subdomains or when nodes are scattered in the entire Internet.

*5.2.1. Breakdown of load movement cost*

In general, a load balancing process needs to gather node load information in a number of rendezvous nodes, which arrange load movement. Fig. 5 shows the breakdown of total moved load in percentage of the moved load in local or in global load balancing phase. We find that most load is moved in the local phase in *p*LAR, *v*LAR and CRA, while KTree moves most load in global phase. The LAR algorithm takes proximity into account in global load balancing phase. The hash-based proximity clustering facilitates it to achieve better performance in both local and global load balancing phases. KTree constructs an auxiliary *k*-ary tree structure, so that the load information of physically close nodes can be forwarded upward along the tree and be gathered for load balancing in global phase.

The figures show that CRA moves more load in local balancing phase than *p*LAR and *v*LAR. It has 16 rendezvous nodes, and our simulation results show that *p*LAR has 60 and *v*LAR has 90 rendezvous nodes. Less rendezvous nodes mean more load information gathered in a node, and more excess load can be solved in local load rearrangement. However, it comes with the cost of proximity-aware performance degradation because excess items may be assigned to a remote node caused by coarse grain load information. Though rendezvous node number has only a small effect on load balance achievement as claimed in [14], this number has a significant impact on proximity-aware load balancing.

*5.2.2. Communication cost*

In addition to load movement cost, communication cost constitutes a main part of load balancing overhead. The cost is directly related with message size and physical path length of the message travelled; we use the product of these two factors of all exchanging messages to represent the cost. It is assumed that the size of a message asking and replying for load information is 1 unit. To let the results be comparable between different schemes, we did not count the communication cost for heavy node or light node determination in KTree. Fig. 6(a) and (b) plots the communication cost of *p*LAR, *v*LAR, KTree and CRA in "ts5k-large" and "ts5k-small", respectively. From these figures, we can see that the communication cost increases with the system load, and that of KTree is much more higher than the others. We also find that *p*LAR incurs much less communication cost than *v*LAR and CRA. Note that the load information communication cost is due to information reporting (to rendezvous nodes) and node probing in global load balancing phase (or information propagation in KTree). Fig. 7 gives breakdown of the cost when the system is heavily loaded. The figure shows that the reporting costs of vLAR, KTree and CRA are almost the same. The high communication cost of KTree is caused by load information indirect propagation in the *k*-ary tree. KTree constructs a tree-structured auxiliary network to help gather load information of physically close nodes. Though KTree facilitates a node to communicate its physically close nodes for load balancing, it requires nodes to report their load information upward step by step to the tree root. This is the main cause for the high total communication cost. In contrast, randomized probing in *p*LAR, *v*LAR and CRA directly reduces the cost.

Recall that *p*LAR enables nodes to report their load information to their physically closest supernode directly, while the information has to be routed based on routing algorithm on the original DHT to reach its destination supernode in other schemes. Therefore, *p*LAR costs less in the reporting phase. The figure also shows that *p*LAR needs less probing cost than *v*LAR and CRA costs the least in probing phase. It is because a supernode probes its peers in top-level supernode DHT with short path length in *p*LAR; but, in other schemes, the probing process is run in the original DHT and their probing message passes though regular nodes to reach other supernodes. Due to the fact that almost all excess load is solved in local load balancing as shown in Fig. 5, CRA has less probing cost in global load balancing.

In summary, *p*LAR and *v*LAR achieve the goal of load balancing as KTree at much less communication cost. *p*LAR incurs less communication overhead than *v*LAR and CAR, but the benefit comes with the cost of supernode DHT maintenance.

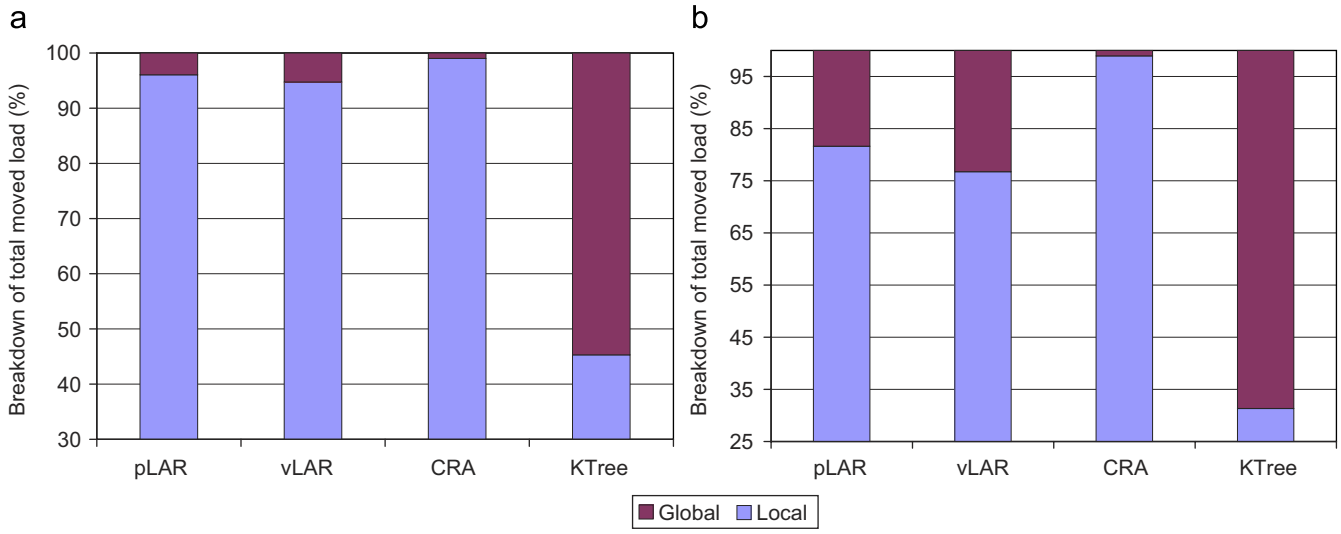



Fig. 5. Breakdown of total moved load of different load balancing schemes. (a) ts5k-large and (b) ts5k-small.

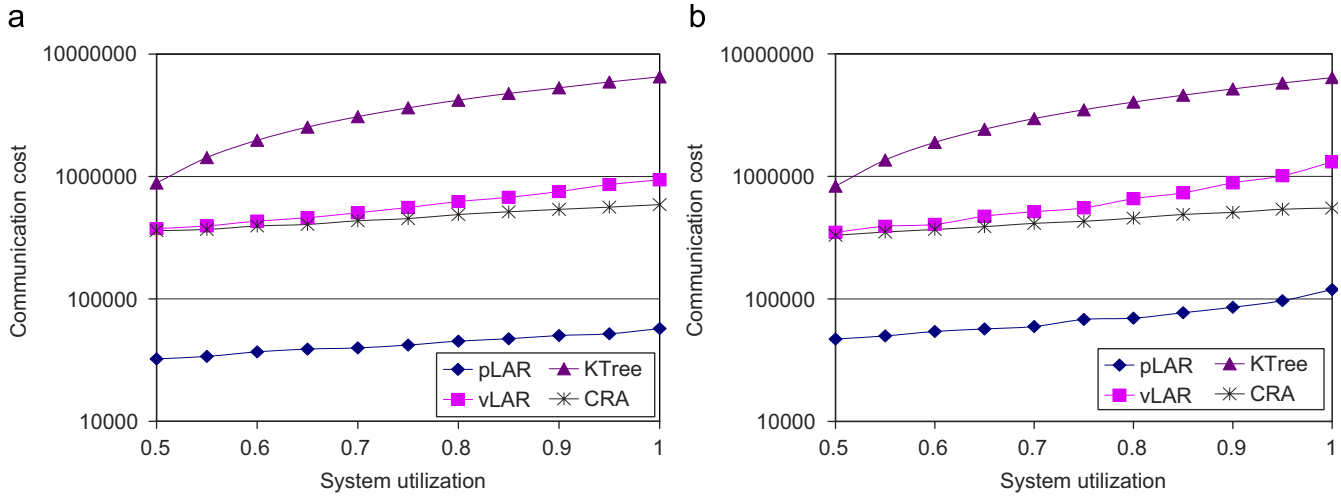



Fig. 6. Communication cost of different load balancing schemes. (a) ts5k-large and (b) ts5k-small.

### 5.3. Churn-resilient load balancing

In P2P networks with churn, a great number of nodes and items join, leave and fail continuously and rapidly. This gives load balancing schemes a challenge because it is hard to achieve the objective of load balance under churn. For example, a node becomes overloaded if it cannot provide sufficient capacity for the load transferred by its leaving neighbors; fast and continuous item joins in a specific node make the node overloaded; when rendezvous nodes for load rearrangement suddenly leave or fail, some nodes may not be able to shed their load in time. In addition to using randomized probing to handle churn like CRA, $p$Cluster and $v$Cluster have maintenance algorithms to deal with churn. They let regular nodes periodically probe their supernode to handle supernode failures and let supernodes help their regular nodes to find another supernode before leaving. As we mentioned that KTree is not flexible enough to handle churn because if a parent fails or leaves without timely fix, the load imbalance of its children cannot be solved. What is more, the tree needs to be reconstructed every time after load transfer, degrading load balancing efficiency.

We evaluated the efficiency of the $p$LAR and $v$LAR in dynamic situations with respect to a number of performance factors. Simulation results verified the superiority of $p$LAR and $v$LAR in DHTs with churn, in comparison with CRA and KTree. In this experiment, we run each trial of the simulation for $20T$ simulated seconds, where $T$ is a parameterized load balancing period, and it was set to 60 s in our test. The item join/departure rate was modelled by a Poisson process with a rate of 0.4; that is, there were one item join and one item departure every 2.5 s. We ranged node interarrival time from 10 to 90 s, with 10 s increment in each step. A node lifetime is computed from arrival rate and number of nodes in the system. For example, when node interarrival time is 10 s, if we fix the steady-state number of nodes in the system to 4096, then the

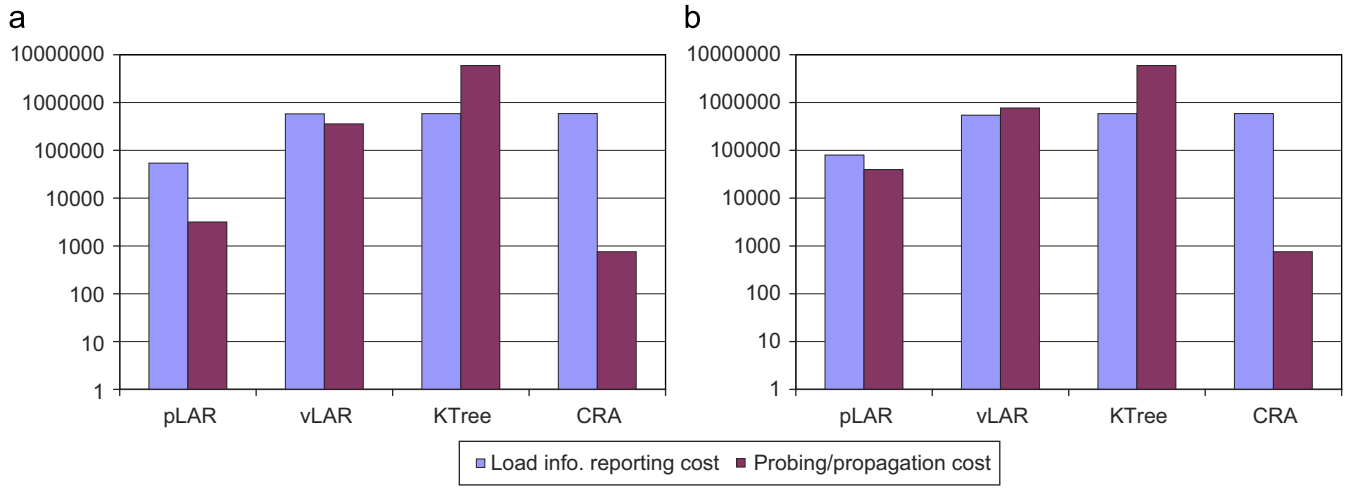



Fig. 7. Breakdown of total communication cost of different load balancing schemes. (a) ts5k-large and (b) ts5k-small.

node lifetime is about $4096 \times 10$ s. The system utilization was set to 0.8. We adopted the same metrics as in [14]:

(1) *Maximum load movement factor*: We measure the load movement factor after each load balancing period $T$ in simulation and take the maximum of those results over a $20T$ period as the maximum load movement factor.
(2) *Maximum and average 99.9th percentile node utilizations*: We measure the maximum 99.9th percentile of the utilizations of the nodes after each load balancing period $T$ in simulation and take the maximum and average of those results over a $20T$ period as the maximum and average 99.9th percentile node utilizations.
(3) *Load movement rate*, defined as the total load moved incurred due to load balancing divided by the total load of items moved due to node joins and departures in the system.

Figs. 8 and 9 plot the performance due to $p$LAR, $v$LAR, KTree and CRA versus node interarrival time. Fig. 8(a) and (b) shows that the average 99.9th percentile node utilizations of $p$LAR, $v$LAR and CRA are around 1.1, the maximum 99.9th percentile node utilizations are slightly higher than the average and kept no more than 1.2, but both of them are between 1.6 and 2 in KTree. The observation that $p$LAR and $v$LAR keep the node utilizations close to 1 implies that on average they can achieve the load balancing goal of keeping each node's load below its capacity even in churn. The result that $p$LAR and $v$LAR are comparable to CRA implies that they are as efficient as CRA to deal with churn. In contrast, higher node utilization of KTree means that it is not resilient enough to cope with churn. $p$LAR, $v$LAR and KTree are all auxiliary networks for load balancing. However, because KTree is not reliable in churn, it cannot achieve load balancing goal to keep nodes lightly loaded in churn. In contrast, $p$LAR and $v$LAR generated by hash-based proximity clustering are resilient to churn with its self-organization.

Fig. 9(a) shows that the maximum load movement factors of $p$LAR, $v$LAR, CRA and KTree are kept between 20% and 25%, which means that all schemes move almost the same system load to achieve load balance. This result suggests that a better load balancing scheme should explore how to move the same amount of load in time under churn; that is, let nodes shed their excess load timely in order to keep their load below their capacities. Fig. 9(b) illustrates the load movement rate. We can observe that the load moved due to load balancing is very small compared with the load moved due to node joins and departures, and it is up to 35% for LAR and 45% for CRA and KTree. When the node interarrival time is 10, the rate is the highest in $p$LAR and $v$LAR. It is because faster node joins and departures generate much higher load imbalance, therefore more load transfer is needed to achieve load balance. The observation that the results of $p$LAR and $v$LAR are comparable to, or even better than, CRA implies that $p$LAR and $v$LAR schemes are as efficient as CRA to handle churn by moving a small amount load. In summary, in the face of rapid arrivals and departures of items of widely varying load and nodes of widely varying capacity, $p$LAR and $v$LAR achieve load balance fast while moving almost the same amount of load as other schemes, up to 23% of the load that arrives into the system. However, KTree cannot handle churn as effectively as the $p$LAR, $v$LAR and CRA.

To evaluate the capability of $p$LAR and $v$LAR to deal with node failures, we tested their performance in node failures. Because the results of $p$LAR are almost the same as $v$LAR, we use one group of results to represent both. Fig. 10 plots the relationship between 99.9th percentile node utilization and load movement factor versus interval time of node failures with and without item joins and departures. Fig. 10(a) shows that, without item joins and departures, $p$LAR and $v$LAR keep every node light loaded under node failures by transferring about 30% total system load. Fig. 10(b) shows that, with item joins and departures, the 99.9th percentile node utilization remains at about 1.1 by transferring almost the same total system load. It

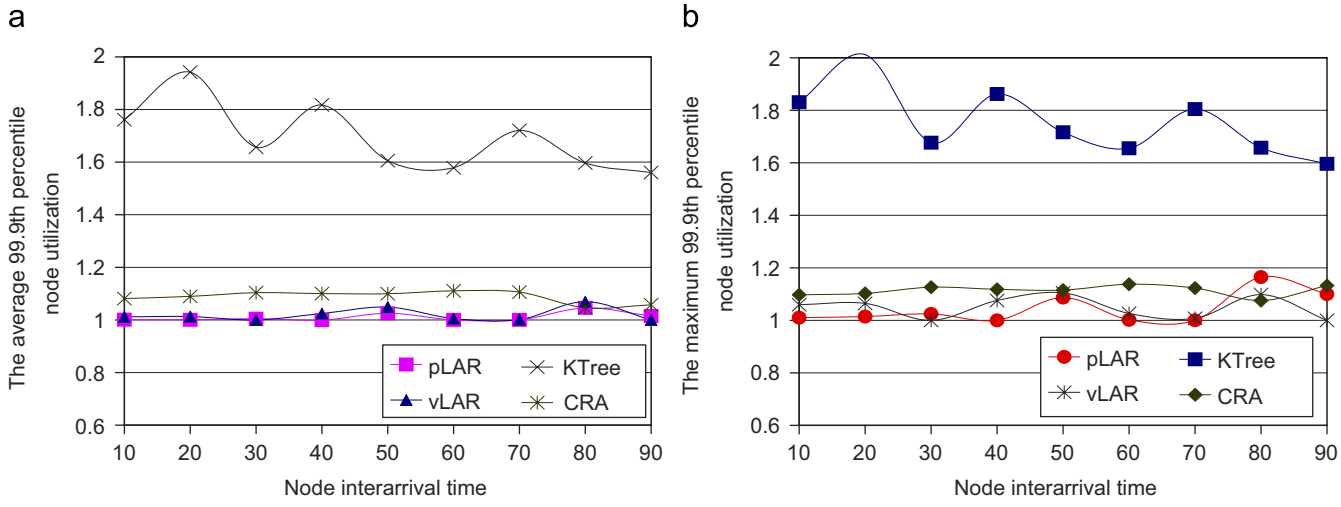



Fig. 8. Node utilizations of different load balancing schemes in DHT networks with churn. (a) Average node utilization and (b) maximum node utilization.

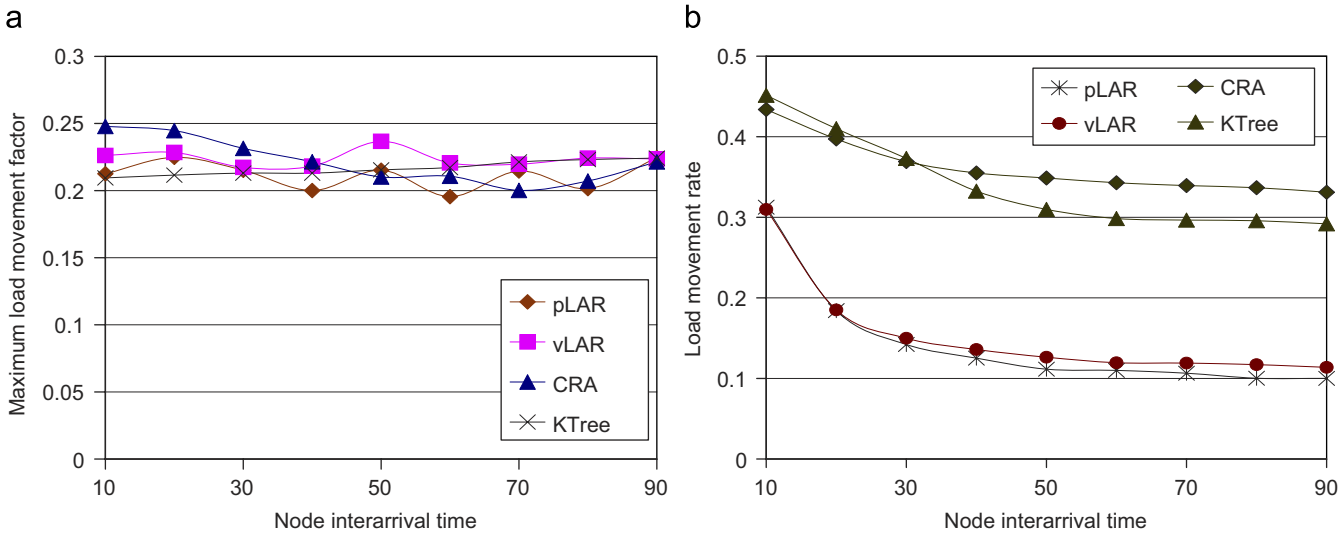



Fig. 9. Load movement overhead of different load balancing schemes in DHT networks with churn. (a) Maximum load movement factor and (b) load movement rate.

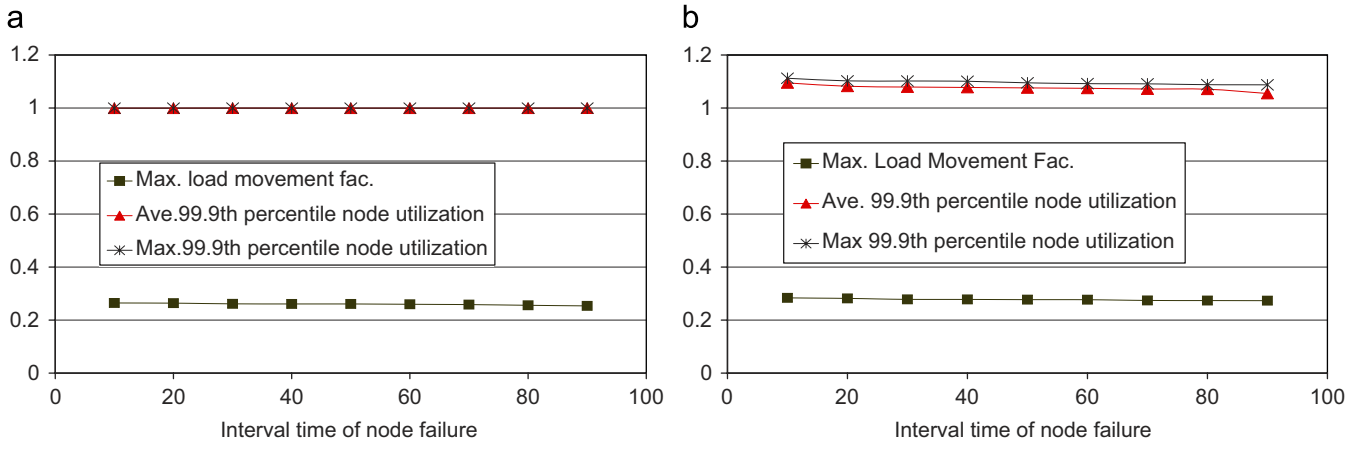



Fig. 10. Effect of *p*LAR and *v*LAR load balancing in DHT networks with node failures. (a) Without item join and departure and (b) with item join and departure.

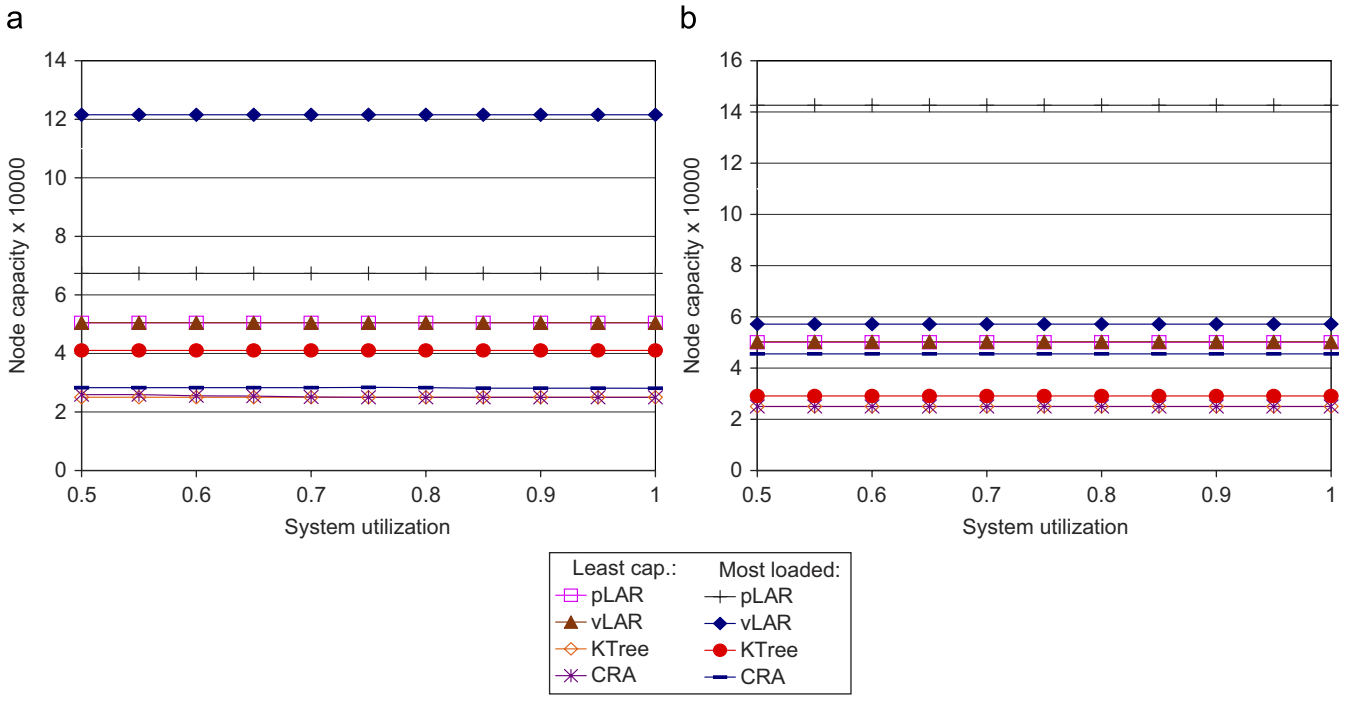



Fig. 11. Capacity of the rendezvous node with the least capacity/most load balancing overhead in different load balancing schemes. (a) ts5k-large and (b) ts5k-small.

implies that a number of nodes are slightly overloaded. This is because sometimes *p*LAR and *v*LAR cannot deal with excess load caused by extra joining items instantly. In conclusion, *p*LAR and *v*LAR load balancing are robust in that they still can achieve load balance under node failures.

### 5.4. Heterogeneity consideration in load balancing

Generally, in load movement load balancing, load information of nodes is gathered in a number of rendezvous nodes, and the rendezvous nodes do load rearrangement from heavy nodes to light nodes. Consequently, the rendezvous nodes afford more load balancing overhead due to load information maintenance and load rearrangement. A rendezvous node may be already heavily loaded and does not have sufficient capacity for load balancing overhead, making the load imbalance problem even more severe, so that an effective load balancing algorithm should not only distribute application load but also load balancing overhead, based on node heterogeneous capacities. To evaluate the performance of load balancing schemes with regard to heterogeneity consideration, we recorded the capacity of least capacity rendezvous node among all rendezvous nodes, and the capacity of the rendezvous node affording most load balancing overhead measured in terms of the number of nodes reporting their load information to the rendezvous node, in each load balancing scheme. Higher capacity of the least capacity rendezvous node means that higher capacity nodes, rather than low-capacity nodes, are assigned as rendezvous nodes to maintain load information and arrange load movement. Higher capacity of the most loaded rendezvous node implies that higher capacity nodes are responsible for more load balancing overhead. They provide a guarantee of effective load balancing operation, and prevent heavy nodes from more load assignment caused by load balancing itself. Fig. 11(a) and (b) shows the experiment results in "ts5k-large" and "ts5k-small", respectively. We can see from the figures that the capacities of the least capacity and most loaded rendezvous nodes in *p*LAR and *v*LAR are higher than that of KTree and CRA. The results confirm that, with the help of *p*Cluster and *v*Cluster, *p*LAR and *v*LAR consider heterogeneity in load balancing by distinguishing regular nodes from supernodes, and let supernodes afford more load balancing overhead.

## 6. Conclusions

Unlike existing supernode clustering approaches which designate a static gateway of regular nodes as their supernode, this paper presents a hash-based proximity clustering approach to construct a self-organized churn-resilient auxiliary supernode network for load balancing in heterogeneous DHT networks. The auxiliary network can be physical or virtual. In the physical network *p*Cluster, regular nodes connect to their physically close supernodes and periodically report their load information to their supernodes. In the virtual network *v*Cluster, regular nodes connect to their logically close supernodes as in the original proximity-oblivious DHT network; physically close nodes put their load information together by routing their load information to a rendezvous supernode or close supernodes. The auxiliary network facilitates the design and implementation of LAR load balancing algorithm. Simulation results show the superiority of the approach, in comparison with a number of other randomized and proximity-aware load balancing algorithms. Benefits of proximity clustering come at the cost of cluster maintenance. Although *p*Cluster and *v*Cluster are

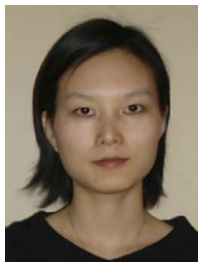
self-organized, there is still need for minimum maintenance as in DHT networks.

## Acknowledgments

The authors are grateful to the anonymous reviewers for their valuable comments and suggestions. This research was supported in part by US Acxiom Corporation, US NSF Grant CCF-0611750, DMS-0624849, CNS-0702488, CRI-0708232 and NASA Grant 03-OBPR-01-0049. An early version of this work [25] was presented in the Proceedings of IPDPS'06.

## References

[1] M. Adler, E. Halperin, R.M. Karp, V. Vazirani, A stochastic process on the hypercube with applications to peer-peer networks, in: Proceedings of STOC, 2003.
[2] I. Ahmad, A. Ghafoor, Semi-distributed load balancing for massively parallel multicomputer systems, IEEE Trans. Software Eng. 17 (10) (1991).
[3] K. Albrecht, R. Arnold, M. Gähwiler, R. Wattenhofer, Aggregating information in peer-to-peer systems for improved join and leave, in: Proceedings of the 4th International Conference on P2P Computing, 2004.
[4] T. Asano, D. Ranjan, T. Roos, E. Welzl, P. Widmaier, Space filling curves and their use in geometric data structure, Theoret. Comput. Sci. 181 (1) (1997) 3–15.
[5] Y. Azar, A. Broder, et al., Balanced allocations, in: Proceedings of STOC, 1994, pp. 593–602.
[6] A.R. Bharambe, M. Agrawal, S. Seshan, Mercury: supporting scalable multi-attribute range queries, in: Proceedings of ACM SIGCOMM, 2004.
[7] M. Bienkowski, M. Korzeniowski, F.M. auf der Heide, Dynamic load balancing in distributed hash tables, in: Proceedings of IPTPS, 2005.
[8] J. Byers, J. Considine, M. Mitzenmacher, Geometric generalizations of the power of two choices, in: Proceedings of ACM SPAA, 2004.
[9] M. Castro, P. Druschel, Y.C. Hu, A. Rowstron, Topology-aware routing in structured peer-to-peer overlay networks, in: Future Directions in Distributed Computing, 2002.
[10] A. Chervenak, M. Cai, M. Frank, A peer-to-peer replica location service based on a distributed hash table, in: ACM/IEEE Conference on Supercomputing (SC), 2004.
[11] S. Fu, H. Shen, C. Xu, Power of two for randomized selections in generalized supermarket models, Technical Report, ECE Department, Wayne State University, 2006.
[12] P. Ganesan, M. Bawa, H. Garcia-Molina, Online balancing of range-partitioned data with applications to peer to peer systems, in: Proceedings of the 30th VLDB Conference, 2004.
[13] L. Garces-Erice, E.W. Biersack, K.W. Ross, P.A. Felber, G. Urvoy-Keller, Hierarchical p2p systems, in: Proceedings of ACM/IFIP International Conference on Parallel and Distributed Computing (Europar), 2003.
[14] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, I. Stoica, Load balancing in dynamic structured P2P systems, Performance Evaluation 63 (3) (2006).
[15] B. Godfrey, I. Stoica, Heterogeneity and load balance in distributed hash tables, in: Proceedings of INFOCOM, 2005.
[16] D. Karger, E. Lehman, T. Leighton, M. Levine, et al., Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web, in: Proceedings of STOC, 1997, pp. 654–663.
[17] D.R. Karger, M. Ruhl, Simple efficient load balancing algorithms for peer-to-peer systems, in: Proceedings of IPTPS, 2004.
[18] G. Manku, Balanced binary trees for ID management and load balance in distributed hash tables, in: Proceedings of PODC, 2004.
[19] M. Mitzenmacher, On the analysis of randomized load balancing schemes, in: Proceedings of SPAA, 1997.
[20] M. Nar, U. Wieder, Novel architectures for p2p applications: the continuous–discrete approach, in: Proceedings of ACM SPAA, 2003.
[21] A. Rao, K. Lakshminarayanan, et al., Load balancing in structured P2P systems, in: Proceedings of IPTPS, 2003.
[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, in: Proceedings of ACM SIGCOMM, 2001, pp. 329–350.
[23] S. Ratnasamy, M. Handley, R. Karp, S. Shenker, Topologically-aware overlay construction and server selection, in: Proceedings of INFOCOM, 2002.
[24] A. Rowstron, P. Druschel, Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems, in: Proceedings of Middleware, 2001.
[25] H. Shen, C. Xu, Hash-based proximity clustering for load bancing in heterogeneous DHT networks, in: Proceedings of IPDPS, 2006.
[26] H. Shen, C. Xu, Locality-aware and churn-resilient load balancing algorithms in structured peer-to-peer networks, IEEE Trans. Parallel and Distributed Systems 18 (6) (2007) 849–862.
[27] H. Shen, C. Xu, G. Chen, Cycloid: a scalable constant-degree p2p overlay network, Performance Evaluation 63 (3) (2006) 195–216 (an early version appeared in Proceedings of IPDPS'04).
[28] I. Stoica, R. Morris, et al., Chord: a scalable peer-to-peer lookup protocol for Internet applications, IEEE/ACM Trans. Networking 11 (1) (2003) 17–32.
[29] M. Waldvogel, R. Rinaldi, Efficient topology-aware overlay network, in: Proceedings of HotNets-I, 2002.
[30] Z. Xu, M. Mahalingam, M. Karlsson, Turning heterogeneity into an advantage in overlay routing, in: Proceedings of INFOCOM, 2003.
[31] Z. Xu, C. Tang, Z. Zhang, Building topology-aware overlays using global soft-state, in: Proceedings of ICDCS, 2003.
[32] E. Zegura, K. Calvert, et al., How to model an Internetwork, in: Proceedings of INFOCOM, 1996.
[33] C. Zhang, A. Krishnamurthy, R.Y. Wang, Brushwood: distributed trees in peer-to-peer systems, in: Proceedings of IPTPS, 2005.
[34] B. Zhao, Y. Duan, L. Huang, A. Joseph, J. Kubiatowicz, Brocade: landmark routing on overlay networks, in: Proceedings of IPTPS, 2002.
[35] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, et al., Tapestry: an infrastructure for fault-tolerant wide-area location and routing, IEEE J. Select. Areas in Commun. 12 (1) (2004) 41–53.
[36] Y. Zhu, Y. Hu, Efficient proximity-aware load balancing for dht-based p2p systems, IEEE Trans. Parallel and Distributed Systems 16 (4) (2005) (an early version appeared in Proceedings of IPDPS'04).



**Haiying Shen** received the B.S. degree in Computer Science and Engineering from Tongji University, China, in 2000, and the M.S. and Ph.D. degrees in Computer Engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Assistant Professor in the Department of Computer Science and Computer Engineering of University of Arkansas. Her research interests include distributed and parallel computer systems and computer networks, with an emphasis on peer-to-peer and content delivery networks, mobile computing, high-performance cluster and grid computing. She is a member of IEEE and ACM.

**Cheng-Zhong Xu** received the B.S. and M.S. degrees from Nanjing University in 1986 and 1989, respectively, and the Ph.D. degree from the University of Hong Kong in 1993, all in Computer Science. He is a Professor in the Department of Electrical and Computer Engineer of Wayne State University and the Director of the Center of Networked Computing Systems. His research interest includes distributed and parallel systems, in particular, reliable and secure Internet services and architecture, energy-efficient mobile and embedded systems, and resource management in cluster and grid computing. He has published more than 120 peer-reviewed articles in journals and conferences in these areas. He is the author of "*Scalable and Secure Internet*

*Services and Architecture*" (Chapman & Hall/CRC Press, 2005) and the leading co-author of "*Load Balancing in Parallel Computers: Theory and Practice*" (Kluwer Academic, 1997). He serves on a number of editorial boards, including IEEE Transactions on Parallel and Distributed Systems, J. of Parallel and Distributed Computing, J. of Parallel, Emergent, and Distributed Systems, J. of Computers and Applications, and International J. of High Performance Computing and Networking. He was a founding co-Chair of International Workshop on Security in Systems and Networks (SSN), a general co-Chair of 2006 International Conference on Embedded and Ubiquitous Computing (EUC'06), and a PC member of numerous conferences. He was a recipient of the "Faculty Research Award" of Wayne State University in 2000, the 2002 "President's Award for Excellence in Teaching," and the 2003 "Career Development Chair Award." He is a senior member of the IEEE and a member of ACM.