Performance Analysis of Complex Query Processing over DHT-based P2P System

Yi Yi Mar
University of Computer Studies, Yangon
yyimar@gmail.com

Aung Htein Maw
University of Computer Studies, Yangon
ahmaw@ucsy.edu.mm

Abstract

Distributed Hash Table (DHT) is a promising approach for building a large data management platform. But it cannot support complex query processing due to its one-dimensional index structure. Complex query processing is an essential role in resource discovery services, location aware services, and file sharing applications. In this paper, complex query is handled and analyzed by building a multi-dimensional indexing scheme over DHT (MIDHT). It is built with two mechanisms: (1) conversion of one-dimensional key-based DHT to multi-dimensional keys-based DHT and (2) generation keys for the users' desired complex query (KeyGenerator). MIDHT is designed and implemented using PlanetSim simulator. It is analyzed under various complex queries. In addition, the analysis results demonstrates that compared with the state-of-the-art approaches, m-Light and LIGHT, MIDHT can save number of DHT-lookups and bandwidth consumption in range query performance.

Key words: MIDHT, DHT-based P2P system, Multi-dimensional Index over DHT

1. Introduction

DHT-based P2P networks employ a globally consistent protocol to ensure that any node can efficiently route a search to some peer that has the desired file, even if the file is extremely rare [17]. DHT is a layer between network layer and application layer. As shown in Figure 1, an application triggers put and get actions on (key, value) pairs of information. Put (ID, item) operation inserts item with key ID and value item in the DHT. Get (ID) operation returns a pointer to the DHT node responsible for key ID. Each peer also knows a certain number of other peers, called neighbors, and holds a routing table that associates its neighbors' identifiers to the corresponding addresses. Queries are routed since the routing scheme allows one to find a peer responsible for a key in O (log N) routing hops, where N is the number of peers in the network. Although the most notable functionality of hash table is quick exact-match lookups, the core idea of DHT impoverishes the query facility. In fact, the index based on hash table is difficult to support more complex query.

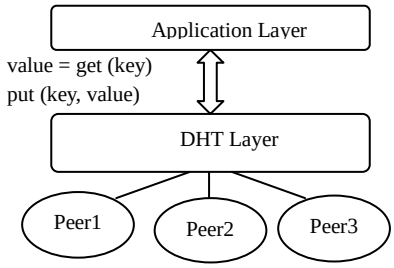

Figure 1. Fundamental functionality provided by DHT

This paper focuses on the processing of various types of complex query by building a multi-dimensional indexing scheme over DHT (MIDHT). KeyGenerator make relevant DHT-lookup operations and so MIDHT only makes lookup operation on the relevant peers. As a consequence, DHT-based P2P system using MIDHT can handle complex query processing by making relevant lookups and can also reduce irrelevant lookups and forwarding steps. The rest of the paper is organized as follows. In Section 2, the existing indexing approaches are discussed as related work. In Section 3, types of complex query are discussed. The architecture of DHT-based P2P system using the proposed MIDHT is described in Section 4 and the required steps for development of MIDHT is mentioned in Section 5. In Section 6, the required parameters for simulation setup are described. In this section, the performance of MIDHT is also evaluated under various types of complex query and compared with other approaches (m-Light and LIGHT). This paper summarizes about the proposed indexing scheme in Section 7.

2. Related Work

There are many researches for supporting complex query processing over DHT-based P2P environment.

Prefix Hash Tree, PHT [12] is the first indexing scheme over DHT that enables more sophisticated query. To process range query, PHT proposed two algorithms. The first algorithm resulted in high latency because all leaves are sequentially traversed until the query is completely solved. In second algorithm, it is parallelized and recursively forward the query until the leaf nodes overlapping the query.

When the requested range is small, it may lead over loading. To solve the overhead in PHT, DST [18] fill the internal nodes with data to violate traversing down to leaf node. So it stores keys not only in leaf nodes but also in internal nodes. To process a range query, it is decomposed into a union of minimum node intervals of segment tree. Then the query is solved by the union of keys returned from the corresponding DST nodes. However, it may leads maintenance overhead because keys are replicated over internal nodes and leaves. DAST [2] is built for range query processing. It constructs an arbitrary segment tree and encapsulate the (key, data) pairs with segmentIds. When processing range query, it divides the requested query into the segments as in AST (arbitrary segment tree). And then it retrieves the data related with segmentIds. It can reduce the number of DHT retrievals. But Accuracy of Range (AOR) can drop because the union of segmentIds can also contain the irrelevant segmentIds.

Distributed Kd-tree (DKDT) [5] supports the processing of similarities query with multi-dimensional datasets. This paper proposed that kd-tree is embedded into the DHT's identifier space to form distributed indexing structure with DKDT. To reduce the number of nodes to be visited, it uses a virtual shrinking mechanism. A virtual node shrinking mechanism is also proposed to allow queries to quickly identify nodes that do not have relevant data. It is obvious that tree compression technique is proposed to ensure that the size of tree does not grow with the dimensionality. And it has to face the bottleneck at the root node and resulted in long network delays for query processing and registration. m-Light [15] also used kd-tree to build efficient indexing scheme over underlying DHT. It proposes a new data aware splitting strategy to distribute data on kd-tree. And then it also proposed a new mechanism to map data from kd-tree to peer nodes. It is high efficient in query processing but still has the drawback in bandwidth and latency consuming.

LIGHT [16] which is a query efficient and low maintenance index structure. The data keys are in an unbounded one-dimensional data space. Due to its one-dimensional key space, it can only handle one-dimensional range query and KNN query. It still has the high overhead in bandwidth consumption and number of DHT-lookups.

3. Complex Query

Complex queries are important as voluminous multi-dimensional data are used in applications including geographical data in Geographical Information Systems (GIS), multimedia retrieval in image or video searching, file sharing and Online Analytical Processing (OLAP). Complex query can be classified as multi-attributes query, range query, cover query, and min/max query. ***Multi-attributes*** *query* is a query composed with more than one attribute to discover the desired information. ***Range query*** is a query to find all the keys in a certain range over the underlying P2P network. It can be classified as single-attribute range query and multi-attribute range query [8]. ***Cover query*** is a query to find all the ranges currently in the system cover a given key. ***Min/Max query*** can be defined as to find min value and max value similar to operations of database like query processing.

4. Architecture of DHT-based P2P System using MIDHT

This section mentions the architecture of DHT-based P2P system using the proposed indexing scheme, MIDHT [9]. It is assumed that Chord is built with more than one hundred peers. Each peer owns the three-tier architecture as shown in Figure 2. The first layer is the application layer and it is for the interactions among peers. The second layer is the indexing layer using the proposed indexing scheme MIDHT over DHT which mainly handles the processing of query. In the third layer or storage layer, content data are stored with related multi-dimensional keys in a distributed manner (DHT).

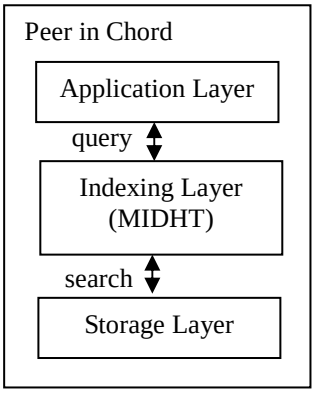

Figure 2. Three tier-architecture for a peer

As shown in Figure 3, when a query is requested to one of the peers in the network, this peer works as an initiator. Before starting the searching process, the initiator computes the multi-dimensional keys for requested query by using *KeyGenerator* of MIDHT. Then the initiator starts to search the data in local storage by using the keys. Here, if the requested data is found in local storage, then it returns the results to the sender. If it cannot find the data in its local storage, then it forwards the keys to the other peers in its routing table (finger table). This forwarding process occurs until the data of the requested query can be retrieved.

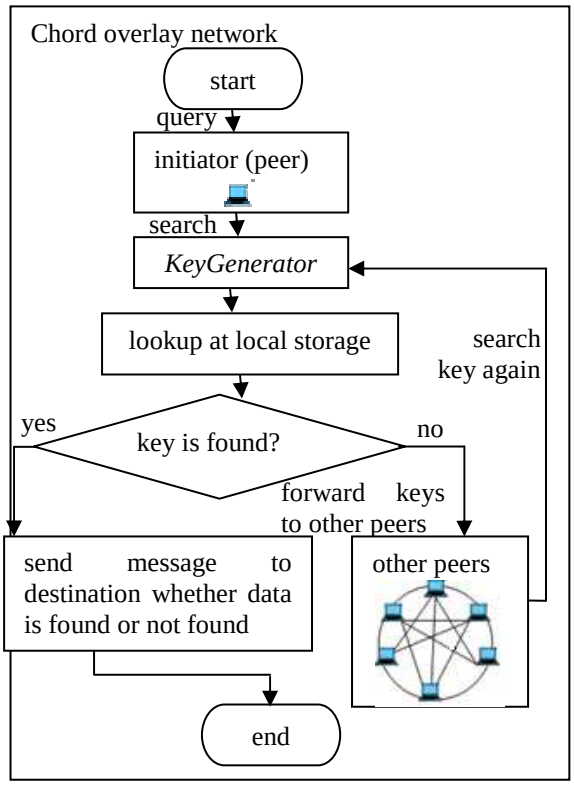

Figure 3. Lookup operation for a query

5. Development of MIDHT

To develop MIDHT, DHT is enhanced to multi-dimensional keys-based and data with multi-dimensional keys are distributed over network. And then *KeyGenerator* is implemented to be used in query processing.

5.1. Converting Keys of DHT from One Dimension to Multi Dimensions

Kd-tree [1] is used to convert keys of DHT from one dimension to multi dimensions. In kd-tree, a record is represented by k-dimensional key. Firstly, the whole dataset needs to be partitioned over kd-tree. Kd-tree can be imbalanced according to the value of splitting threshold, $T_{SP.}$ To build an optimal kd-tree, T_{SP} is important to be optimal. For the purpose of constructing kd-tree to be optimal, T_{SP} is defined to be optimal at "200". For defining the value of T_{SP}, evaluation steps are described in details at [10]. Before staring kd-tree construction, each dimension of the whole dataset is converted into range [0, 1]. When each dimension has converted, kd-tree can be constructed. In this section, an optimal kd-tree is built using Algorithm 1. After constructing kd-tree the half points generated while building tree stored in Tree Info File (T_{IF}) and which is stored in each peer.

Algorithm 1: BuildKdTree (P, T_{SP}, i, n)

Input: dataset 'P' with numeric dimensions {d_1, d_2, ..d_n}, T_{SP} initialized with "200", i: dimension flag, 'n' number of dimensions used in the system

Output: The root of a kd-tree storing P and T_{IF}

1. **if** P contains only T_{SP} records **then**
2. **return** a leaf storing this point
3. **if** i>n **then**
4. i←1
5. **else if** i<1 **then**
6. i←1
7. *V*.label← '#'
8. *V* ←P
9. **else**
10. i←i+1
11. **if** records(*V*) greater than T_{SP} **then**
12. Split P into two subsets with a line *l* through the median d_i dimension of P. Let P_1 be the set of records from the one side of *l* and on *l*, and let P_2 be the set of records from the other side of *l*.
13. T_{IF}← median of d_i
14. j_1←i
15. j_2←i
16. V_{left}.label← *V*.label+'0'
17. V_{right}.label← *V*.label+'1'
18. V_{left}←BUILDKDTREE (P_1, T_{SP} j_1, *n*)
19. V_{right}←BUILDKDTREE (P_2, T_{SP} j_2, *n*)
20. *V*.leftTree←V_{left}
21. *V*.rightTree ← V_{right}
22. **return** *V*

5.2. Distributing Data from Kd-tree to Peers

After partitioning data on kd-tree, multi-dimensional keys of data are being generated Chord [14] is used as a DHT-based overlay network which is a ring-shaped overlay. In Chord, IDs are assigned to both data and peers. Based on the IDs, Chord's algorithm determines which peers are going to be responsible for which data. Chord uses SHA1 [4] to generate ID for peer and data. The data is placed where data ID is closest to the peer ID (i.e., dataID <= peerID). To get peerID, peer's IP is hashed and for dataID, key of data is hashed.

5.3. *KeyGenerator* for Query Processing

Before starting searching process, multi-dimensional data keys are generated for a requested complex query as shown in Figure 4. *KeyGenerator* generates the multi-dimensional keys for request by using T_{IF}.

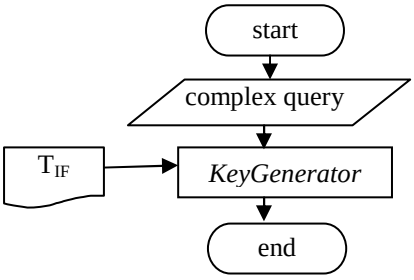

Figure 4. Key generation for requested query

Algorithm 2: *KeyGenerator* (Q, D, T_{IF}, i, n)

Input: Q: request query with multi dimensions and range, D: set of boundary for dimension $\{d_1...d_n\}$ used in the system, T_{IF}: set of half point boundary of nodes in kd-tree for the whole dataset, i: dimension flag, n: number of dimensions used in the system

Output : KeySet: keys for search query

1. DQ$\{d_1,d_2,..d_n\}\leftarrow$ dimensions of Q
2. **for** i=1 **to** n **do**
3. X$\leftarrow$DQ(d_i)
4. $d(X) = \sum_{k=1}^{n} f(x_k)$
5. $nv(X) = d(X) / 10^m$
6. DQ (d_i) $\leftarrow$ nv (X)
7. i $\leftarrow$ i+1
8. SQ$\leftarrow$ DQ
9. BuildLocalTree (D, T_{IF}, i, n)
10. Search (SQ, *V*)

KeyGenerator algorithm as shown in Algorithm 2 requires the parameters Q, T_{IF}, i and n, where Q is the complex query, D is the set of dimensions used in the system, T_{IF} is the list of half points generated from kd-tree, i is the flag for the dimension (last dimension when kd-tree is constructed based on it), and n is the number of dimensions used in the system. Firstly, KeyGenerator converts dimensions of the input complex query into range [0, 1] respectively. And then it builds the local tree using BuildLocalTree (D, TIF, i, n) shown in Algorithm 3.

Algorithm 3: BuildLocalTree (D, T_{IF}, i, n)

Input : D, $T_{IF,}$ i, n

Output: LTree with root *V*: a kd-tree

1. **if all** LTree.*V*.boundary $\in$ T_{IF} **then**
2. **return** LTree
3. **else**
4. **if** i>n **then**
5. i$\leftarrow$1
6. **else if** i<1 **then**
7. i$\leftarrow$1
8. LTree.*V*.label$\leftarrow$ '#'
9. LTree.*V*.boundary $\leftarrow$D
10. **else**
11. i$\leftarrow$i+1
12. **while all** LTree. *V*.boundary $\notin$ T_{IF} **do**
13. split$\leftarrow$median of d_i
14. $l_b\leftarrow$lower bound of d_i
15. $u_b\leftarrow$upper bound of d_i
16. $d_1\leftarrow[l_b$, split]
17. $d_2\leftarrow$[split, u_b]
18. $D_1\leftarrow$D
19. $D_2\leftarrow$D
20. $D_1.d_i\leftarrow d_1$
21. $D_2.d_i\leftarrow d_2$
22. $j_1\leftarrow$i
23. $j_2\leftarrow$i
24. V_{left}.label$\leftarrow$ *V*.label+'0'
25. V_{right}.label$\leftarrow$ *V*.label+'1'
26. $V_{left}\leftarrow$ BuildLocalTree (D1, T_{IF}, j_1, n)
27. $V_{right}\leftarrow$BuildLocalTree (D2, T_{IF}, j_2, n)
28. *V*.leftTree$\leftarrow V_{left}$
29. *V*.rightTree $\leftarrow$ V_{right}
30. ***return*** LTree

Algorithm 3 generates a local tree LTree with root V. LTree is a kd-tree which is starting to build with each dimension in the range [0, 1]. Building of local tree is stopped when all the boundary of the leaves are including in the range of the half points in TIF. Finally, KeyGenerator then starts to search after constructing LTree. This is done by using the algorithm Search (SQ, V) shown in Algorithm 4. In this algorithm, SQ is the searched query and V is the root of LTree generated from the previous step. In this algorithm, search query, SQ, where each dimension in SQ is in range [0, 1]. Firstly Search algorithm checks the leaf nodes of LTree where the point of SQ is covered. KeyGenerator returns the set of leave labels whose boundary can cover SQ as the keys for the requested complex query.

Algorithm 4: Search (SQ, *V*)

Input :SQ:query for searching where dimensions of SQ in numeric form in range [0, 1], *V*: node of LTree

Output: KeySet: all data keys for SQ

1. **if** *V* = $\emptyset$ **then**
2. *V* $\leftarrow$ LTree.root
3. **else if** *V* **is** a leaf **then**
4. **if** SQ $\subset$ *V*.boundary **then**
5. KeySet $\leftarrow$*V*.label
6. **else**
7. **for all** leaves $\in$ LTree **do**
8. *V*$\leftarrow$Ltree.leaf
9. Search (SQ, *V*)
10. **return** KeySet

6. Experimental Setup of MIDHT

For the purpose of performance evaluation, the experimental results have been obtained by implementing MIDHT using PlanetSim simulator [6].

Run times are reported as obtained on a Pentium (R) Dual-Core (2.00 GHz and 2GB) under Windows XP SP2 running Sun Java v.1.6.

In this paper, both real dataset and synthetic dataset are used in the experiments. *North-East* datasets [11] and *DBLP* dataset [3] are used. *North-East* dataset contains 123, 593 postal addresses (points). For *DBLP* dataset, pre-processing steps are required to make it ready to be used. The essential steps are data parsing and data cleaning. Data parsing parses publication records in *XML* data format in each category such as *article*, *inproceeding*, *proceedings*, *book*, *incollection*, *phdthesis*, *masterthesis*, and *www*. Data cleaning detects and eliminates the irrelevant records. In this paper, DBLP is uses with 700, 000 inproceeding records. The statistics of kd-tree construction is shown in Table 1. Simulation steps for creating Chord network in shown in Table 2.

Table 1. Simulation results for kd-tree

DBLP-Dataset Size	Kd-tree Construction		
	Time (sec)	Tree Depth	Number of Data Keys
700, 000	34.031	27	5550

Table 2 Simulation results for building Chord

Simulation Parameters (network size)	Network Creation		Key Insertions (5550 keys)	
	Sim Steps	Time [sec]	Sim Steps	Time [sec]
1000	3448	13.954	14648	58.079

6.1. System Performance under Types of Complex Query

MIDHT is evaluated using various types of complex query to show that it can provide the complex query over DHT-based P2P system. DBLP dataset with 700, 000 inproceeding records is used. Network size or number of peers in the network is 1000. MIDHT is measured using several metrics explored such as the number of lookup hops, lookup message overhead, bandwidth, and latency. Communication over a computer network has the characteristics relating to the overlay network structure. Therefore, MIDHT is implemented in simulation environment and therefore the statistics for metric are assumed to get the nearest results in real world test-bed [7].

For the experiments under various types of complex query, MIDHT mainly focuses on the total number of lookup hop to be less than or within O (log *N*) which is the maximum lookup hop in original DHT where *N* is the number of peers in network.

For the purpose of evaluating the performance of complex queries, ***Q1*** is assumed that *multi-attributes query*, ***Q2*** is *range query*, ***Q3*** is *cover query*, ***Q4*** is *min query*, and ***Q5*** is *max query*.

6.1.1. Lookup Performance of Complex Queries

In these experiments, one DHT-lookup is conducted. ***Q1*** is raised with four shown as follows-

> ***Q1*****:** Retrieve the paper published by author name with "Steve Tappel" title is "Some Algorithm Design Methods." published in book "AAAI", and published in year 1980.

This is four dimensional query using four attributes: author name, paper title, book title, and published year. ***Q2*** is with four attributes (author name, paper title, book title, and published year) and one range (year between 1998 and 2002). It is shown as follows-

> ***Q2*****:** Retrieve the paper published by author name with "Tetsuro Katayama" title is "Generating a device driver with a formal specification language" published in book "Applied Informatics", and published in year between 1998 and 2002.

In this experiment, a query is used with one attribute (author name) to find all papers related with this attribute shown in ***Q3*** as follows-

> ***Q3*****:** Retrieve the papers published by author name with "Matteo Carocci".

In this experiment, ***Q4*** is to retrieve papers published in minimum year and ***Q5*** is to retrieve papers in maximum year. In this multi-dimensional key-based indexing system, the minimum value is with the key "#000....000" and the maximum is with the key "#111…111". Therefore, for this query, when the local tree has been built, *KeyGenerator* generates the keys "#000…000" for min query and "#111…111" for max query.

> **Q4:** Retrieve papers published in minimum year
> **Q5**: Retrieve papers published in maximum year

For each query, the statistics are shown in Table 3. In each experiment, the hop count, message overhead, response time or latency and bandwidth consumption are recorded.

Table 3. Statistics for lookup performance under types of complex query

Query Types	Hop Count	Msg. Overhead (bytes)	Latency (sec)	Bandwidth (bits/sec)
Q1	8	464	0.015	247466.66
Q2	4	232	0.016	116000.0
Q3	6	348	1.922	1448.49
Q4	9	522	0.015	278.4
Q5	9	522	0.016	261

The results in Table3 indicate that MIDHT can handle complex query with the hop count less than O (log *N*). The results in Table 4 are the average

statistics for 30 DHT-lookups. The query is randomly conducted in 30 times and search for different inproceeding records. In these experiments, ***Q1***, ***Q2***, and ***Q3*** are composed with different attributes at each time.

Table 4. Average statistics for 30 DHT-lookups

Query Types	Hop Count	Msg. Overhead (bytes)	Latency (sec)	Bandwidth (bits/sec)
Q1	7.93	460.14	0.007	16732.12
Q2	7	406	0.006	18774.56
Q3	6.13	355.73	6.51	918.79
Q4	9	522	0.015	278.4
Q5	9	522	0.016	261

As shown in Table 4, the evaluation metrics are recorded as average. These average results for 30 times DHT-lookups are almost as much the results of one DHT-lookup. According to the Table 3 and Table 4, it is found that the lookup hop is less than the maximum hop count of original Chord (O (log *N*)).

6.1.2. Lookup Performance under Increasing Network Size

In this experiment, DHT-lookup is conducted 30 times. The query ***Q1***, ***Q2***, and ***Q3*** are randomly generated and average hop count is recorded under varying network size (number of peers in network).

Table 5. Hop count of lookup under varying network size

Query Types	Network Size-256	Network Size -512	Network Size-1024	Network Size-2048
Q1	5.73	6.87	6.7	7.43
Q2	6.2	6.53	7.4	7.6
Q3	5.23	6	6.4	7

As shown in Table5, network size is varied from 256 to 2048. In this experiment, ***Q4*** and ***Q5*** are not considered as they have no impact of increasing network size. According to the results in Table 5, it can be found that the average hop of MIDHT is less than Chord.

6.2 System Analysis

The performance of MIDHT is analysed and compared with the state-of-the-art indexing approaches, m-Light and LIGHT, in range query performance. m-Light and LIGHT evaluate the range query performance in bandwidth cost. The bandwidth cost is captured by the total number of DHT-lookups. In this experiment, MIDHT also use the total number of DHT-lookups (# of DHT-lookups) to be bandwidth consumption in order to compare with these two approaches. To evaluate the performance, m-Light uses NE dataset. LIGHT uses DBLP dataset and then it converts the author name to a floating number in the domain of [0, 1] and used as the data keys. It uses DBLP dataset containing approximately 250, 000 distinct data keys. It divided the whole dataset into five smaller datasets with 50, 000 data keys each. The experiments are conducted against all the five small datasets and the average performance is reported.

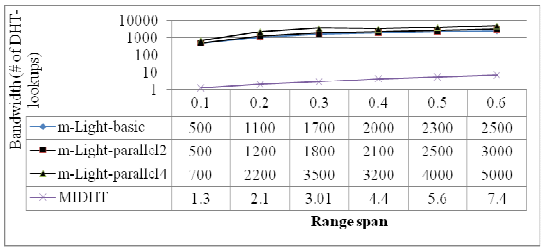

Figure 5. Comparisons of MIDHT and m-Light in range query performance

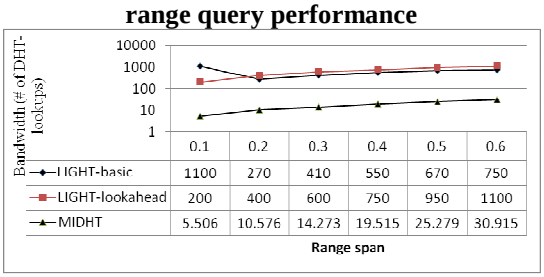

Figure 6. Comparisons of MIDHT and LIGHT in range query performance

Figure 5 shows the bandwidth consumption of MIDHT compared with m-Light under varying range span. While in Figure 6, MIDHT is compared with LIGHT under varying range span. The experimental results in these figures indicate that MIDHT saves 75% in number of DHT-lookups than m-Light and 14% saves in number of DHT-lookups than LIGHT.

7. Conclusion

According to the results in all of the experiments, MIDHT can be considered that it can handle various types of complex query over DHT-based P2P system. It uses total number of lookup hops always less than *O* (*log N*). Moreover, MIDHT can outperform m-Light and LIGHT in range query performance. However, it still needs to handle KNN query which is a type of complex query.

References

[1] J. L. Bentley, "Multi-dimensional Binary Search Trees used for Associative Searching", in the International Journal of Communications of the ACM, Vol. 18, pp. 509-517, September, 1975.

[2] X. Chen and S. A. Jarvis, "Distributed Arbitrary Segment Trees: Providing Efficient Range Query Support over Public DHT Services", the 18th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'07), 2007.

[3] DBLP. http://dblp.uni-trier.de/xml

[4] FIPS. PUBS 180-2 Secure Hash Standard U.S. Department of Commerce/NIST, August 1, 2002.

[5] J Gao and P. Steenkiste, “Efficient Support for Similarity Searches in DHT-based Peer-to-Peer Systems”, in Proceedings of IEEE International Conference on Communications, Glasgow, June 24-28, 2007, pp.1867-1874.

[6] P. Garcia, C. Pairot, R. Monderjar, J. Pujol et al., “PlanetSim: A New Overlay Network Simulation Framework”, in Proceedings of the 4th International Conference on Software Engineering and Middleware, 2005, pp. 123-137.

[7] D. L. Ines, G. Abdelkader, and P. A. Laur, "Facebook Games: The Point where Tribes and Casual Games Meet", in the International Conference Game and Entertainment Technology (GET2010)", Spain, July 26-30, 2010.

[8] D. Li, J. Cao, X. Lu, K. C. C. Chan, B. Wang, J. Su, H. Leong et al., “Delay-Bounded Range Queries in DHT-based Peer-to-Peer Systems”, in Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, 2006, pp. 64.

[9] Y. Y. Mar, A. H. Maw, and K. M. Nwe, “Efficient Indexing Scheme over DHT”, In Proceedings of the 9th International Conference on Computer Applications (ICCA 2011), Yangon, Myanmar, March 2-3, 2011, pp. 145-150.

[10] Y. Y. Mar, A. H. Maw, and K. M. Nwe, “ Usage of Kd-tree in DHT-based Indexing Scheme”, in the 2nd International Conference on Network and Computer Science, Singapore, April 1-2, 2013, pp. 456-460.

[11] North-East dataset. http://www.chorochronos.org/?q=node/60

[12] S. Ramabladran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, “Prefix Hash Tree : An Indexing Data Structure over Distributed Hash Tables”, PODC, 2004

[13] SpatialDataGenerator. http://www.chorochronos.org/?q=node/49

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”, in Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, NY, USA, 2001, pp. 149-160.

[15] Y. Tang, J. Xu, S. Zhou and W. Lee, “m-Light: Indexing Multi-dimensional Data over DHTs”, in Proceedings of the 29th IEEE International Conference on Distributed Computing Systems, Montreal, QC, June 22-26, 2009, pp. 191-198.

[16] Y. Tang, S. Zhou, and J. Xu, “LIGHT- A Query-Efficient Yet Low-Maintenance Indexing Scheme over DHTs”, in the International Journal of IEEE Transactions on Knowledge and Data Engineering, Vol. 22, Issue.1, pp.59-75, January 2010.

[17] V. Vishnumurthy and P. Francis, “A Comparison of Structured and Unstructured P2P Approaches to Heterogeneous Random Peer Selection”, in the Proceedings of the USENIX Annual Technical Conference, Santa Clara, CA, USA, 2007, pp. 309-322.

[18] C. Zheng, G. Shen, S. Li and S. Shenker, “Distributed Segment Tree: Support of range query and cover query over DHT”, in the 5th International Workshop on Peer-to-Peer Systems (IPTPS), February 2006.