CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **28**:1848–1869
Published online 11 November 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3160

SPECIAL ISSUE PAPER

Supporting multidimensional range queries in Hierarchically Distributed Tree

Yunfeng Gu*,†, Azzedine Boukerche, and Robson E. De Grande

PARADISE Research Laboratory, School of Information Technology and Engineering University of Ottawa, Ottawa, Canada

SUMMARY

An examination of the multidimensional range query in existing peer-to-peer (P2P) overlay networks indicates that multidimensional range queries are sensitive to underlying topologies; this is because partitioning and mapping of multidimensional data space are two interconnected parts of a process that must be carried out cooperatively. The first section focuses on how to preserve data localities, whereas the second section concerns how to accommodate and maintain data localities at the P2P overlay layer. There are many studies that have been conducted on the first section since 1966, and those works that are well accepted are mostly based on recursive decomposition, which forms a tree structure in nature. However, less effort has been made to provide comparable support from the P2P overlay layer. In our previous work, we proposed the Hierarchically Distributed Tree (HD Tree) in order to better support multidimensional range queries in the P2P overlay network. This paper further explores error-resilient routing and load balancing strategies that can be employed in the HD Tree. We also provide a complete set of experimental results for all routing operations: Join and Leave of nodes, range queries at different levels of selectivity, and the dynamic load balancing scheme. Comparisons are made by conducting simulations under both the ideal and the error-prone routing environment and within various ary HD Trees. The experimental results show that load balancing in the HD Tree can be adjusted dynamically and globally, and it is actually a trade-off between distributing the basic load and the involvement of nodes in range querying. The experimental results also indicate that a maximum of 10 percent of routing nodes' failures do not have significant effects on the performance of range queries. However, a lower ary HD Tree appears to have better routing performance, whereas a higher ary HD Tree achieves a higher fault-tolerant capacity. Nevertheless, the performance of range queries in a higher ary HD Tree can be further optimized if all possible routing options can be fully explored in the error-prone routing environment. Copyright © 2013 John Wiley & Sons, Ltd.

Received 30 May 2012; Revised 30 August 2013; Accepted 14 September 2013

KEY WORDS: fault-tolerance; error-resilient; distributed; data structure; HD tree; peer-to-peer (P2P); overlay, associative searching; multidimensional; range query

1. INTRODUCTION

Peer-to-peer (P2P) is a potentially disruptive technology with numerous applications, such as distributed directory system, P2P domain name services, network file systems, massively parallel systems, distributed e-mail systems, and massive-scale fault-tolerant systems. However, these applications cannot be realized without a robust and reliable P2P search network. The main challenge is how to locate the data requested from the application data space in the P2P identifier space. This problem has been solved by using a randomization function—the Distributed Hash Table [1–3]

*Correspondence to: Yunfeng Gu, PARADISE Research Laboratory, School of Information Technology and Engineering University of Ottawa, Ottawa, Canada.

†E-mail: yungu@site.uottawa.ca

(DHT). The DHT-based system is also known as the structured P2P overlay network, because it must normally maintain a tightly controlled topology. In a DHT-based system, a key is guaranteed to be found if it exists, and the performance of any operation can be theoretically proven and experimentally examined. Some very popular DHT-based systems include the following: CAN [4, 5] (d-torus); Chord [4, 6] (ring); Pastry [4, 7] (mesh); Tapestry [4, 8] (mesh); Kademlia [4, 9] (XOR metric); Viceroy [4, 10] (butterfly); etc. In addition to the structured P2P overlay network, the unstructured P2P overlay network is often organized in a flat or random manner. Data contents are replicated among peers according to popularity, and queries are easily made by using flooding or random walks. P2P overlay networks appear to be very effective at retrieving frequently accessed contents. Although the unstructured P2P overlay network is ad hoc in nature and does not guarantee the correctness and performance of a query, it is well suited for file sharing in the mass market. Some very popular unstructured P2P overlay networks are Gnutella [4, 11, 12], Freenet [4, 13], FastTrack/KaZaA [4, 14–17], BitTorrent [4, 18, 19], eDonkey [4, 20–23], etc.

Our recent work has focused on the multidimensional range query. We believe that this is the most generalized form of many other queries. The multidimensional range query in P2P environments provides efficient indexing services. It has many applications in large-scale distributed environments, such as grid computing, publish/subscribe systems, multiplayer games, group communication and naming, P2P data sharing, and global storage. Many of these applications collect, produce, distribute, and retrieve information from, and into, distributed data space. The scale of such data space can be extremely large, and its dimensionality varies. The distributing and managing of such data space, even the index part, appear to present a great challenge to distributed environments. However, for DHT-based systems, the very features that make them good, such as load balancing and random hashing, work against the range query. The main challenge is the preservation of data localities. We have to consider how data space is partitioned and mapped into the identifier space. Quad-tree [24–26], K-d tree [25–27], Z-order [28, 29], and Hilbert curve [30] are four well-known works in this area. They all share the following common feature: the use of recursive decomposition for data space partitioning, which forms a tree structure by nature. No matter which decomposition scheme is used, there are two facts that can be observed:

- *Data localities expand exponentially as the dimensionality of data space increases, and*
- *Data localities extend exponentially as the recursive decomposition of data space increases.*

Accordingly, the P2P overlay network layer should possess a certain inherent nature in order to accommodate and maintain data localities with exponentially expanding and extending rates. Among all existing P2P overlay networks we have examined so far, we do not consider DHT-based systems. SkipNet [31] and P-Grid [32] are designs that have claimed to support the range query; however, there have been no experimental results that can be found so far. Mercury[33] uses random sampling for data distribution, and its experimental results show a very limited selectivity. Multiattribute addressable network [34] employs locality-preserving hashing, and its maximum selectivity is about 1 percent in a two-dimensional data space. Other systems, such as SCARP [35], MURK [35], ZNet [36], Skipindex [37], Squid [38], and SONAR [39], which use recursive decomposition to perform partitioning, have underlying overlay architectures that are either adapted structures from DHT-based systems, such as Chord, CAN, or Skip Graphs [40], which are the only structures proposed for the range query. No system has direct mapping. They must all manage an extra tree-like structure internally. Their experimental results show very limited selectivity. We are unable to have a complete view of how the P2P system behaves in high-dimensional range queries with a wide range of selectivity.

Multidimensional range queries have a sensitive underlying topology. It is feasible to adapt existing P2P overlay structures that were originally designed for the exact key query; however, their inherent nature shows no consistency for accommodating data localities that have been well preserved using recursive decomposition schemes employed at a higher layer. This inconsistency not only involves considerable routing and maintenance costs but also, at the architectural level, makes a simple, scalable, and reliable P2P solution very difficult or even impossible. In our previous work [41], we proposed a novel distributed data structure, the Hierarchically Distributed Tree

DOI: 10.1002/cpe

(*HD Tree*) [41, 42], to support multidimensional range queries at the structured P2P overlay layer. The motivation for proposing this new data structure was stimulated by the observation that recursive decomposition has an inherent similarity to a tree structure. Our previous work intended to adapt the tree structure to distributed environments. However, it covered only the design of basic routing strategies and the dynamic features of nodes. In order to fully address a new P2P overlay structure, this paper further explores error-resilient routing strategies and the dynamic load balancing in *HD Tree*. We will present a complete set of simulation results for range querying in data spaces with various dimensionalities, at different levels of selectivity, and with both ideal and error-prone routing environments.

2. PREVIOUS RELATED WORK

In this paper, we use a five-tuple of letters (k, h, d, n, n_i) to denote the basic parameters of a complete tree: k refers to k-ary, h and d refer to the height and depth of a tree, n is the total number of nodes, and n_i is the total number of nodes at depth i.

2.1. Hierarchically Distributed Tree

'An *HD Tree* is a cyclic graph built over a complete k-ary tree; the add-on distributed structure is constructed by making a limited number of extra connections; each node in the system can be reached by any other node at the same depth via not only the hierarchical structure but also by the distributed structure and with comparable routing performance' [41].

In k-ary *HD Tree*, the base tree structure is coded by following the prefix coding strategy (see the solid links in Figure 1), and extra links are connected to nodes at the next depth by following the suffix coding strategy (see the dashed links in Figure 1). However, the root node has no extra links. An internal node with the same value at each digit of the code (*SPeer*) has $k - 1$ extra links, and all other internal nodes have $k + 1$ extra links; this is because the *SPeer* node has the same prefix and suffix code as the *SPeer* parent node and the *SPeer* child node. For the same reason, the *SPeer* leaf node has no extra links, and all other leaf nodes have only 1 extra link. Therefore, the total number of links in a full *HD Tree* constructed this way is $2(n - 1) - kh$.

In an *HD Tree* structure, the prefix coded sibling nodes are *HSiblings* of each other, whereas the suffix coded sibling nodes are *DSiblings* of each other. Accordingly, *HParent* and *HChild* (*HChildren*) are prefix coded parent and child (children) nodes, whereas *DParent* and *DChild* (*DChildren*) are suffix coded parent and child (children) nodes, respectively.

2.2. Basic routing strategies

2.2.1. Basic hierarchical operations. The basic hierarchical operation is a single hop time operation. It is mainly used to forward routing requests toward the destination depth. There are four basic hierarchical operations. *2HP* or *2DP* passes the routing request to a parent node. Alternately, *2HC* or *2DC* passes the routing request to a child node:

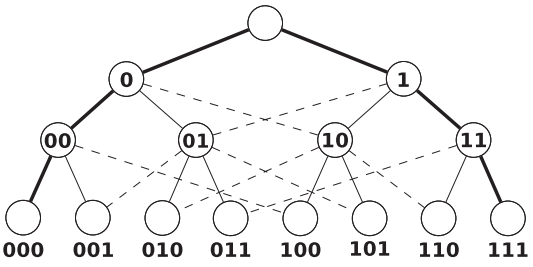


Figure 1. Constructing a 2-ary Hierarchically Distributed Tree.

DOI: 10.1002/cpe

$$abcde \xrightarrow{2HP} abcd \qquad abcde \xrightarrow{2DP} bcde$$
$$abcd \xrightarrow{2HC(e)} abcde \quad bcde \xrightarrow{2DC(a)} abcde^{\ddagger}$$

Hierarchical routing consists of a series of repeated basic hierarchical operations. It continuously forwards a routing request toward root or leaf nodes and reaches the destination node by crossing many different depths.

2.2.2. Basic distributed operations. Unlike basic hierarchical operations, the basic distributed operation is a 2-hop time operation. It consists of two basic hierarchical operations, which always involve a parent node and a child node. The basic distributed operation is mainly used to forward a routing request toward the destination node at the same depth. There are four basic distributed operations. *D2H* (*2DP* and *2HC*) and *H2D* (*2HP* and *2DC*) are digit-shift operations. *D2H* forwards the routing request to an *HSibling* via *DParent* or *HChild* while *H2D* forwards the routing request to a *DSibling* via *HParent* or *DChild*. On the other hand, *D2D* (*2DP* and *2DC*) and *H2H* (*2HP* and *2HC*) are digit-replacement operations. *D2D* forwards the routing request to a *DSibling* via *DParent* while *H2H* forwards the routing request to an *HSibling* via *HParent*.

D2H: $abcde \xrightarrow{2DP} bcde \xrightarrow{2HC(f)} bcdef$ or
$abcde \xrightarrow{2HC(f)} abcdef \xrightarrow{2DP} bcdef$

H2D: $abcde \xrightarrow{2HP} abcd \xrightarrow{2DC(f)} fabcd$ or
$abcde \xrightarrow{2DC(f)} fabcde \xrightarrow{2HP} fabcd$

D2D: $abcde \xrightarrow{2DP} bcde \xrightarrow{2DC(f)} fbcde$

H2H: $abcde \xrightarrow{2HP} abcd \xrightarrow{2HC(f)} abcdf$

Distributed routing is essentially a series of such digit operations that are continuously applied toward one direction. It forwards a routing request to a parent node and a child node alternately and reaches the destination node by traversing only two adjacent depths. *D2H* or *H2D* routing is a series of *D2H* or *H2D* operations, and *D2D* or *H2H* routing is a series of *D2H* or *H2D* operations followed by a *D2D* or *H2H* operation. During each operation of distributed routing, if the new digit is chosen properly by following every digit value in the destination code sequentially, the destination is guaranteed to be reached in, at most, d steps of such 2-hop time operations.

According to [41], the average case time complexity of distributed routing in the ideal routing environment can be computed as follows:

$$\mathcal{T}_{avg}(DR) \leqslant 2d - \frac{2}{k-1} + \frac{2}{k^d(k-1)}.$$

2.2.3. Basic routing mechanism. Routing in *HD Tree* is nothing more than a digit operation, which occurs only at the left-most or right-most side in the current *HCode*. By properly arranging these digit operations toward the destination code, the destination node can always be reached within $2h$ hops time at worst. However, better performance can be obtained if there exists a *match*. A *match* is a common sub-code that exists between the current *HCode* and the destination *HCode*. In our previous work [41], we proposed a distributed routing oriented combined routing algorithm (*DROCR*), which attempts to use the existing *match* and the distributed operations as much as possible. Hierarchical operations are applied only when the current node and the destination are not at the same depth, or when distributed operations cannot be applied. In this paper, we will extend *DROCR* algorithm to the error-prone routing environment.

[‡]Normally, *2HC(i)* and *2DC(i)* are used to denote 'send to the *i-th HChild* or *DChild*'.

DOI: 10.1002/cpe

In an ideal routing environment, hierarchical operations are to forward the routing request toward the destination's depth. If there is no *match* between the current node and the destination, there are two options to truncate the current *HCode* toward the destination code: *2HP* and *2DP*. There also exist two options to extend the current *HCode* toward the destination code: *2HC* and *2DC* (as shown in Figure 2). *2HC* or *2DC* must be applied in order to generate a *match* that starts from either the right-most side or the left-most side of the current *HCode*, and this can be carried out by choosing the child node whose sequence number equals the digit at the left-most side or the right-most side of the destination code. On the other hand, if a *match* can be found between the current node and the destination (as shown in Figure 3), *2HP* or *2DP* is applied to truncate the current *HCode* toward the *match*, and *2HC* or *2DC* is applied to extend the existing *match* toward the destination code.

When a node at the destination's depth receives a routing request, distributed operations may be applied to extend an existing *match* (shown in Figure 4) or to generate a *match* (shown in Figure 5) toward the destination node. If no distributed operation can be applied to the existing *match*, the hierarchical operation can always be used instead. The number of choices for the routing operation varies depending on the number of *matches* that can be found between the current node and the destination. If no *match* exists, there are at least two routing options: a *D2H* operation starting from the left-most side of the destination code and an *H2D* operation starting from the right-most side of the destination code (shown in Figure 5).

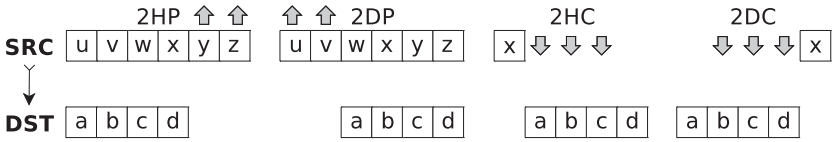


Figure 2. Hierarchical routing without match.

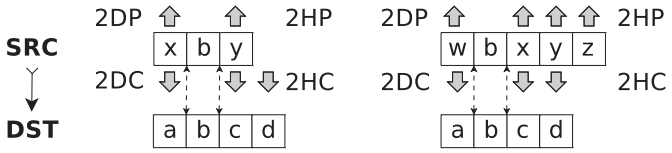


Figure 3. Hierarchical routing with match.

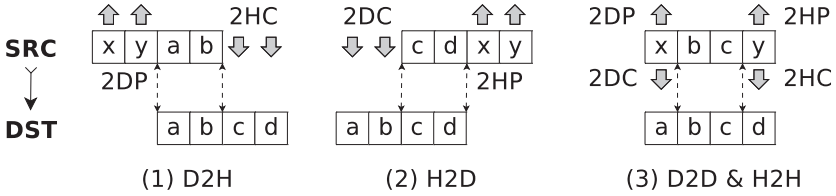


Figure 4. Distributed routing with match.

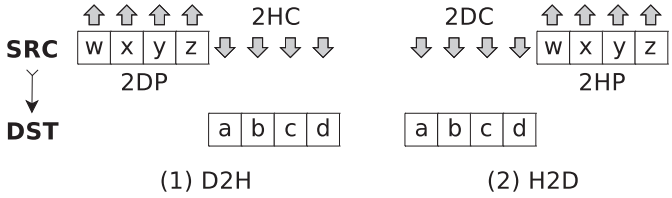


Figure 5. Distributed routing without match.

DOI: 10.1002/cpe

According to [41], the time complexity of *DROCR* in the ideal routing environment can be computed as follows:

$$\mathcal{T}(DROCR) = d_{SRC} + d_{DST} - 2m^{\S}$$

2.3. Maintenance operations

HD Tree is built over a complete tree structure. In a very large-scale distributed environment, nodes join or leave the system constantly. *HD Tree* must maintain a complete tree structure at all times. A new node needs to find an empty leaf entry at the bottom of the *HD Tree* before it can join the system, and an existing node must take a proper leaf node as the replacement before it departs from the system. *HD Tree* uses *Entry Table* to track the status of all leaf entries at each different hops time. The *Entry Table* is a binary table maintained at each routing node [¶], which is updated in a progressive manner after each Join or Leave operation.

In a full *HD Tree* of height h, directing a Join or Leave request from any node to a node at depth $h-1$ is $h-1$ hops time operation at worst, and finding the proper entry for the Join or Leave operation based on the *Entry Table* is a $2h$ hops time operation at worst. Therefore, the Join or Leave operation in *HD Tree* is based on maintaining a binary entry table costs $3h - 1$ hops time at worst, and it is in $O(lg(n))$.

Fact 1
In a full HD Tree of height h, the worst case time complexity of Join or Leave operation is $\mathcal{T}_{worst}(Join/Leave) < 3h$.

3. ERROR RESILIENT ROUTING

HD Tree doubles the number of neighbors in its equivalent tree[‖] structure at the cost of doubling the total number of links of a tree. As a consequence, routing the routing in *HD Tree* could be designed to be highly error resilient. In the case of the failure of a neighboring node, a very simple solution would be to redirect the routing request to a functioning neighbor and to start the routing process over again, which would require another $2h$ hops time at worst. However, by comparing the *match* between functioning neighbors and the destination node and by rearranging the sequence of hierarchical operations and distributed operations, better performance might be achieved.

3.1. Connecting SPeers

In a k-ary *HD Tree*, the root has k neighbors, all internal nodes except *SPeers* have $2k + 2$ neighbors, and *SPeers* have $2k$ neighbors. All leaf nodes have two parent nodes except *SPeers,* which are isolated from the system in the event of the failure of a single parent. However, this problem can be resolved by introducing extra links at each depth to connect the *SPeer* parent to an *SPeer* child of another *SPeer* of the same depth. There are many ways to conduct these extra connections. One tactic we employed for error resilient routing was to connect the *i-th SPeer* to the *SPeer* child of *(i+1)-th SPeer* at the same depth (as shown in Figure 6). This improvement increases the total links in a k-ary *HD Tree* from $2(n-1) - kh$ to $2(n-1) - k$.

3.2. Error resilient routing

In the error-prone routing environment, whenever a normal routing operation fails, the *DROCR* algorithm switches the normal routing mode into the by-passing mode. In the by-passing mode, all possible routing operations are attempted in order to sidestep neighboring nodes that have failed.

[§] d_{SRC}: depth of the source node; d_{DST}: depth of the destination node; m: length of the match.
[¶] Internal nodes.
[‖] The equivalent tree of an *HD Tree* is the complete tree over which this *HD Tree* is constructed.

DOI: 10.1002/cpe

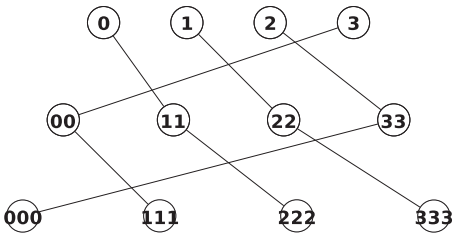


Figure 6. Connecting SPeers.

If there is no *match* between the current node and the destination, procedure '*bp_none_match()*' is called:

(1) if $d_{DST} \leqslant d_{SRC}$
  (a) if *2HP* succeed, return success;
  (b) if *2DP* succeed, return success;
(2) set $i = 0$;
(3) if *2HC[i]* [**] succeed, return success;
(4) set $j = d_{DST} - i - 1$;
(5) if *2DC[j]* [††] succeed, return success;
(6) if $i < d_{DST}$
  (a) *i++;*
  (b) go back to 2;
(7) return *bp_any()*;

In this procedure, if *2HP* and *2DP* cannot proceed, '*bp_none_match()*' tries each digit sequentially from the left-most side and the right-most side of the destination code in order to generate a *match* by using either *2HC* or *2DC* operations.

If there exists at least one *match* between the current node and the destination, procedure '*bp_match()*' is called. Let i and j ($i < j$) be the location index of the current *match* in the destination code:

(1) if *2HP* steps > 0 [‡‡]
  (a) if *2HP* succeed, return success;
(2) if *2DP* steps > 0
  (a) if *2DP* succeed, return success;
(3) if *2HP* steps == 0 && *2HC* steps > 0 [§§]
  (a) if *2HC[j+1]* succeed, return success;
(4) if *2DP* steps == 0 && *2DC* steps > 0
  (a) if *2DC[i-1]* succeed, return success;
(5) if next *match* exists
  (a) set *i, j* to next *match*;
  (b) go back to 1;
(6) return *bp_none_match()*;

In the procedure earlier, two parent nodes are tested in order to by-pass the failure toward the current *match*. If *2HP* and *2DP* can not proceed, *2HC* and *2DC* are tried in order to extend the current *match* toward the destination code. If the by-passing operations fail in the current *match*, the next

[**] *Send to an HChild whose sequence number equals to the i-th digit of the destination code.*
[††] *Send to a DChild whose sequence number equals to the j-th digit of the destination code.*
[‡‡] *2HP operation will not truncate the current match (as shown in Figure3).*
[§§] *2HC operation will not break the current match (as shown in Figure4).*

DOI: 10.1002/cpe

match is tried. If by-passing operations fail in all *match*es that can be found, '`bp_none_match()`' is tried at last.

'`bp_none_match()`' and '`bp_match()`' are intended to generate a new *match*, and to maintain or even extend an existing *match*, even if the normal routing operation has been interrupted by failures. However, better performance might be achieved when compared with a random by-passing, but these operations cannot guarantee the success of by-passing all failures. This is because the next node is determined by a digit in the destination node instead of through trying all functioning neighbors. Therefore, the procedure '`bp_any()`' is called when both '`bp_none_match()`' and '`bp_match()`' have failed.

```
(1) if HParent not visited
   (a) if 2HP succeed, return success;
(2) if DParent not visited
   (a) if 2DP succeed, return success;
(3) set i = 0;
(4) if i-th HChild not visited
   (a) if 2HC(i) ¶¶ succeed, return success;
(5) if i-th DChild not visited
   (a) if 2DC(i) ||| succeed, return success;
(6) if i < k
   (a) i++;
   (b) go back to 3;
(7) return failure;
```

As the *HD Tree* is a cyclic graph, '`bp_any()`' avoids looping by selecting the next node that has not yet been visited. '`bp_any()`' fails if by-passing operations cannot proceed after all previously nonvisited neighbors have been attempted.

3.3. Send to SPeer DParent and send to SPeer DChild

System failure occurs in *HD Tree* if a functioning node exists that has been totally isolated from the system. If there is no *system failure* in an *HD Tree*, the by-passing mechanism is supposed to get around failures and eventually reach the destination. However, '`bp_any()`' guarantees a successful return from the by-passing mode only if the current node is not an *SPeer*.

An *SPeer* node has the same *SPeer DParent* as its *SPeer HParent*, and it also has the same *SPeer DChild* as its *SPeer HChild*. In the routing operations we have discussed in previous sections, *2HP* and *2DP* pass the routing request to the same parent node if the current node is *SPeer*, so do *2HC* and *2DC* if *SPeer* child node is chosen as the next node.

In the by-passing mode, when the current node is a *SPeer* leaf node, by-passing operations cannot proceed if *HParent* fails. On the other hand, when an *SPeer* leaf node is the destination node, no routing operations can be applied to by-pass the failure of the *HParent*.

2SDP (send to *SPeer DParent*) and *2SDC* (send to *SPeer DChild*) are two additional routing operations employed to deal with this situation. *2SDP* forwards the routing request to another *SPeer* parent, called *SDP* (*SPeer DParent*), via an extra link shown in Figure 6. Conversely, *2SDC* forwards the routing request to another *SPeer* child, called *SDC* (*SPeer DChild*), via an extra link also shown in Figure 6. Figure 7 presents a comparison of these routing operations at an *SPeer* node.

We describe a complete by-passing mechanism '`by_pass()`' as follows:

```
(1) set SDC_DEST to Null;
(2) set DEST to destination;
```

¶¶ *Send to the i-th HChild.*
||| *Send to the i-th DChild.*

DOI: 10.1002/cpe

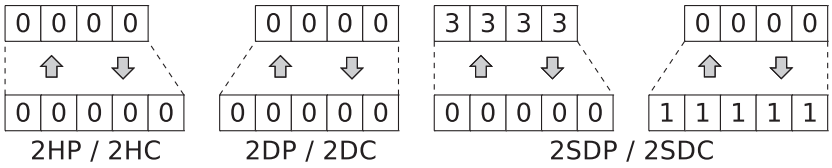


Figure 7. Comparison of SPeer's routing operations.

```
(3) if DEST is SPeer leaf
   (a) if HParent of DEST fails
      (i) set SDC_DEST to DEST;
     (ii) set DEST to SDP of DEST;
(4) if SDC_DEST not null
   (a) if current node == DEST
      (i) set DEST to SDC_DEST;
     (ii) set SDC_DEST to null;
    (iii) 2SDC, return success;
(5) if match not found
   (a) if bp_none_match() == success
      return success;
(6) if match found
   (a) if bp_match() == success
      return success;
(7) if sender exists
   (a) send back to sender, return success;
(8) if 2SDC succeed, return success;
(9) 2SDP;
(10) return success;
```

In this simulation, we only considered the failures of routing nodes. We have tested this simple by-passing mechanism in a 2-ary, 4-ary, and 8-ary *HD Tree*. The maximum failure rate that can be handled so far is about 10 percent of routing nodes' failures.

4. SUPPORTING MULTIDIMENSIONAL RANGE QUERY

The purpose of range queries is to find the minimum complete set of data nodes[***] whose data covers the range data requested. The details concerning how a range query can be carried out are determined by the recursive decomposition scheme employed at the higher layer of the *HD Tree*, the multicast scheme used to forward the query to all relevant data nodes, and the routing mechanism used to forward the range data back to the initiator of the query.

4.1. Data space partitioning and mapping

Figure 8 presents an abstract view of the *HD Tree* overlay system. Application data are partitioned and distributed among peers in *HD Tree* by using the recursive decomposition scheme. Figure 9 depicts an example showing the partitioning of a two-dimensional data space using the Z-order space-filling curve. The first two orders of this process have been directly mapped into a 4-ary *HD Tree* of height 2. The code of each node in this *HD Tree* corresponds to the data range in a certain order of the partitioning process.

[***]Leaf nodes

DOI: 10.1002/cpe

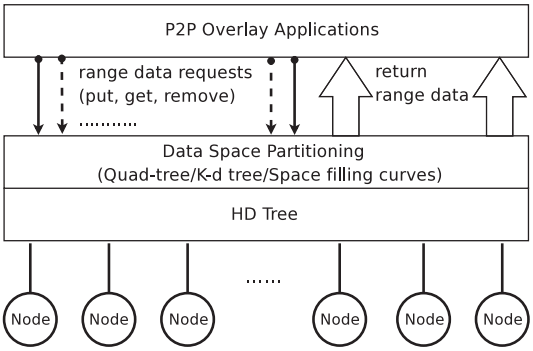
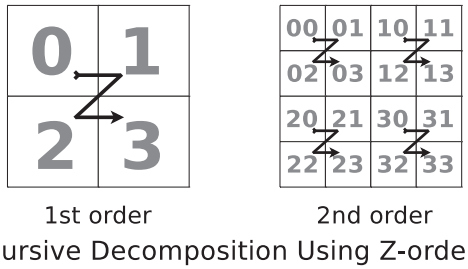


Figure 8. System overview.



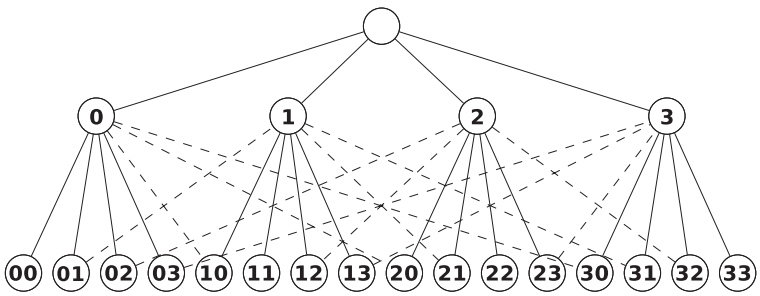
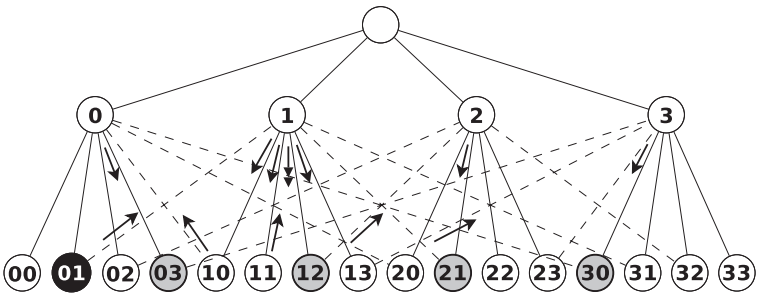


Figure 9. Direct mapping from data space to identifier space.



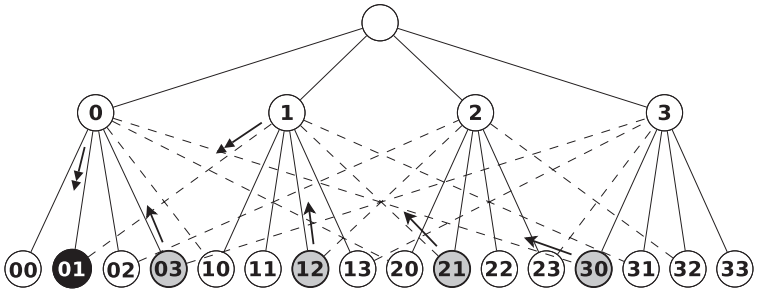
Initiator: 01; Range data requested: 03, 12, 21, 30

Figure 10. D2H query in Hierarchically Distributed Tree.

4.2. Multidimensional range queries

We use Figures 10 and 11 to show the complete process of a range query in *HD Tree*. The range data requested in this example are 03, 12, 21, and 30 (as shown in Figure 9). The initiator of this query is node 01. One such standard query process is called *D2H* query, which always selects some

DOI: 10.1002/cpe



Senders: 03, 12, 21, 30; Receiver: 01;

Figure 11. Response to the range query.

HSiblings via *DParent* in order to forward the query. The process is described as follows:

- the initiator (node 01) begins the query process and sends the request to *DParent;*
- *DParent* (node 1) forwards the request to *HChildren* whose last digit represents the first order of the current data range in the query (0, 1, 2, and 3 in this case);
- *HChildren* (nodes 10, 11, 12, and 13) pass requests to each of their *DParent*s;
- *DParents* (nodes 0, 1, 2, and 3) continue forwarding requests to *HChildren* whose last two digits represent the second order of the current data range in the query (03, 12, 21, and 30 in this case);
- the range data is found at each of the *HChild* (nodes 03, 12, 21, and 30).

The response to this query is a point-2-point routing process. In Figure 11, the requested range data is returned to the initiator node from each data node.

5. DYNAMIC LOAD BALANCING

Figure 9 demonstrates direct mapping from two-dimensional data space to *HD Tree* identifier space, using a Z-order space-filling curve. Each order of recursive decomposition corresponds to nodes at the depth of that order. The data distribution load at each data node is uniquely identified directly by the node ID. This figure also indicates that data space is evenly partitioned and uniformly distributed into identifier space. Recursive decomposition partitions data space and maps it into identifier space. It stops when the recursive order reaches the height of the full *HD Tree*. Each data node manages the data distribution load at this order, referred to as the *basic load*.

However, in a real P2P environment, some nodes might have a lower capacity for participating in system-wide operations. On the other hand, some data might be accessed more frequently and become hot spot areas in data space. All these factors may lead to a bottleneck or some heavily loaded nodes in the *HD Tree*.

5.1. H2H and H2D local adaptation

H2H and *H2D local adaptation* are local load balancing schemes affecting only sibling nodes and *HParent*. The overloaded node re-partitions current load and redistributes it to *HSiblings* or *DSiblings* via *HParent* (as shown in Figure 12). This load adaptation scheme is a 2-hop time operation ($H2H$ or $H2D$), and it is a local load balancing scheme because it disseminates load to nodes close to the hot spot area.

H2H and *H2D local adaptation* do not change the direct mapping feature in *HD Tree*. However, they alternate the mapping from data space to identifier space locally and temporally. The *basic load* at the heavily loaded node is partitioned into k smaller parts of load. Each part loses the lowest digit on the right but obtains one digit from 0 to $k-1$ on the right-most (*H2H*) or left-most (*H2D*) side. This transition process is also depicted in Figure 12. This local load adaptation scheme might fail if

DOI: 10.1002/cpe

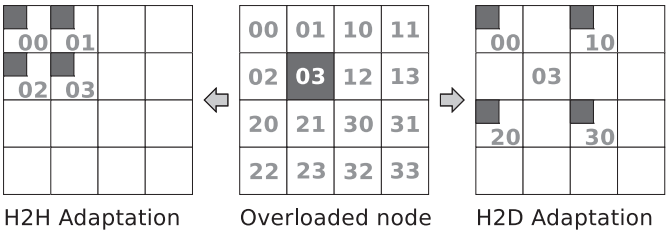


Figure 12. H2H and H2D local adaptation.

other siblings are also heavily loaded. However, it does not generate any additional routing costs to *D2H* range queries because all *data nodes* must be accessed via *HParent* in *D2H* queries.

5.2. D2H repartitioning

The local adaptation scheme provides a temporary yet simple solution for load balancing. However, it is not able to manage the situation when a big hot spot area spreads across many nodes in *HD Tree*. We propose another system-wide solution, *D2H repartitioning*, which is based on the recursive decomposition scheme employed at a higher layer. *D2H repartitioning* is a global load balancing scheme. It continues the recursive decomposition process, and it re-partitions and redistributes the *basic load* at each data node. *D2H repartitioning* is a dynamic load balancing scheme that balances the load at each data node system-wide in a dynamic manner. In a full *HD Tree* of height h, the order of *D2H repartitioning* can be adjusted between 0 and h depending on the capacity of the node and hot spot areas in data space.

In order 0, the *basic load* is managed at each data node. The overloaded node initiates the next order of *D2H repartitioning*. This load balancing process re-partitions the load in the current order and redistributes it to *HSiblings* via *DParent*. All other nodes start the next order, *D2H repartitioning* process, once they receive the redistributed load in the next order from *HParent*. *D2H repartitioning* will stabilize at the next order within $2h$ hops time; after which, all data nodes will have had their load in the current order repartitioned and redistributed to *HSiblings*, which are 2 hops away via *DParent*. The global load balancing process stops at this order if no overloaded node can be found. Otherwise, the same process will be initiated again, but it will continue at another higher order.

Through this method, the load at each data node is repartitioned and redistributed. This, in turn, recombines the load in the next order system-wide at each data node. This process can be repeated at an even finer grade as the order of *D2H repartitioning* goes higher. Therefore, the hot spot areas are most likely to be dissolved among all data nodes in the system. Figure 13 depicts this *D2H repartitioning* process at the overloaded node 03.

D2H repartitioning alternates the mapping from data space to identifier space system-wide. In each order of the *D2H repartitioning* process, the load identified by node ID in the previous order is partitioned into k smaller parts of load. Each part loses the highest digit on the left (*2DP*) but obtains one lowest digit from 0 to $k-1$ on the right (*2HC*). This transition process is also shown in Figure 13.

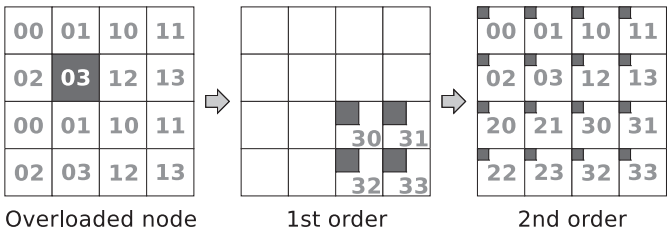


Figure 13. D2H load repartitioning.

DOI: 10.1002/cpe

D2H repartitioning does not change the direct mapping feature in *HD Tree*, because the new order of repartitioning becomes global knowledge after $2h$ hops stabilization time. However, the involvement of data nodes caused by a range query expands exponentially: the same range query involving only one data node at order 0 will become a range query involving k^j data nodes at the order j, and it will become the range query involving the entire system at the order h. This is the main trade-off between load balancing and the performance of range queries.

6. EXPERIMENTAL RESULTS

We built an *HD Tree* P2P overlay structure in the network simulator 2 (ns2). The reason we chose ns2 as the implementation platform was to facilitate our future study of different scenarios that might be applied to the lower layer network infrastructure, the higher layer data space partitioning and the P2P overlay application layer.

6.1. Simulation environment

Currently, the *HD Tree* overlay is built over the traditional wired network infrastructure. At the P2P overlay application layer, we simplify the multidimensional data space in real application scenarios to an abstract data space with [0, 1] range at each dimension. At the data space decomposition and mapping layer, we employed a Z-order space-filling curve. An *HChild* at depth d holds a certain part of the abstract data space, which was previously decomposed by its *HParent*, half at each dimension according to the sequence determined by the Z-order space-filling curve. Therefore, the *HCode* of a node indicates which part of the abstract data it is currently managing (as shown in Figure 9). In order to support multidimensional range queries in the error-prone routing environment, error resilient routing has been incorporated into the *DROCR* algorithm.

We have tested *HD Tree* structure in both ideal and error-prone distributed routing environments. In an ideal distributed routing environment, the maximum network size we tested for basic routing operations includes a 2-ary *HD Tree* of height 12 (8191 nodes), a 4-ary *HD Tree* of height 6 (5461 nodes), and a 8-ary *HD Tree* of height 4 (4681 nodes). However, in an error-prone distributed routing environment, the maximum network size we are able to use for the range query is reduced to a 2-ary *HD Tree* of height 9 (1023 nodes), a 4-ary *HD Tree* of height 5 (1365 nodes), and a 8-ary *HD Tree* of height 3 (585 nodes). The maximum dimensionality we tested, in all simulation runs, is up to 6. Each node in *HD Tree* manages a certain number of virtual nodes in order to accommodate the dimensionality changes.

We must note that *HD Tree* itself is highly scalable. These limitations are set according to the capability of our hardware resources, because ns2 is a single process simulator.

6.2. Average routing performance

Simulations for validating the average routing performance are conducted in an ideal routing environment. These include the average routing performance of basic distributed routing and combined routing. The average routing performance is evaluated among all combinations of source and destination pair at the same depth, from 1 to the tree height; it is measured by the fraction of the average number of hops used from the source to the destination node over two times their depth.

$$Avg.\,Routing = \frac{average\,\#\,of\ hops\,used}{2 \times d}$$

In this part of the experiment, we first simulated each of the four distributed routing operations separately (*D2H*, *D2D*, *H2D*, and *H2H*); we also simulated combinations of these four distributed routing operations (*X2X* routing) and the combination of both hierarchical and distributed routing (*DROCR* routing). The results are shown by using different points or line-points in the first and second row of Figure 14; the comparisons are also made by drawing performance lines for each of the equivalent tree routing.

DOI: 10.1002/cpe

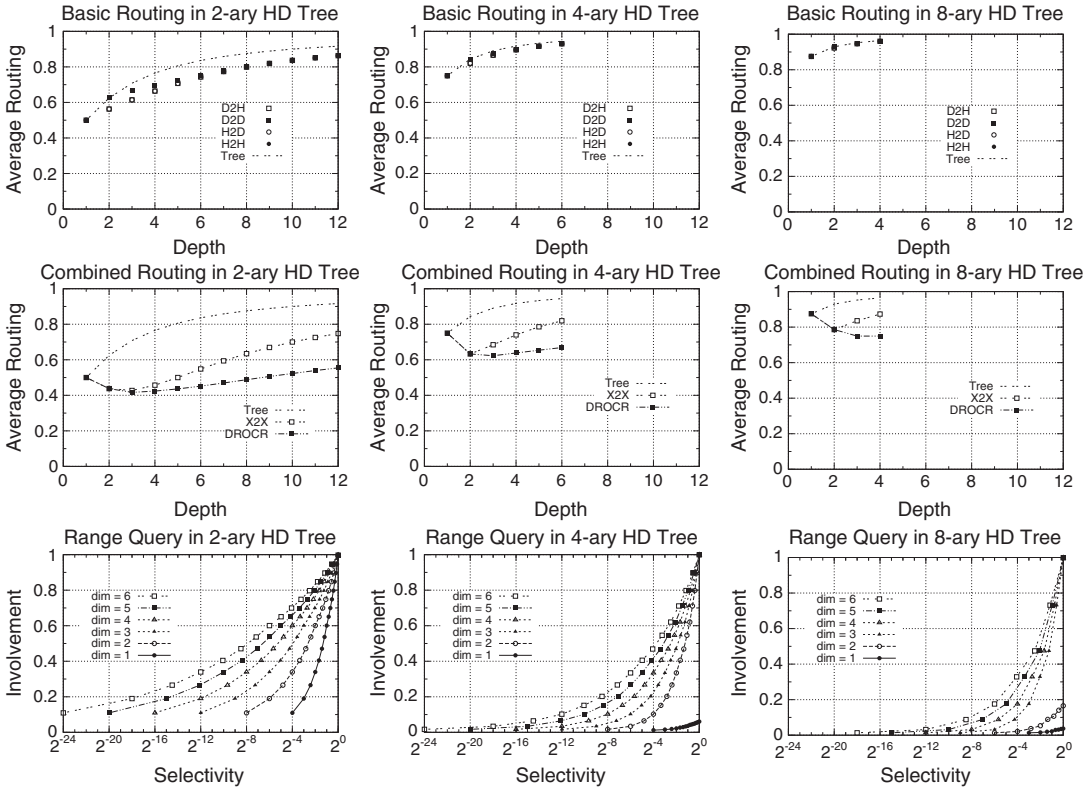


Figure 14. Routing and range query in ideal routing environment.

In *HD Tree,* the average performance of basic distributed routing has an upper bound. This upper bound is the average routing performance of a *SPeer* at the same depth, and it is the same as the performance of the routing in the equivalent tree structure. They are all in $O(log_k n)$ but outperform $2d$ hops time: the worst case of time complexity of distributed routing in *HD Tree*. Our simulation results, shown in the first row of Figure 14, comply with this analysis. They also indicate that each of the four basic distributed routing operations alone achieves slightly better performance than that which is possible in an equivalent tree structure. In addition, *D2H* and *H2D* have the same routing performance, as is also the case for *D2D* and *H2H*; this is because they are the reverse distributed operations of each other.

We also claimed that the average routing performance in *HD Tree* is comparable with all existing structured P2P overlay systems. Although it is true that they all appear to be bound by $O(lgN)$, the significance of this part of the experiment does not depend on this. In *HD Tree*, distributed routing has more routing options by which to reach the same destination than its equivalent tree routing, and all options appear to have comparable routing performance ($O(log_k n)$). Therefore, by combining four distributed routing algorithms together (*X2X* routing), we are able to achieve more performance gain over the equivalent tree routing. This is because the shortest path is more likely to be found by comparing the *match* found in *D2H*, *H2D*, *D2D*, and *H2H* routing all together. Moreover, the *DROCR* algorithm in our previous work, which is the complete combination of both distributed routing and hierarchical routing, achieves the greatest performance gain. This occurs because as long as there is a *match* between the source and destination pair, the shortest path can always be found. The second row of Figure 14 shows that the performance gain of *DROCR* over a tree almost doubles that of *X2X* over a tree at a greater depth.

Simulation results shown in the first two rows of Figure 14 also indicate that the lower ary *HD Tree* achieves better routing performance compared with the higher ary *HD Tree* but less performance gain over the equivalent tree routing. The reason for this is that the lower ary *HD Tree* has a relatively large number of routing nodes, but fewer routing options, when compared with the higher

DOI: 10.1002/cpe

ary *HD Tree*. However, the performance results all gradually increase and approach the performance of the equivalent tree routing as the depth increases step by step. Because the increase in depth, routing between source and destination pairs without any *match* becomes the dominant element in computing the average routing performance. Therefore, a certain performance trade-off might exist between the data distribution load and the routing performance. As we have known, the data distribution load at each leaf node decreases exponentially with the increase in height. Meanwhile, the average routing performance increases much more slowly.

6.3. Performance of range query

Range queries in *HD Tree* are conducted in both ideal and error-prone routing environments. The performance of range queries is evaluated by the involvement against the selectivity (as shown in the third row of Figures 14 and 15), the average routing performance against the involvement (as shown in the first column of Figure 16), and the by-passing rate against the involvement (as shown in the second column of Figure 16). The involvement is measured by a fraction of the total number of nodes involved in the entire query process over the total number of functioning nodes in the system:

$$Involvement = \frac{total\ nodes_{involved}}{total\ nodes_{functioning}}$$

The selectivity is measured by another fraction of the size of the range data in the query over the entire data space:

$$Selectivity = \frac{range\ query\ size}{entire\ data\ space}$$

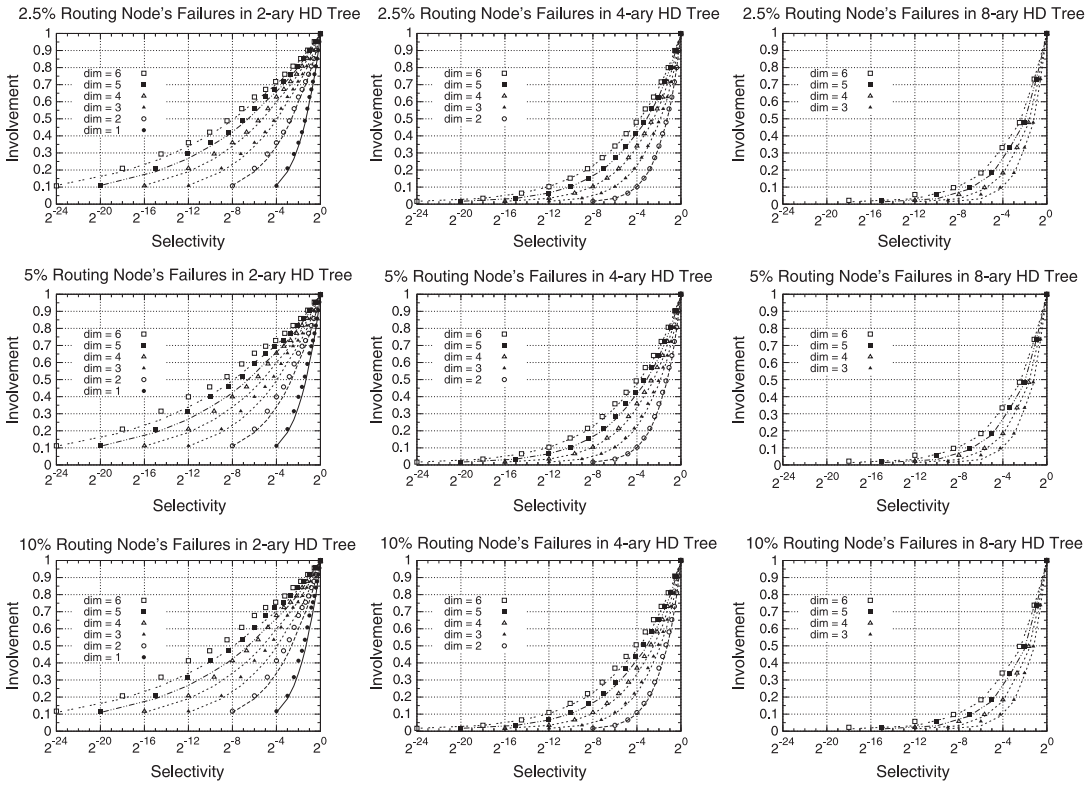


Figure 15. Range query in error-prone routing environment.

DOI: 10.1002/cpe

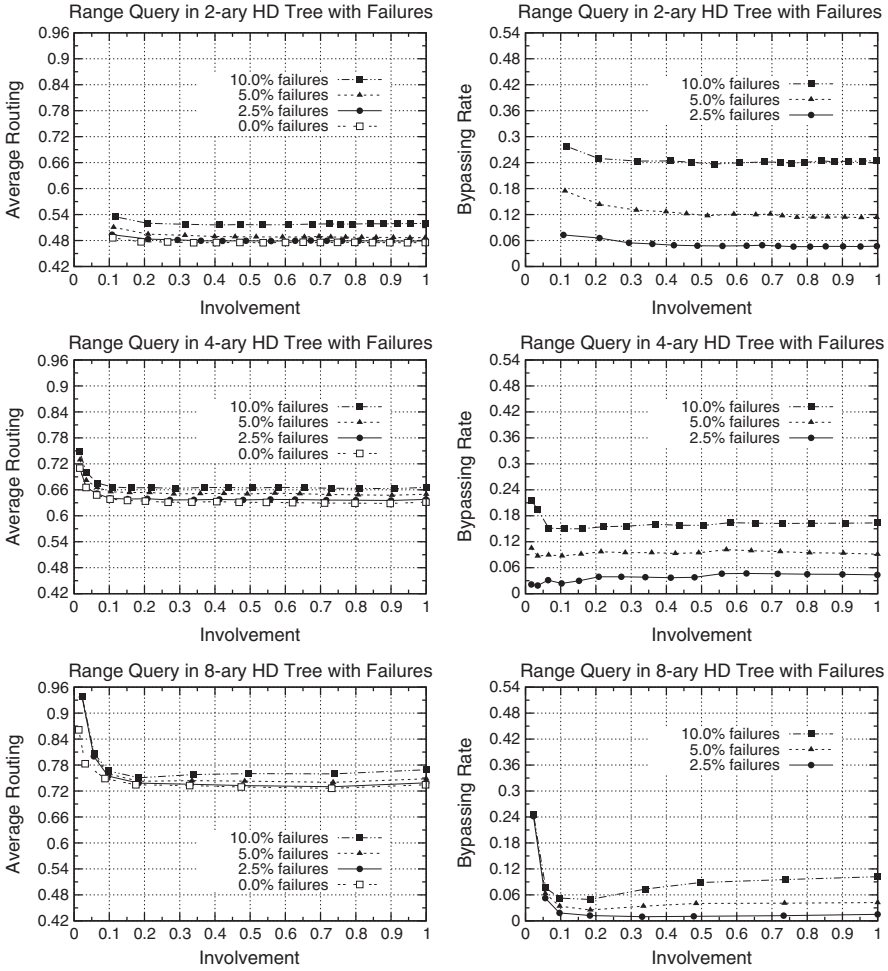


Figure 16. Routing performance of range query in error-prone routing environment.

The by-passing rate indicates the total number of redirected routing caused by failures over the total number of routing conducted for the same range query:

$$\textit{By-passing rate} = \frac{\textit{total by-passing routing}}{\textit{total routing}}$$

In this part of the simulation, we vary the range data in the query from a very small range size to the entire data space. There are a total of 16 steps in the 2-ary and 4-ary *HD Tree*, and 8 steps in the 8-ary *HD Tree*; the range data size increases from $\frac{1}{16}$ to 1 and from $\frac{1}{8}$ to 1 at each dimension correspondingly.

In each step of the simulation, we collected the performance for the same range query initiated by different functioning nodes. Each node must run as the initiator of the same query once. The initiator starts the query and then stops until all range data requested has been received. The average performance for the same range query is evaluated among all functioning nodes in the system.

In order to set up an error-prone routing environment in the simulation, we randomly selected some routing nodes at a depth of 0 to $h - 1$ and set their states to the failure mode. Provided these failures do not cause any *system failure*, any functioning node is supposed to survive the error-prone environment and to fulfill the routing task that it carries. The maximum failure rate we have tested for so far is up to 10 percent of the routing node's failures.

DOI: 10.1002/cpe

The performance is evaluated in four different cases as follows:

(1) The variation in the involvement in a fixed multidimensional data space is evaluated by varying the selectivity of range data from a very small range size to the entire data space. It is indicated by performance lines and points in the third row of Figures 14 and 15.
(2) The comparison of the involvement in different multidimensional data space is made by varying dimensionality from one to six. It is shown by different performance lines and points in the third row of Figures 14 and 15.
(3) The comparison of the involvement among a 2-ary *HD Tree* of height 9 (the first column at the third row of Figure 14 and the first column in Figure 15), a 4-ary *HD Tree* of height 5 (the second column at the third row of Figure 14 and the second column in Figure 15), and a 8-ary *HD Tree* of height 3 (the third column at the third row of Figure 14 and the third column in Figure 15).
(4) The comparison of the involvement in the ideal routing environment (performance lines in Figure 15) and the error-prone routing environment (performance points in Figure 15).
(5) The variation in the average routing and by-passing rate of range queries at a fixed failure rate is produced by varying the involvement of nodes at different levels of selectivity, which is indicated by each performance line and point in Figure 16.
(6) The comparison of the average routing and by-passing rate in range queries at different failure rates is produced by varying the total number of routing nodes' failures, which is indicated by different performance lines and points in Figure 16.

In both ideal and error-prone routing, the experimental results show that the involvement of range queries varies proportionally according to the selectivity within the same data space but deviates considerably with the change of dimensionality (as shown in the third row of Figures 14 and 15). However, the total number of nodes involved in range queries has been effectively minimized. Because *HD Tree* has a natural consistency with the recursive decomposition scheme used for the partitioning of data space, it does not need to maintain an extra data structure to accommodate data locality's changes with exponentially expanding and extending rates.

In the error-prone routing environment, extended simulations for range queries in *HD Tree* with three levels of the routing node's failures are conducted. The experimental results (Figure 15) show that even 10 percent of routing node failures cause no significant variations in the involvement. In fact, it is not even so noticeable in the situation of a 4-ary and a 8-ary *HD Tree* (as shown in the second and third column of Figure 15), because a 4-ary *HD Tree* has doubled the routing options of a 2-ary *HD Tree*, and a 8-ary *HD Tree* has doubled the routing options of the 4-ary.

However, the lower selectivity cases in 8-ary *HD Tree* (as shown in the third column of Figure 15) indicate that a finer designing of the error-resilient *DROCR* algorithm must be considered if a higher percentage of routing node failures needs to be tested. The current routing algorithm is able to make use of at most one *match* in each of the four basic distributed operations in the by-passing routing mode. The performance may be further optimized if all *matches* between the source and the destination pair can be found and properly employed in the by-passing routing mode. In addition, routing operations via extra links connecting *SPeers* (as shown in Figures 6 and 7) should be further explored and merged into the *DROCR* algorithm.

The average routing and by-passing rate of range queries depicted in Figure 16 confirm our analysis in the previous and current subsection. Our analysis can be summarized as follows:

- *The lower ary HD Tree has a better average routing performance.*
- *The higher ary HD Tree has a better fault-tolerant capacity.*
- *The error-resilient routing in a higher ary HD Tree will be further improved if a finer design of by-passing routing can be explored.*

A simple but necessary fact to mention is that a 2-ary *HD Tree* of height 9 has 512 data nodes; these are indexed by 511 routing nodes, and the average data distribution load at each data node is $\frac{1}{512}$. In comparison, a 4-ary *HD Tree* of height 5 has 1024 data nodes. They are indexed by 341

DOI: 10.1002/cpe

routing nodes, and the average data distribution load at each data node is $\frac{1}{1024}$. Additionally, a 8-ary *HD Tree* of height 3 has 512 data nodes. They are indexed by 73 routing nodes. Of importance is the fact that this 8-ary *HD Tree* has the same data distribution load as the 2-ary *HD Tree* of height 9.

6.4. Performance of Join and Leave

Simulations for the Join and Leave operations of nodes are conducted in an ideal routing environment. Join and Leave operations run in a full *HD Tree* of height h. We set up a full 2-ary *HD Tree* of height 10, a full 4-ary *HD Tree* of height 5, and a full 8-ary *HD Tree* of height 3 for these two maintenance operations.

Before each step of the Join or Leave operations, we initialized the simulation environment by randomly setting a number of leaf nodes at $h + 1$ as existing leaves. Join operation joins *HD Tree* as a new leaf node at $h + 1$ until all leaf entries at $h + 1$ are fully filled, whereas Leave operation takes an existing leaf node at $h + 1$ as the replacement until all leaf entries at $h + 1$ are empty. The performance of Join and Leave operations is evaluated against the number of existing leaf nodes at $h + 1$ and the number of empty leaf entries at $h + 1$, respectively, and it is measured by the fraction of the average number of hops used to find the leaf entry over $3h$:

$$Average\ Join/Leave = \frac{average\#of\ hops\ used}{3h}$$

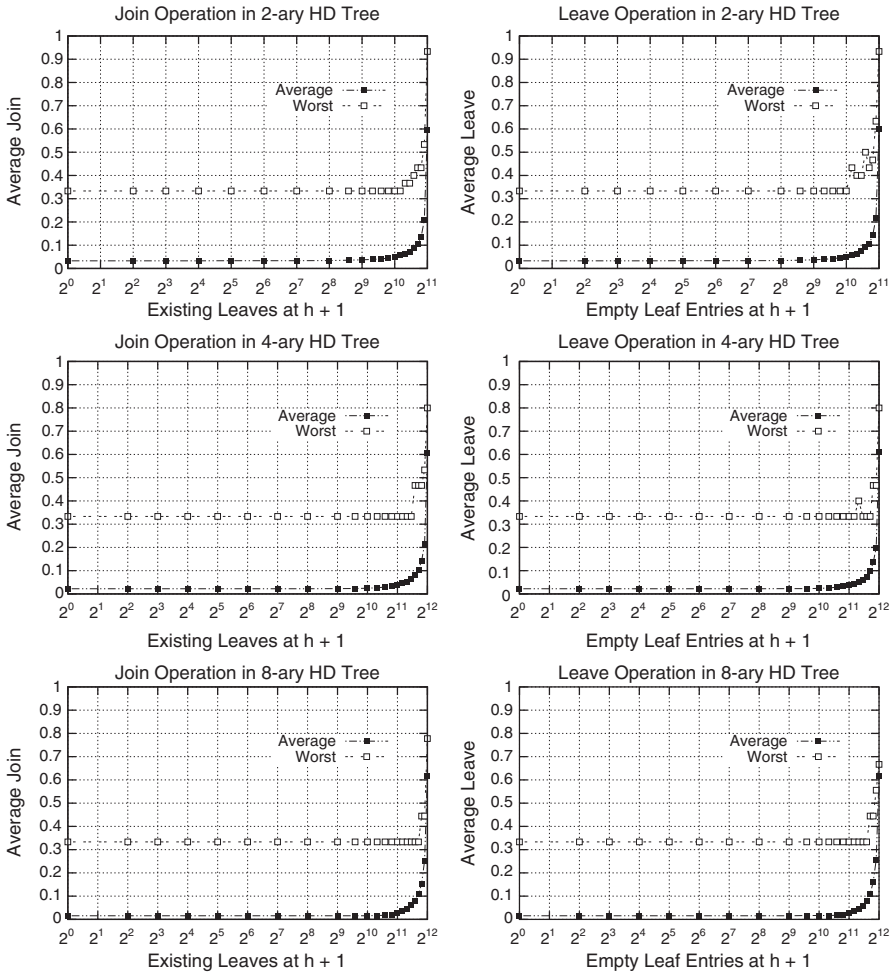


Figure 17. Performance of Join and Leave operations.

DOI: 10.1002/cpe

In each step of the Join or Leave operation, we collected and computed the performance data by initiating the same Join or Leave request from each node at a depth of less than $h + 1$. Figure 17 shows Join and Leave performance in both the average and the worst cases. The performance of Join operation increases with the increase in the number of existing leaves at depth $h + 1$; on the contrary, the performance of Leave operation increases with the increase in the number of empty leaf entries at depth $h + 1$. However, according to Subsection 2.3, the performance of both Join and Leave operations should be less than 1, and the experimental results shown in Figure 17 comply with this analysis very well.

6.5. Performance of D2H repartitioning

The performance of *D2H repartitioning* is examined by both the involvement bar (the second row in Figure 18) and the average routing bar (the second row in Figure 19) for the same range query at each order of *D2H repartitioning*. In order to illustrate the connection between a single range query at different orders of *D2H repartitioning* and range queries at different levels of selectivity, we have set up the same simulation environment in a 2-ary, 4-ary, and 8-ary *HD Tree* for *D2H repartitioning.* This was likewise performed for different range queries in the ideal routing environment. The comparison is made by plotting the involvement (the first row in Figure 18) and the average routing performance (the first row in Figure 19) of range queries against the selectivity. Moreover, in order to show the impact of load balancing on each data node in *HD Tree*, the partitioning bar for *basic load* at the heavily loaded node is compared with the order of *D2H repartitioning* (the third row in both Figures 18 and 19).

In this part of the simulation, the range query that is used to collect the performance data at different orders of *D2H repartitioning* involves only one data node at order 0, and we assume the *basic load* at this data node was a hot spot area in data space. The experimental results in the first two rows of Figure 18 show that *D2H repartitioning* at each order has an equivalent number of range queries at certain levels of selectivity that correspond to it. This indicates that a range query at a higher order of load balancing involves significantly more nodes that participate in the same operation,

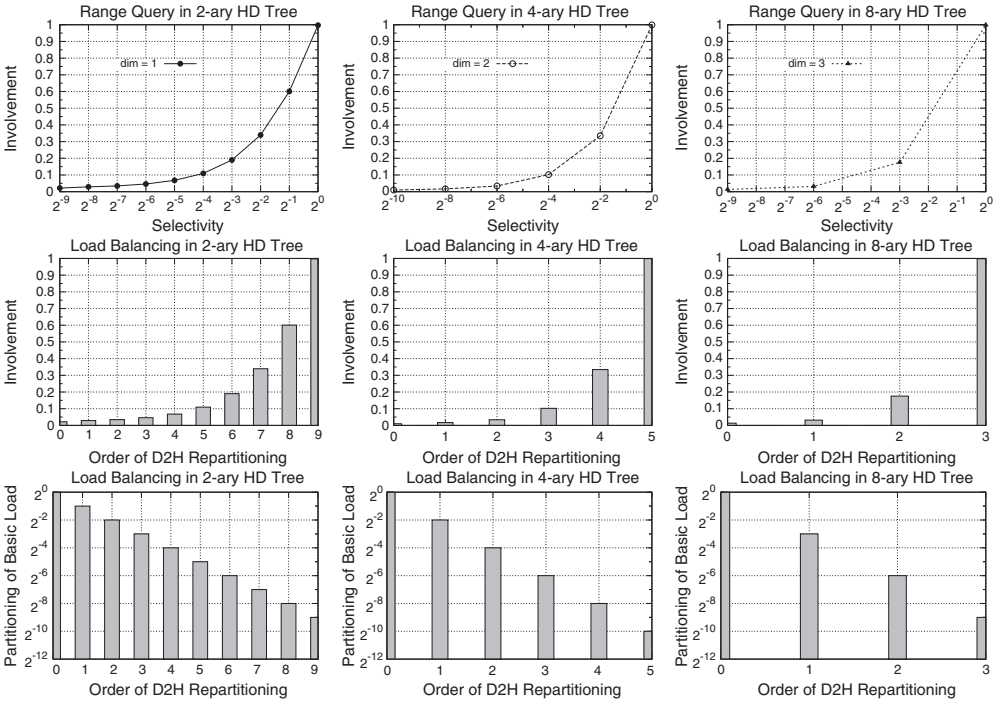


Figure 18. Range query at different order of D2H repartitioning.

DOI: 10.1002/cpe

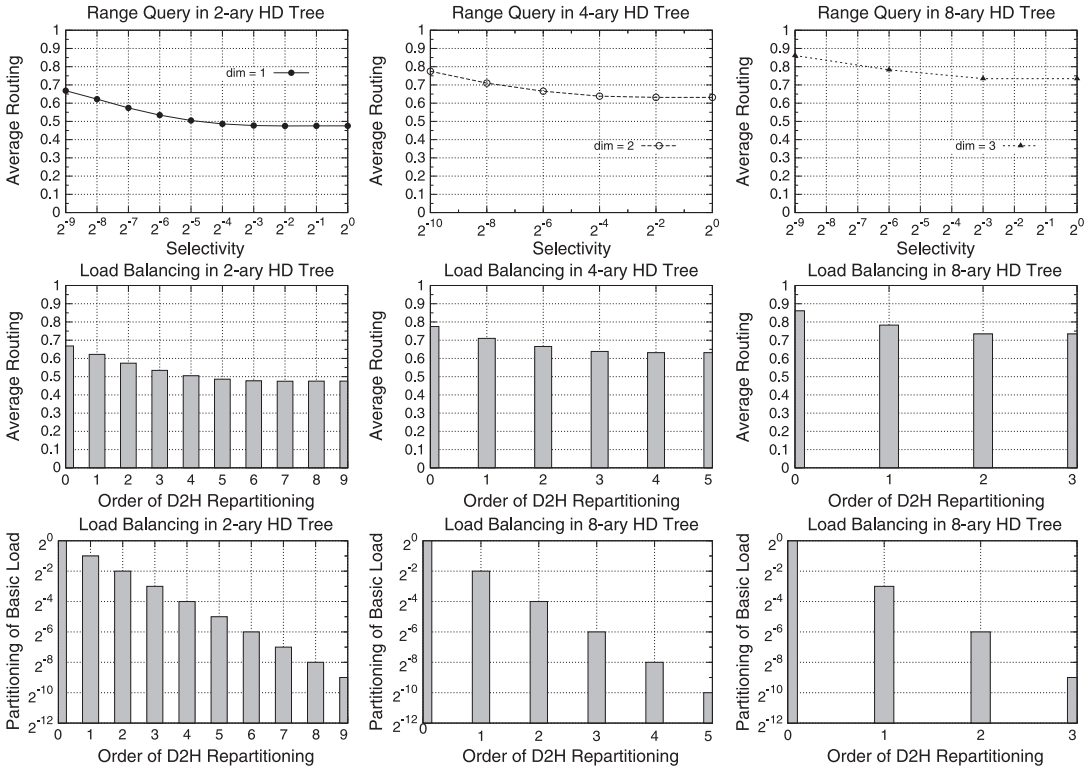


Figure 19. Routing performance at different order of D2H repartitioning.

and the involvement expands exponentially. However, the experimental results in the first two rows of Figure 19 also show that the average routing performance of a range query at different orders does not vary so much. Like range queries at different levels of selectivity, it begins to stabilize at a certain value as the involvement expands.

In both figures, the third row shows the partitioning bar of load at the heavily loaded node in each order. The left-most bar at order 0 is the initial *basic load*. The exponential deduction of this bar is the consequence of an exponential increase in the involvement of data nodes in each order. This indicates that the hot spot area in data space is most likely to be resolved at a certain order of *D2H repartitioning*. All these experimental results comply with our analysis in Section 5, which addresses the main trade-off between load balancing and the involvement of range query.

7. CONCLUSION

The development of *HD Tree* presents another recent effort toward building a robust, reliable, and efficient P2P search network. It provides a data distribution management solution for general purposes, and it supports distributed applications at the architectural level and at the internet scale. *HD Tree* is the first distributed data structure proposed in this area that is able not only to adapt the hierarchical data structure to the distributed environment but also to give a complete view of system states when processing multidimensional range queries at different levels of selectivity and in various error-prone routing environments.

HD Tree is highly scalable because of its embedded tree structure. However, *HD Tree* must maintain a complete tree structure at all times. The experimental results show that the performance of all basic operations, including routing and Join and Leave operations, is bound by $O(lg(n))$ in an ideal routing environment. The maximum random failures we have tested so far comprise about 10 percent of the routing node's failures, with very limited performance variations. Range querying in

DOI: 10.1002/cpe

HD Tree is actually a multicast routing operation that finds the minimum complete set of neighboring data nodes by which the data held covers the data requested. On the other hand, load balancing becomes a recursive decomposition process at another level in which the *basic load* at each data node is repartitioned and mapped back into identifier space again. The *D2H repartitioning* scheme makes load balancing a dynamic and global process in *HD Tree*.

HD Tree provides an optimal solution for the multidimensional range query because *HD Tree* is direct mapping by nature; thus, it supports fault-tolerance and load balancing inherently. *HD Tree* always outperforms the equivalent tree structure. *HD Tree* presents outstanding properties that support multidimensional range queries in the distributed environment. Our next step will be to fully explore the fault-tolerant capacity in a higher ary *HD Tree*; at the same time, we will assign real application scenarios to the new data structure. In our future work, we will continue to adapt this distributed data structure to different network environments and to explore the newly arising potential of both wired and wireless environments.

ACKNOWLEDGEMENTS

This work is partially supported by NSERC, the Canada Research Chairs Program, the Canada Foundation for Innovation Funds, OIT/Ontario Distinguished Researcher Award, and the Ontario Early Career Award.

REFERENCES

1. Karger D, Lehman E, Leighton T, Levine M, Lewin D, Panigrahy R. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. *In ACM Symposium on Theory of Computing*, El Paso, TX, USA, May 04 - 06, 1997; 654–663.
2. Dabek F, Zhao B, Druschel P, Kubiatowicz J, Stoica I. Towards a common API for structured peer-to-peer overlays. *Peer-to-Peer Systems II (IPTPS 2003)*, Berkeley, CA, USA, February 2003; 33–44.
3. Karp B, Ratnasamy S, Rhea S, Shenke S. Spurring adoption of DHTs with OpenHash, a public DHT service. *Peer-to-Peer Systems III (IPTPS 2004)*, Berkeley, CA, USA, February 2004; 195–205.
4. Keong L, Crowcroft J, Pias M, Sharma R, Lim S. A Survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials* 2005:72–93.
5. Ratnasamy S, Francis P, Handley M, Karp R. A scalable content addressable network. *Proceeding ACM SIGCOMM, 2001* 2001; **31**(4):161–172.
6. Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F, Balakrishnan H. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions* 2003; **11**:17–32.
7. Rowstron A, Druschel P. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. *In IFIP/ACM International Conference on Distributed Systems Platforms (Middleware),* (November 2001), Heidelberg, Germany, Nov, 2001; 329–350.
8. Zhao BY, Huang L, Stribling J, Rhea SC, Joseph AD. Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on (IEEE JSAC)* 2004; **22**(1):41–53.
9. Maymounkov P, Mazieres D, Kademlia: a peer-to-peer information system based on the XOR metric. *Proceedings IPTPS*, MA, USA, February 2002; 53–65.
10. Malkhi D, Naor M, Ratajczak D. Viceroy: a scalable and dynamic emulation of the butterfly. *Proceedings ACM PODC 2002*, CA, USA, July 2002; 183–92.
11. (Available from: http://www.gnutellaforums.com.) Gnutella development forum. [Last accessed in January 2010].
12. (Available from: http://rfc-gnutella.sourceforge.net/Proposals/Ultrapeer/Ultrapeer.htm.) Gnutella ultrapeers. [Last accessed in January 2010].
13. Clarke I, Sandberg O, Wiley B, Hong TW. Freenet: a distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science* 2001; **2009**:46–66.
14. (Available from: http://developer.berlios.de/projects/gift-fasttrack.) Fasttrack P2P Technology. [Last accessed in January 2010].
15. (Available from: http://en.wikipedia.org/wiki/Kazaa.) Kazaa from Wiki. [Last accessed in January 2010].
16. (Available from: http://www.kazaa.com.) Kazaa Media Desktop. [Last accessed in January 2010].
17. (Available from: http://en.wikipedia.org/wiki/Fasttrack.) Fasttrack from Wiki. [Last accessed in January 2010].
18. (Available from: http://www.bittorrent.com.) Official BitTorrent website. [Last accessed in January 2010].
19. (Available from: http://www.bittorrent.org.) Official BitTorrent specification. [Last accessed in January 2010].
20. (Available from: http://www.overnet.org.) the overnet file-sharing Network. [Last accessed in January 2010].
21. (Available from: http://en.wikipedia.org/wiki/Overnet.) the overnet from Wiki. [Last accessed in January 2010].
22. (Available from: http://www.edonkey2000.org.) official eDonkey200 website. [Last accessed in January 2010].
23. (Available from: http://en.wikipedia.org/wiki/EDonkey2000.) eDonkey2000 from Wiki. [Last accessed in January 2010].
24. Finkel RA, Bentley JL. Quad Trees: a data structure for retrieval on composite keys. *Acta Informatica* 1974; **4**(1):1–9.

DOI: 10.1002/cpe

25. Hanan S. The quadtree and related hierarchical data structures. *ACM Computing Surveys(CSUR)* 1984; **16**(2): 187–260.
26. (Available from: http://www.itl.nist.gov/div897/sqg/dads/HTML/kdtree.html.)[Last accessed in January 2010].
27. Bentley JL. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 1975; **18**(9):509–517.
28. Morton GM. A computer oriented geodetic data base and a new technique in file sequencing. *Technical Report*, IBM Ltd, Ottawa, Canada, 1966.
29. Orenstein JA, Merrett TH. A class of data structures for associative searching. *In PODS '84: Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, April, 1984, Waterloo, Ontario, Canada, 1984; 181–190.
30. Jagadish HV. Linear clustering of objects with multiple attributes. *In SIGMOD '90: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* 1990; **19**(2):332–342.
31. Harvey N, Jones MB, Saroiu S, Theimer M, Wolman A. SkipNet: a scalable overlay network with practical locality properties. *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems USITS'03*, Seatle, WA, US, March 2003; 9–23.
32. Aberer K, Cudre-Mauroux P, Datta A, Despotovic Z, Hauswirth M, Punceva M, Schmidt R. P-Grid: a self-organizing structured P2P system. *ACM SIGMOD Record* 2003; **32**(3):29–33.
33. Bharambe A, Agrawal M, Seshan S. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM* 2004; **34**:353–366.
34. Cai M, Frank M, Chen J, Szekely P. MAAN: a multiattribute addressable network for grid information services. *In Proceedings of the 4th International Workshop on Grid Computing*, Washington, DC, USA, November 2003; 184–191.
35. Ganesan P, Yang B, Garcia-Molina H. One torus to rule them all: multi-dimensional queries in P2P systems. *In WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, Maison de la Chimie, Paris, France, 2004; 19–24.
36. Shu Y, Ooi BC, Tan K, Zhou A. Supporting multi-dimensional range queries in peer-to-peer systems. *In 5th IEEE International Conference on Peer-to-Peer Computing, 2005*, P2P 2005; 173–180.
37. Zhang C, Krishnamurthy A. *SkipIndex: Towards a Scalable Peer-to-peer Index Service for High Dimensional Data*, Vol. TR-703-04. Princeton University, 2004.
38. Schmidt C, Parashar M. Flexible information discovery in decentralized distributed systems. *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, Seattle, Washington, USA, 2003; 226–235.
39. Schütt T, Schintke F, Reinefeld A. A structured overlay for multi-dimensional range queries. *Euro-Par 2007, Parallel Processing*, Rennes, France, 2007; 503–513.
40. Aspnes J, Shah G. Skip graphs. *In Proceedings SODA*, January, 2003, Baltimore, Maryland, USA, 2003; 384–393.
41. Gu Y, Boukerche A. HD Tree: a novel data structure to support multi-dimensional range query for P2P networks. *Journal of Parallel and Distributed Computing* 2011; **71**:1111–1124.
42. Gu Y. Hierarchically distributed tree. *Ph.D Thesis*, University of Ottawa, Canada. (In preparation).

DOI: 10.1002/cpe