Brushwood: Distributed Trees in Peer-to-Peer Systems

Chi Zhang[1], Arvind Krishnamurthy[2,⋆], and Randolph Y. Wang[1,⋆⋆]

[1] Princeton University
[2] Yale University

Abstract. There is an increasing demand for locality-preserving distribution of complex data structures in peer-to-peer systems. Current systems either do not preserve object locality or suffer from imbalances in data distribution, routing state, and/or query processing costs. In this position paper, we take a systematic approach that enables the deployment of searchable tree structures in p2p environments. We achieve distributed tree traversal with efficient routing distance and routing state. We show how to implement several p2p applications using distributed tree structures.

1 Introduction

In recent years, a group of Distributed Hash Table-based (DHT-based) peer-to-peer infrastructures, exemplified by Chord, Pastry, Tapestry, CAN, *etc.* [16,15,20,13], have received extensive attention. Such systems provide many attractive properties, including scalability, fault tolerance and network proximity. A number of applications have been built using DHTs, like distributed file systems and application level multicast. However, the original DHT schemes only provide searching in hashed key space, which is not sufficient to support applications with complex data structure and semantics [12,8]. To support such applications, some specific schemes have been proposed to enhance DHTs. For example, Space Filling Curves [1], Prefix Hash Tree [14], *etc.*, are used to support range queries, but these approaches are specific to their target problems. For many applications, it is not clear how to distribute existing data structures without destroying the intrinsic locality critical to performance. In this paper, we propose a general paradigm for distributing and searching tree data structures in peer-to-peer environments while preserving data locality.

1.1 Locality-Sensitive Applications

We target data-intensive applications that can benefit from locality-preserving distributions in one of two manners. On one hand, many of our target applications require support for queries that are more complex than exact lookups in a flat name space. For such applications, the data is typically organized in hierarchical search trees that enable them to perform similarity queries and updates. On the other hand, for some of our target applications, locality-preserving data organization is a critical performance issue. These applications often exhibit strong correlation among data accesses. For example, file system users frequently access a small set of files and directories. The logical

⋆ Krishnamurthy is supported by NSF grants CCR-9985304, ANI-0207399 and CCR-0209122.
⋆⋆ Wang is supported by NSF grants CCR-9984790 and CCR-0313089.

M. Castro and R. van Renesse (Eds.): IPTPS 2005, LNCS 3640, pp. 47–57, 2005.
© Springer-Verlag Berlin Heidelberg 2005

structure of a tree hierarchy is a good representation of the access locality in such applications. Centralized systems often benefit from access locality when the data structure is laid out appropriately on secondary storage. In a distributed peer-to-peer system, the high communication costs, which can be thought of as being analogous to the storage access latency and throughput limits of centralized systems, make locality-preserving distributions essential. These issues lead to the following question: can we implement these hierarchical tree data structures in peer-to-peer computing platforms while preserving the inherrent locality?

1.2 Challenges to Peer-to-Peer Systems

Besides common requirements like scalable peer state and efficient routing, a peer-to-peer searchable tree faces several other problems:

- **Tree lookups:** In some trees, the search key is given as a tree path. In more general cases, the path or ID of the destination node(s) is not known a priori, but discovered through top-down tree lookup for a data key. With high network communication costs, efficient lookup demands high locality in the mapping of nearby tree nodes and minimal routing steps when the nodes are apart. Some systems use DHTs to distribute individual tree nodes, possibly by hashing each node to a unique key. To search such trees, a DHT lookup is needed for every tree edge starting from the root, resulting in lookup costs that could be as high as $O(\log^2(n))$ [4] or $O(\log\log(n) \cdot \log(n))$ [14] for structures with balanced depth. This process can be even more inefficient if the tree has long branches. Besides, the root tends to become a bottleneck and a single point of failure.
- **Skewed data and load balancing:** DHTs depend on hashing to ensure uniform distribution of data among participating processors. However, hashing destroys data locality and is therefore not suitable in our application settings. Using unhashed data keys suffers from skewed data distribution. Some systems, such as [17], use sampling techniques to achieve asymptotic load balance. However, in case of dynamic load changes, reactive load balancing schemes are more desirable [11,6].
- **Tree maintenance:** Most practical tree structures are dynamic, as they are subject to online insertion, deletion and structural changes. While the maintenance is easy in centralized settings, it can affect many nodes in a distributed tree. For example, a distributed B-tree [10] replicates internal nodes to improve search efficiency. This optimization however requires the system to perform tree updates in a consistent manner, thereby requiring complex protocols for maintaining tree consistency.

In this paper, we propose a distributed tree scheme called Brushwood. We solve the problems of how to partition a tree while preserving locality and load balance and how to search the partitioned tree efficiently in peer-to-peer systems.

2 Design of Brushwood

Our solution is based on a linearization of the tree. The upper half of Figure 1 (a) illustrates a file system tree. The directories are drawn as circles. Edges labeled with

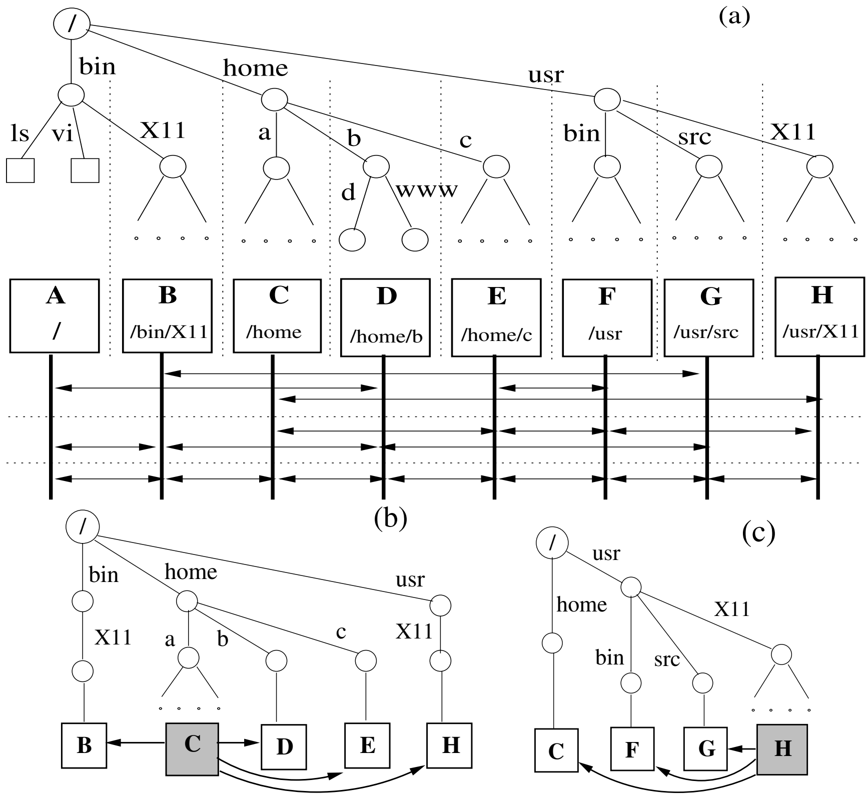


Fig. 1. Partitioning and Distribution of a File Tree

names represent directory entries. We linearize the tree nodes by pre-order traversal and then partition them into eight segments as shown by the dotted vertical bars. This partitioning method preserves locality since the low level subtrees are not split. The partitions are assigned to eight processors A - H, shown as the rectangles below the tree. We use the word "processor" to denote peer-to-peer nodes, in order to avoid confusion with tree nodes. Each processor is identified by its left boundary, which is the left-most tree node in the partition. The path name inside a processor box shows the left boundary of that partition.

To ensure system scalability, we limit the knowledge of individual processors about the tree and other peers. Each processor only knows $\log N$ peers and their partition boundaries in an N-processor system. A tree lookup can be done within $\log N$ steps regardless of the shape of the tree. We extend Skip Graphs/Nets [3,9] to achieve such an efficient lookup.

Conceptually, a processor in a Skip Graph maintains $\log N$ levels of peer pointers, pointing to exponentially farther peers in the linear ordering of N processors. The arrows under processor boxes in Figure 1 depict the three levels of peer pointers between

the processors. Processors construct their local partial view of the tree from the boundaries of their peers. Figure 1 (b), (c) show the partial view of C and H, respectively.

Now we show how to perform object location in a distributed tree by illustrating the lookup of file /bin/X11/X from processor H. H uses its partial view to find the the peer that is farthest in the same direction as the target (given the pre-order linearization of tree nodes) without passing over the target. In this example, H determines that the target is to the left of peer C, which has a boundary of /home, and it forwards the request to C. C in turn uses its partial tree view, and determines that the peer that is closest to the target is peer B with a left boundary of /bin/X11. So it forwards the request to B. Tree lookup is therefore performed starting from any processor by "jumping" among the processors with each hop reducing the distance to the target, instead of traversing a tree path from the root to the target. The number of hops is therefore logarithmic in the number of processors, regardless of tree depth.

Generally, an application tree provides two pieces of information to enable the distributed lookup toward a target key:

- **A label l_{edge} on each tree edge.** There is a total order among the labels on edges out of a node, for example, the dictionary order for the entry names.
- **A comparison function f_{node} in each tree node.** This function compares a target key to the label of an edge of this node, telling whether it matches this edge, or falls to the left/right of it.

A node identifies its partition by a sequence of $\langle f_{node}, l_{edge} \rangle$ values from the root to its left boundary node. We define this sequence as the *Tree ID*. This ID is sent to peers so that a partial tree view can be constructed. The nature of the target key, f_{node}, and l_{edge} values are specific to the application. For example, in a file system tree, target keys are directory paths, each l_{edge} is a string, and f_{node} is simply string comparison. In more general trees, the target might not be specified explicitly by a tree path. For example, in a high dimensional index tree (see Section 3.1), each tree node corresponds to a region of space, the target key is simply a point coordinate or a range, and f_{node} encapsulates information regarding a *split plane* that can be used to decide which branch to follow.

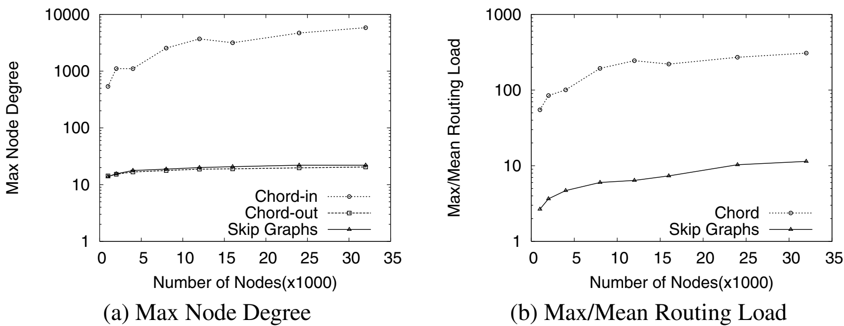
For certain operations, such as a range query in high dimensional space (Section 3.1), the target objects are located by a generalization of the above process. The querying node may find that the target range is relevant to more than one branch, and it would therefore forward the request to multiple peers simultaneously, resulting in a "multicast" query.

Maintaining the partitioned tree in the above scheme is quite simple. Insertion and deletion of a branch only affects the processor whose boundaries enclose the target branch. For instance, insertion of /home/b1 affects only processor D.

Several optimizations are possible in Brushwood distributed tree. We provides data redundancy by allowing neighboring processors to maintain overlapping partitions. Besides added availability, it also improves locality, because the partitions now cover larger subtrees. The P-Table mechanism from Skip Nets provides proximity-aware routing similar to Pastry. It can be further enhanced by proximity-aware load balancing (Section 2.2).

2.1 Choice of Routing Substrate

Our tree routing depends on a linear ordering of partitions. In this sense, any linear space DHT routing facility can be used. We choose Skip Graphs for two reasons. First of all, Skip Graphs do not impose constraints on the nature and structure of keys. It can work with complex keys, like the variable-length Tree IDs, as long as there is a total ordering. Second, even if one can encode tree nodes into key values, such unhashed and often skewed keys can cause routing imbalance in some DHTs, as they use key values to decide peering relation. Skip Graphs do not suffer from this problem because its peering is decided by purely random membership vectors, even though the keys are unhashed.



(a) Max Node Degree (b) Max/Mean Routing Load

Fig. 2. Imbalance under Skewed Key Distribution

We simulated Chord and Skip Graphs with a skewed key distribution to show the imbalance in routing. Figure 2 (a) depicts the maximal processor degrees of Chord and Skip Graphs with 1K$\sim$32K processors. The processor keys are derived from a normal distribution with standard deviation 0.125 in the range $[0, 1]$. With such unhashed keys, Chord processors falling into the sparsely populated regions will manage larger portions of the keyspace, and are therefore likely to have a large number of in-bound peers. Furthermore, the imbalance in peer distribution also leads to imbalance in routing costs. We route 1000 messages between random pairs of nodes. Figure 2 (b) shows the imbalance as the ratio of maximal routing load to mean load.

2.2 Load Balancing

Balancing the assignment of tree nodes to processors is an important issue, because the distribution of items in the tree could be skewed and might also change with time. We propose a dynamic load balancing scheme that augments previous work [11,6,2].

Each processor maintains load information about the nodes in its partial tree. The load in an internal node is the aggregated load on all processors managing portions of this node. The root node therefore is associated with the global average load. Each processor periodically gets load information from its peers and does its aggregation from the bottom up the partial tree. Load information therefore propagates through the

entire system via a combination of local aggregation steps and peer-to-peer exchanges. This process can be proved to converge after $O(\log N)$ steps.

There are two types of load balance operations, both taking advantage of the load information in the partial tree. When a processor joins, it navigates the tree to find a processor with high load, and partitions its data set. If a processor sustains significantly higher load than global average, it may navigate the tree to find an underloaded processor. This processor is forced to quit its current position and rejoin to take over half of the load from the overloaded processor. We favor a physically nearby processor in the above navigation, so that the data items may retain network proximity after the partition.

3 Applications

3.1 Multi-dimensional Indexing

The first application we build with Brushwood is a high dimensional index supporting complex queries. The data set being indexed are points in a D-dimensional Cartesian space. The typical queries are not exact point matches, but are searches for points falling in a certain range, or close to a given point. Such data sets are frequently found in multimedia databases, geographic information systems, data mining, decision support, pattern recognition, and even text document retrieval.

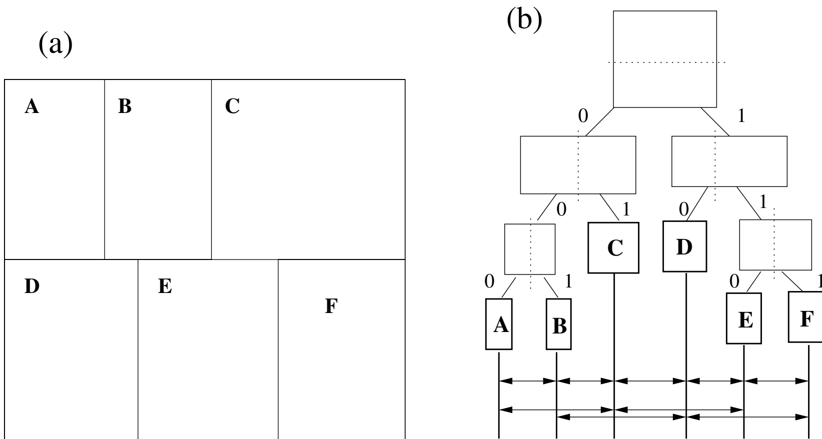


Fig. 3. Partitioning of Search Space

Partitioning K-D Tree. SkipIndex [19], our peer-to-peer high dimensional index, distributes a K-D tree [5] with Brushwood. K-D tree is a widely used index tree for high dimensional data. It hierarchically partitions the search space and data set into smaller and smaller regions. Each internal node specifies a partition dimension and a split position, and splits its region into two children. The data points are stored in leaf nodes. Figure 3 (a) illustrates partitioning of a 2-D search space to six processors, (b) shows the corresponding K-D tree and the skip graph routing tables.

Insertion and query operations in SkipIndex navigate the distributed tree to reach appropriate leaf nodes. The target is specified by a high-dimension point (insertion) or range (range query). To enable Brushwood tree lookup, SkipIndex defines the following elements:

- l_{edge} is 0 or 1, denoting left or right child.
- f_{node} compares the target point or range to the splitting plane of the node. For a point, it only returns one matching child branch. For a range, it may return both branches.

As we described before, the tree ID of a processor is given by the tree path from the root to the left boundary node of its partition. For each internal node along the path, it includes a tuple of $\langle dim_{split}, pos_{split}, 0/1 \rangle$, specifying the dimension and position of the split plane and the branch taken. A processor builds its routing state as a partial K-D tree containing the tree IDs of peers and itself.

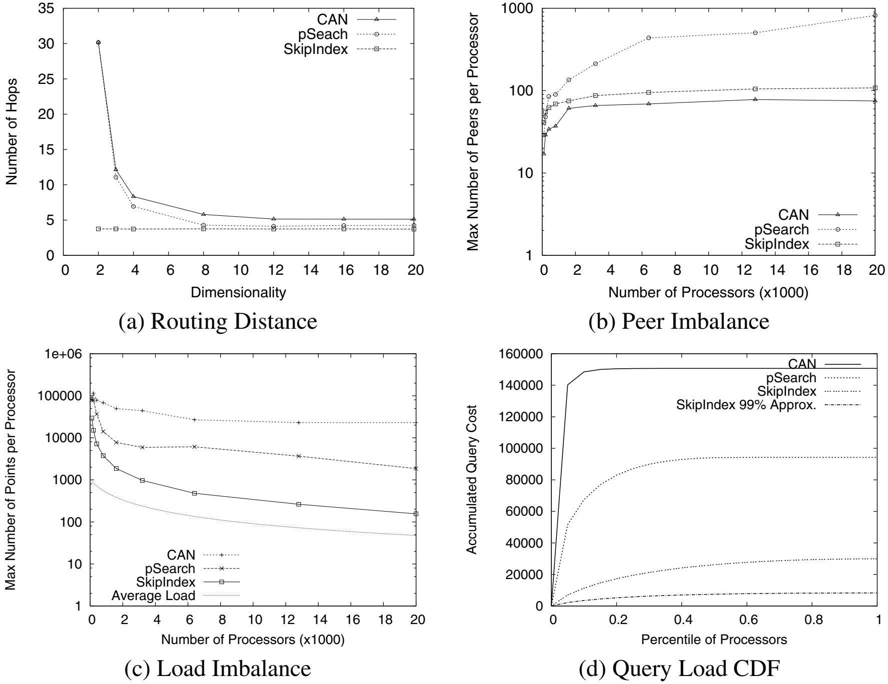
When a processor joins, it locates a heavily loaded node (Section 2.2) and partitions its search space. A key benefit provided by Brushwood is the flexible choice of the split plane. We partition the most distinguishing dimension, so that the points in the two partitions are less similar, and the partitions are less likely to be involved together in a query. We split along the median of the items to balance the load.

Insertion and lookup of a point is straight-forward. At each hop, the processor navigates its partial tree by comparing the point to the split planes in tree nodes from root down, and it forwards the request to a peer that is maintaining a region that is closest to the target point.

Complex Queries. Range query in SkipIndex exploits another type of tree lookup. Now the target is a range in high dimensional space. While navigating the partial view, at each tree node, the target range is tested for intersection with the regions corresponding to its children. Each intersecting branch is further traversed until the traversal reaches the leaves of the partial tree. If the leaf is a remote region, a request is routed to the peer to search within the region. Otherwise, a local search is performed to find matching points.

Nearest neighbor search returns k points having the smallest Euclidean distances to a query point. While range query performs a parallel lookup of the K-D tree, our nearest neighbor search algorithm performs a sequence of lookups, gradually refining the results. At each lookup step, a search request is routed to a processor managing a search region close to the query point. Such regions are searched in the order of expanding distances from the query point. We can perform an exact search where we exhaust all processors that may contain a closer point. We also provide an approximate search that significantly reduces the search cost with controllable accuracy.

We evaluated SkipIndex with a 20-dimension image feature vector data set. This data set is highly skewed. We compare with pSearch [17] which uses unhashed CAN to index high dimensional vectors. Compared to CAN, SkipIndex allows more flexible partition of search space. Brushwood routing is also more stable in face of skewed data distribution. In Figure 4 (a), Brushwood routing in SkipIndex shows routing distances unaffected by data dimension, while CAN and pSearch suffer when data dimension is



(a) Routing Distance (b) Peer Imbalance

(c) Load Imbalance (d) Query Load CDF

Fig. 4. Routing and load balance comparisons

low. Figure 4 (b) compares the maximal number of peers. Brushwood/SkipIndex routing exhibits more stable routing state, which confirms the analysis in Section 2.1. Under skewed data distribution, SkipIndex enjoys better load balance as shown in Figure 4 (c). Figure 5 compares the nearest neighbor search cost measured by the number of processors visited per query, averaged across 1000 queries. SkipIndex achieves lower exact search cost than pSearch thanks to the flexibility in space partitioning. Approximation further reduces the cost significantly. Note that this query span does not fully reflect the cost of search, because the load on the processors may be unequal. CAN achieves low query span as most of the data objects are maintained by a small number of high load processors. To better understand the search cost, Figure 4 (d) depicts the CDF of query load measured by the number of object distance calculations during the search. SkipIndex exhibits lower total cost and more balanced load distribution.

3.2 Distributed File Service

Now we go back to the example in Section 2 to review the potential for implementing a partitioned file service using Brushwood. It is well known that disk locality is critical to file system performance. In a distributed file service, locality of a file and its directory also impacts performance, since the lookup of objects costs network communication. By keeping related objects on the same processor, one can reduce the lookup overhead.

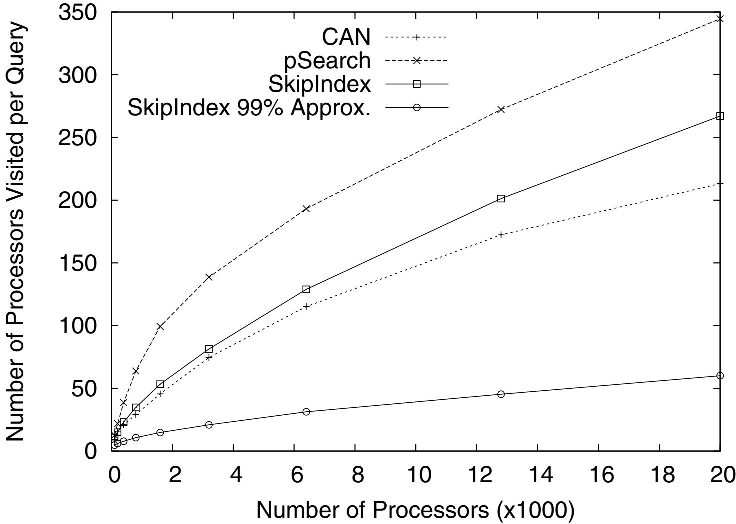


Fig. 5. Nearest Neighbor Search Cost

Availability is another reason to consider distribution locality. Accessing a large set of processors for a given task is more vulnerable to failures than accessing a few, if the redundancy level is the same.

We analyze an NFS trace from Harvard University [7] to confirm the above observations. The trace was collected on EECS department server running research workload. We use a week-long period of October 22 to 28, 2001. There are a total of 29 million requests involving 540K file handles. We reconstructed the file system tree from the trace. The tree is split into 1000 partitions using the load balancing process described in Section 2.2.

To measure the access locality, we identify the user "sessions" in the trace activities. A session is defined as a series of operations sent by the same user with intervals less than 5 minutes. The maximal length of a session is limited to 1 hour. There are a total of 6470 sessions in the period, with an average duration of 701.8 seconds. The user activity shows strong locality within a session. Table 1 gives the number of unique blocks/files/directories/partitions accessed during an average session. Tree partition appears to be the best granularity to exploit locality.

To evaluate availability, we replay the trace with Poisson failures. We set the mean-time-to-failure as 10 hours, and the mean-time-to-repair as 5 minutes to simulate a dynamic peer-to-peer environment. The file system is distributed to 1000 processors with four different schemes: hashing by block ID, hashing by file ID, hashing by directory ID, and tree partitioning. We randomly place two copies of each block/file/directory/partition on the processors. Only if both replicas fail, a request fails. The second row of Table 1 shows the number of sessions experiencing request failures under different distribution schemes. When data locality improves, a client depends on less number of servers to perform the same task. Therefore, better locality reduces the chance of encountering server failures.

Table 1. Trace Analysis Results

Distribution scheme	Block	File	Directory	Partition
Number of unique objects accessed per session	1594.14	117.28	21.26	6.01
Number of sessions seeing request failures	236	153	53	21

4 Related Work

As far as we know, our work is the first general scheme to efficiently distribute, maintain, and traverse search trees in peer-to-peer systems. Previous efforts on distributed search trees, like replicated B-tree [10], focus on parallelizing the operations and do not exploit the symmetric node capability of peer-to-peer systems. DHTs like CAN, Chord, Pastry and Tapestry achieve scalability and resilience by building self-organizing overlays to locate resources in peer-to-peer systems. But since these systems use hashing to achieve load-balance, they are not suitable for maintaining complex data structures. Several schemes [17,6] use unhashed DHTs for complex queries in flat key space, but it is not clear how to build a general search tree.

Our dynamic load balancing scheme is inspired by previous work [11,6]. However, instead of using random sampling, our scheme uses peer-wise gossiping to aggregate load information in the distributed tree, which directs reactive load adjustment operations. Similar aggregation schemes are used in previous systems like [18].

Multi-dimensional queries in peer-to-peer systems have been addressed in a few other systems. We had discussed pSearch earlier. Mercury [6] provides range query by indexing the data set along each individual attributes. It uses random sampling to ensure efficient routing ($O(\log^2 N)$ hops) under skewed data distribution. However, the per-attribute index makes Mercury inappropriate for nearest neighbor query which involves all dimensions.

5 Conclusions

In this paper, we propose a general scheme to efficiently distribute and navigate tree data structures in peer-to-peer systems. The approach is shown to be effective in several locality-sensitive applications. We believe that more applications will benefit from this system for maintaining complex data structures in peer-to-peer environments.

References

1. A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Second IEEE International Conference on Peer-to-Peer Computing*, 2002.
2. J. Aspnes, J. Kirsch, and A. Krishnamurthy. Load balancking and locality in range-queriable data structures. In *Proc. of PODC*, 2004.
3. J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of Symposium on Discrete Algorithms*, 2003.
4. B. Awerbuch and C. Scheideler. Peer-to-peer systems for Prefix Search. In *PODC*, 2003.

5. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9), 1975.
6. A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *SIGCOMM*, 2004.
7. D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing email and research workloads. In *USENIX Conference on File and Storage Technologies*, 2003.
8. M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *Proceedings of IPTPS02*, 2002.
9. N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USITS*, 2003.
10. T. Johnson and P. Krishna. Lazy updates for distributed search structures. In *Proceedings of ACM SIGMOD*, 1993.
11. D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *IPTPS*, 2004.
12. P. Keleher, B. Bhattacharjee, and B. Silaghi. Are virtualized overlay networks too much of a good thing. In *Proc. of IPTPS*, 2002.
13. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM*, 2001.
14. S. Ratnasamy, J. Hellerstein, and S. Shenker. Range Queries over DHTs. Technical Report IRB-TR-03-009, Intel Research, 2003.
15. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *ICDCS*, 2002.
16. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.
17. C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of SIGCOMM*, 2003.
18. R. van Renesse and K. P. Birman. Scalable management and data mining using astrolabe. In *IPTPS*, 2002.
19. C. Zhang, A. Krishnamurthy, and R. Y. Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. Technical Report TR-703-04, Princeton Univ. CS, 2004, http://www.cs.princeton.edu/∼chizhang/skipindex.pdf.
20. B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 2004.