Hash-based Proximity Clustering for Load Balancing in Heterogeneous DHT Networks

Haiying Shen and Cheng-Zhong Xu,

Department of Electrical and Computer Engineering
Wayne State University, Detroit, MI 48202
{shy,czxu}@ece.eng.wayne.edu

Abstract

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 nature of DHT networks and time-varying load characteristics.

In this paper, we present a hash-based proximity clustering approach for load balancing in heterogeneity 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 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 approach, 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 algorithms due to the use of proximity information.

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 [14, 9, 11, 19, 13]. 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. In addition, factors like non-uniform file size, time varying file popularity, and node heterogeneity in capacity make the load balance 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 and proximity. In addition, DHT networks are often highly heterogeneous. 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, 5, 20, 7, 12]. "Virtual nodes" [14, 5, 20] and "item movement" [7] 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.* [8] and Godfrey *et al.* [5] 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 [20]. The algorithm is based on an additional network constructed on top of Chord. Although the network is self-organized, it needs extra cost for reconstruction after every load transfer, and the algorithm is hardly applicable to DHT with churn. In [12], 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.

This paper applies the concept of proximity-aware randomization to general DHT networks. It 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. 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 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 how locality-aware randomized load balancing algorithm implements 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. Node heterogeneity in P2P networks makes the load balancing problem even more severe. To alleviate the problem, Stoica *et al.* [14] proposed an abstraction of "virtual servers," in which each real node runs $\Omega(\log n)$ virtual severs, 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, but it incurs large space overhead and compromises lookup efficiency. Brighten *et al.* [6] 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 [7] 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. Most recently, Bienkowski *et al.* [2] proposed a node leave and re-join strategy to balance the key Id intervals across the nodes.

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 load reassignment. Rao *et al.* [8] 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 by randomized probing. Based on this work, Godfrey *et al.* [5] developed churn resilient algorithm (CRA) for dynamic DHTs with churn. In this work, when a node's fraction of capacity used exceeds a predetermined threshold, its excess virtual nodes will be moved to light nodes immediately without waiting for next periodic balancing.

An alternative to "virtual server" migration is "item movement." Karger and Ruhl [7] proved that the "virtual server" method could not be guaranteed to handle item distributions where an 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 move items between the nodes for load balancing.

Note that the load re-assignment 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 bandwidth requirement for load balancing. One of the early work to utilize the proximity information to guide load balancing is due to Zhu and Hu [20]. They suggested to build a *k*-ary tree structure on top of a DHT overlay, and use proximity information to map physically close heavy and light nodes. Load information will be propagated from tree leaf nodes upwards along the tree. When the total length of information reaches a certain threshold, the tree node would execute load rearrangement. However, the tree construction and maintenance are costly, especially in DHTs with churn. Without timely fixes, the tree will be destroyed, degrading load balancing efficiency. Besides, the tree needs to be reconstructed every time after virtual server transferring.

Shen and Xu proposed locality-aware randomized load balancing algorithms to take advantage of the hierarchical structure of Cycloid to cope with both dynamism and proximity [12]. 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 highly capacity and fast connections and regular nodes are nodes with low capacity and slower connections. 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 [4, 18] take proximity into account by clustering physically close nodes in 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 it generates excessive load on the network. Xu *et al.* [16] 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 leave, 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 in 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. The interconnections between the supercodes and their associated regular nodes can be defined by their routing tables. We distinguish the interconnections in two forms: physical and virtual. A physical cluster, denoted by *p*Cluster, is a structure in which each node is connected to its physically 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 [10, 16]. 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 use the vector of distances $< d_1, d_2, \ldots, d_m >$ as its coordinate in Cartesian space. Two physically close nodes will have similar landmark vectors.

We use space-filling curves [1], such as Hilbert curve as in [16], to map m-dimensional landmark vectors to real-numbers, 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 is 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 physical closeness of 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 nodes and regular nodes as keys in the top-level supernode DHT. The top-level supernode DHT can be any type of DHT such as Chord, Pastry or CAN, 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. 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 nearby clusters. 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.

We use a "proximity-neighbor selection" technique as described in [3, 15] 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.

Algorithm 1 Pseudo-code for node n joining in *p*Cluster containing node n'.

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

Algorithm 2 Pseudo-code for node n leaving *p*Cluster.

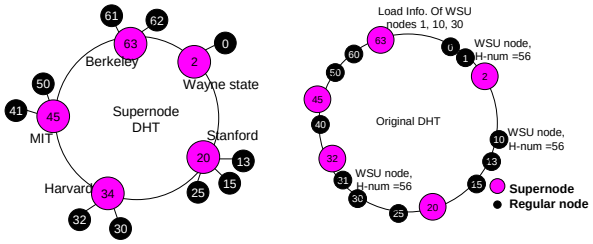
```
n.leave(){
 1: if n is a regular node then
 2:    notify(supernode);
 3: else
 4:    //n is a supernode
 5:    if backuplist.size>0 then
 6:       s=backuplist.getone();
 7:       //transfer supernode information to a backup
 8:       s.clientlist=clientlist;
 9:       s.backuplist=backuplist;
10:    else
11:       //no backup supernode, transfer regular nodes
12:       for i = 0 up to clientlist.size do
13:          client=clientlist[i];
14:          if predecessor is closer to client than successor then
15:             move client to predecessor;
16:          else
17:             move client to successor;
18:          end if
19:       end for
20:    end if
21: end if
22: }
```

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. DHT protocols dealing with node and item joins and departures can be directly used to handle supernode and regular node joins and departures in *p*Cluster. When a node joins the *p*Cluster, it must know at least one node, and uses *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, 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 *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. Algorithm 1 and 2 show the pseudocode of node join and departure in *p*Cluster, respectively.

Figure 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 Chord. Each supernode functions as a node in a flat Chord. If *n*40 wants to join in the *p*Cluster, *n*40 asks its known node *n*2 to find the supernode with Id closest to 40 based on routing algorithm, which is *n*45. If *n*40 is a supernode, *n*45 moves *n*41 to *n*40. 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 *n*40. It is the responsibility of stabilization to build routing table and other links for *n*40, and update other supernode routing tables. If *n*40 is a regular node, it becomes a client of *n*45. If a node, say *n*45, wants to leave the system. According to Algorithm 2, it moves *n*41 to *n*34, and *n*50 to *n*63. The routing tables which have *n*45 will be updated in stabilization. If *n*41 wants to leave the *p*Cluster, it only need to disconnect its link to *n*45.

Node failure is an important problem in DHT since it leads to intact topology and deprecate 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 get a reply from its supernode s after a certain time period T, n assumes s fails, it uses *p*Cluster node join protocol to connect to another supernode again. For example, if *n*41 does not get reply from *n*45 after T, by joining algorithm, it will connects to *n*32.

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 gather together in the supernode. For example, Nodes *n*61 and *n*62 report their load information to *n*63 periodically, which does load rearrangement, and notify heavy nodes to move excess load to light nodes.



(a) *p*Cluster (b) *v*Cluster

Figure 1: Example of proximity-aware DHTs.

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 or logically close supernode in the load balancing process. Algorithms 3 and 4 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 use 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,objet)`. 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. And the nodes further forward the information to their supernodes. Figure 1(b) shows an example of *v*Cluster in Chord. In the example, regular node *n*1, *n*10 and *n*30 send their load information to the DHT with their Hilbert number 56 as destination. The information will first arrive at *n*60, and then is forwarded to *n*63. The *n*63 does load rearrangement between physically close nodes *n*1, *n*10 and *n*30.

3.3 *p*Cluster versus *v*Cluster

Both *p*Cluster and *v*Cluster facilitate locality-aware load balancing. They achieve the goal in different ways. In

Algorithm 3 Pseudo-code for node n joining in *v*Cluster containing node n'.

```
n.join(n'){
1:  //get successor using Chord joining algorithm
2:  n.joinChord(n')
3:  if n is a regular node then
4:    //get the supernodes of its successor and predecessor
5:    suc_supernode=successor.supernode;
6:    pre_supernode=successor.predecessor.supernode;
7:    //find a closer supernode
8:    if pre_supernode is closer to n than suc_supernode then
9:      supernode=pre_supernode
10:   else
11:     supernode=suc_supernode
12:   end if
13:   supernode.addto_clientlist(n);
14: else
15:   //n is a supernode
16:   successor.supernodejoin_forwardnotify(n);
17:   successor.predecessor.supernodejoin_backwardnotify(n);
18: end if
19: }
20:
21: //n get supernode n' join notification
22: n.supernodejoin_backwardnotify(n'){
23: if n is a regular node then
24:   if n is closer to n' than supernode then
25:     supernode=n'
26:     supernode.addto_clientlist(n);
27:     predecessor.supernodejoin_backwardnotify(n');
28:   end if
29: end if
30: }
31:
32: n.supernodejoin_forwardnotify(n'){
33: the same as supernodejoin_backwardnotify, except that the
    last instruction should be:
    successor.supernodejoin_forwardnotify(n');
34: }
```

Algorithm 4 Pseudo-code for node n leaving *v*Cluster.

```
n.leave(){
1:  if n is a supernode then
2:    //transfer regular nodes to their closest supernode
3:    suc_s=find_supernode_forward();
4:    pre_s=find_supernode_backward();
5:    for i = 0 up to clientlist.size do
6:      client=clientlist[i];
7:      if client is closer to suc_s than pre_s then
8:        client.supernode_change_notify(suc_s);
9:      else
10:       client.supernode_change_notify(pre_s);
11:     end if
12:   end for
13: end if
14: }
```

*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 speedup load balancing. In contrast, *v*Cluster can save storage space and cost for supernode DHT construction and maintenance.

4 Locality-Aware Randomized 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 depredate 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 [12], Shen and Xu proposed locality-aware randomized (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. LAR algorithms run in two phases. First, regular nodes report their load information to certain supernodes. Recall that with the help of the auxiliary network, the load information of physically close nodes gather together in a supernode or close supernodes. Second, the supernodes arrange load movement. Each supernode has a pair of sorted donating lists (DSL) and starving lists (SSL). The DSL is used to store load information of light nodes and the SSL is used to store load information of heavy nodes. A supernode 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 SSLs and DSLs, which is called global load balancing.

LAR introduces a factor of randomness in the probing process in a range of proximity in global load balancing to deal with DHT proximity and dynamism. 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 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, a supernode randomly contacts other supernodes in its routing table and neighbor list first, in order to move load between relative closer nodes. After all neighbors are probed, the supernode randomly contacts other supernodes in the entire Id space.

5 Performance Evaluation

We designed and implemented a simulator in Java for evaluation of the LAR based on *p*Cluster (*p*LAR) and *v*Cluster (*v*LAR) on Chord DHT and compare their performance with churn resilient algorithm (CRA) [5], and KTree method [20]. 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, and also compared the resilience of the schemes in Chord with churn. In CRA, we set 16 directories as in [5]. We set the load information size threshold for load balancing in each KTree node as 15. For simplicity, we define a node with capacity greater than a predefined threshold as supernode; otherwise a regular node. Table 1 lists the parameters of the simulation and their default values. In the following, *node utilization* represents the fraction of its target capacity that is used, and *system utilization* represents the fraction of the system's total target capacity that is used.

We use two transit-stub topologies generated by GT-ITM [17]: "ts5k-large" and "ts5k-small" with approximately 5,000 nodes each. "ts5k-large" has 5 transit do-

Table 1: Simulated environment and algorithm parameters.

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:25000, upper bound: 25000*10
Supernode threshold	50000
Number of items	20480
Existing item load	Bounded Pareto: shape: 2, lower bound: mean item actual load/2 upper bound: mean item actual load/2*10

mains, 3 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, 5 transit nodes per transit domain, 4 stub domains attached to each transit node, and 2 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.

5.1 Proximity-Aware Load Balancing

In this section, we will show how *p*Cluster and *v*Cluster help LAR to achieve high proximity-aware performance. Figure 2(a) and (b) show the cumulative distribution function (CDF) of total moved load of each load balancing scheme with system utilization approaches to 1 in "ts5k-large" and "ts5k-small" respectively. 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, KTree move most load in short distances while CRA move most load in long distances. From Figure 2(b), we can have the same observations as in "ts5k-large," although the performance difference between schemes is not so significant as in "ts5k-large." The more load moved in the shorter distance, the higher 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 regards 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 when nodes are from several big sub domains or when nodes are scattered in the entire Internet.

5.1.1 Breakdown of Load Movement Cost

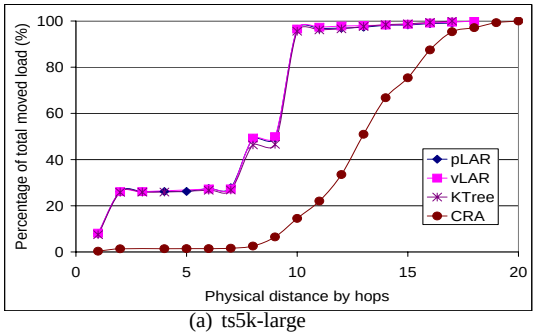
In general, a load balancing process needs to gather node load information in a number of rendezvous nodes, which arrange load movement. Figure 3 shows the breakdown of total moved load in percentage of the moved load in local or in global load balancing phase. We find that most load is moved in the local phase. 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.

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 means 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 depredation because excess load 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 [5], this number has significant impact on proximity-aware load balancing.

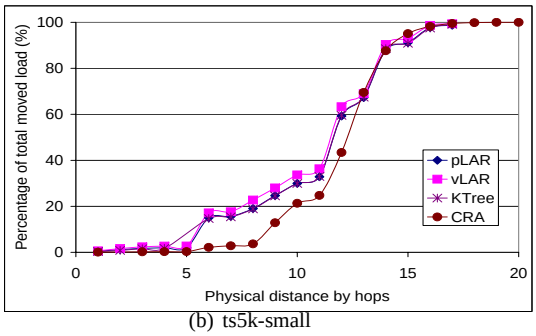
5.1.2 Communication Cost

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. Figure 4(a) and (b) plot 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), Figure 5 gives breakdown of the cost when the system is heavily loaded. The figure shows that the reporting cost 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. Though a node contacts its physically close nodes for load

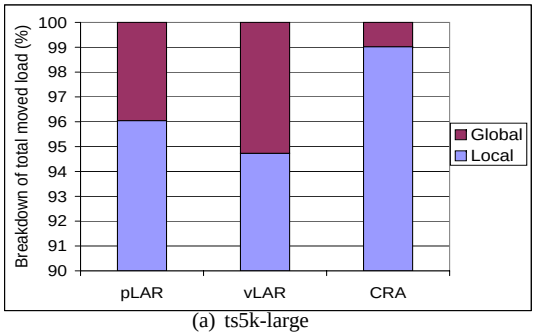


(a) ts5k-large

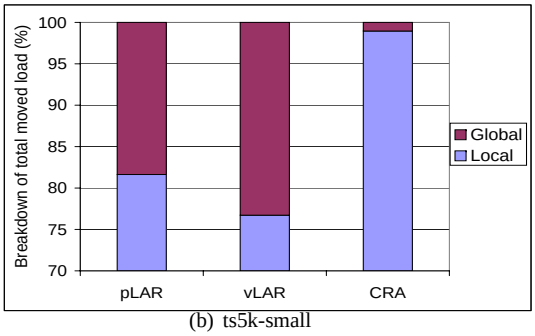


(b) ts5k-small

Figure 2: CDF of total moved load distribution of different load balancing schemes.



(a) ts5k-large



(b) ts5k-small

Figure 3: Breakdown of total moved load of different load balancing schemes.

balancing, the tree structure requires the load information to be reported step by step upward to the root. In contrast, randomized probing in other schemes reduces the cost.

Recall that *p*LAR enables nodes to report their load information to their physically closest supernode directly, and enables a supernode probe its peers in top-level supernode DHT with short path length; while the load and probing 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 than *v*LAR and CRA, and it costs less in the probing phase than *v*LAR. Due to the fact that almost all excess load is solved in local load balancing as shown in Figure 3, CRA has least 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.

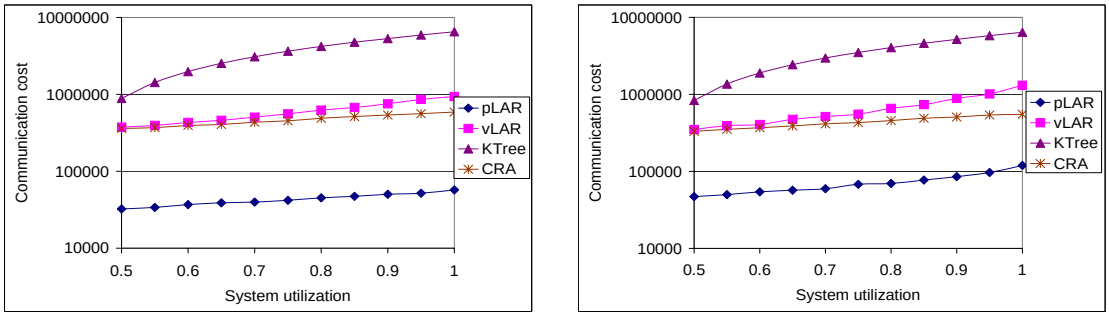
5.2 Churn Resilient Load Balancing

Churn in P2P networks gives load balancing schemes a challenge because it is hard to achieve load balance with frequent node and item joins and departures. For example, a node become overloaded if it cannot provide sufficient capacity for the load transferred by its leaving neighbors; fast and continuous item joinings in a specific node make the node overloaded; when rendezvous nodes for load rearrangement suddenly leave or fail, some nodes may cannot 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.

We evaluated the efficiency of the *p*LAR and *v*LAR in dynamic situations with respect to a number of performance factors. 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 seconds 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 seconds. We ranged node interarrival time from 10 to 90 seconds, with 10 second increment in each step. The system utilization was set to 0.8. We adopted the same metrics as in [5].

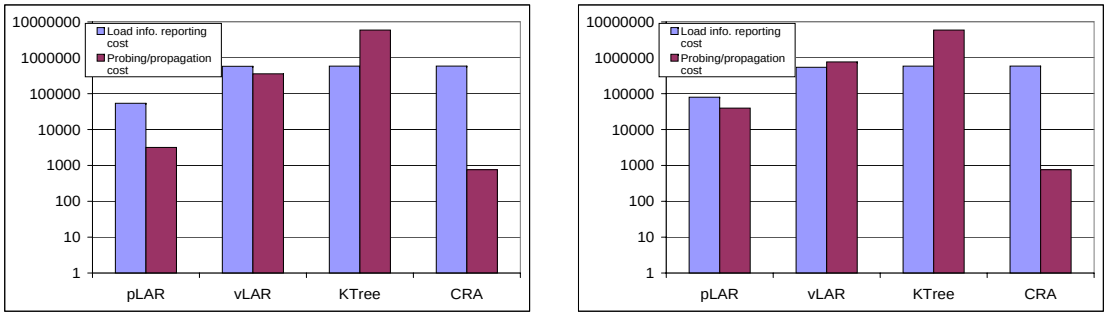
(1) *Maximum load movement factor*. Load movement factor is the total load transferred due to load balancing divided by the system actual load. We measure the factor after each T and take the maximum of the results over a 20T period as the maximum load movement factor.

(2) *Maximum and average 99.9th percentile node utilizations*. We measure the maximum 99.9th percentile of the node utilizations after each T and take the maximum and average of the results over 20T as the maximum and average 99.9th percentile node utilizations.



(a) ts5k-large (b) ts5k-small

Figure 4: Communication cost of different load balancing schemes.



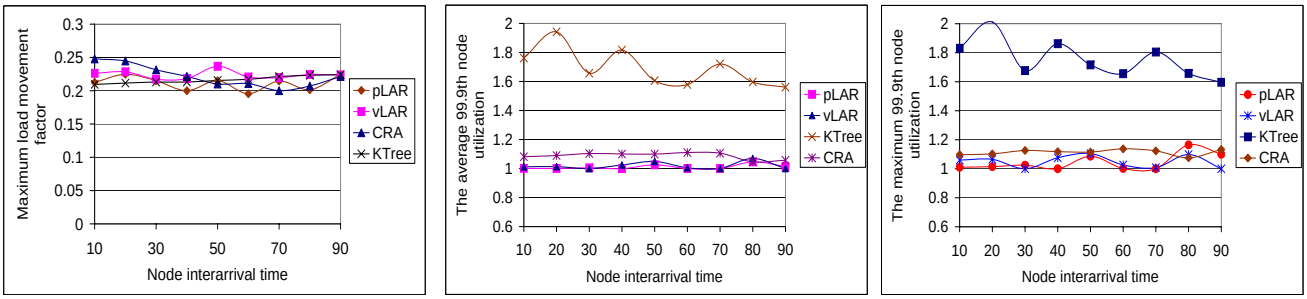
(a) ts5k-large (b) ts5k-small

Figure 5: Breakdown of total communication cost of different load balancing schemes.

Figure 6 plots the performance due to *p*LAR, *v*LAR, KTree and CRA versus node interarrival time. Figure 6(a) shows that the maximum load movement factor of each scheme is 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. Node utilization is a metric to evaluate this performance. Figure 6(b) and (c) show 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. Keeping the node utilization close to 1 implies that on average, *p*LAR and *v*LAR can achieve the load balancing goal of keeping each node light even in churn. The results of *p*LAR and *v*LAR are comparable to CRA implies that *p*LAR and *v*LAR 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. 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.

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 locality-aware randomized load balancing algorithms. 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.



(a) Maximum load movement factor (b) Average node utilization (c) Maximum node utilization

Figure 6: Effect of different load balancing schemes in DHT networks with churn.

Acknowledgement

This research was supported in part by U.S. NSF grant ACI-0203592 and NASA grant 03-OBPR-01-0049.

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, 1997.

[2] M. Bienkowski, M. Korzeniowski, and F. M. auf der Heide. Dynamic load balancing in distributed hash tables. In *Proc. of IPTPS*, 2005.

[3] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. Presented at Intl. Workshop on Future Directions in Distributed Computing, 2002.

[4] L. Garces-Erice, E. W. Biersack, K. W. Ross, P. A. Felber, and G. Urvoy-Keller. Hierarchical p2p systems. In *Proc. of ACM/IFIP International Conference on Parallel and Distributed Computing (Europar)*, 2003.

[5] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *Proc. of IEEE INFOCOM*, 2004.

[6] P. B. Godfrey and I. Stoica. Heterogeneity and load balance in distributed hash tables. In *Proc. of IEEE INFOCOM*, 2005.

[7] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proc. of IPTPS*, 2004.

[8] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Proc. of IPTPS*, 2003.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, pages 329–350, 2001.

[10] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. of IEEE INFOCOM*, 2002.

[11] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, 2001.

[12] H. Shen and C. Xu. Locality-aware randomized load balancing algorithms for structured p2p networks. In *Proc. of ICPP*, pages 529–536, 2005.

[13] H. Shen, C. Xu, and G. Chen. Cycloid: A scalable constant-degree p2p overlay network. *Performance Evaluation*, 2006.

[14] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 1(1):17–32.

[15] M. Waldvogel and R. Rinaldi. Efficient topology-aware overlay network. In *Proc. of ACM Workshop on HotNets*, 2002.

[16] Z. Xu, M. Mahalingam, and M. Karlsson. Turning heterogeneity into an advantage in overlay routing. In *Proc. of IEEE INFOCOM*, 2003.

[17] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proc. of IEEE INFOCOM*, 1996.

[18] B. Zhao, Y. Duan, L. Huang, A. Joseph, and J. Kubiatowicz. Brocade:landmark routing on overlay netwoks. In *Proc. of IPTPS*, 2002.

[19] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: An Infrastructure for Fault-tolerant wide-area location and routing. *IEEE Journal on Selected Areas in Communications*, 12(1):41–53, 2004.

[20] Y. Zhu and Y. Hu. Efficient, proximity-aware load balancing for DHT-based p2p systems. *IEEE Transactions on Parallel and Distributed Systems*, 16(4), 2005.