DRing: A Layered Scheme for Range Queries over DHTs

Nicolas Hidalgo*, Erika Rosas*, Luciana Arantes*, Olivier Marin*, Pierre Sens* and Xavier Bonnaire†
*Université Pierre et Marie Curie, CNRS
INRIA - REGAL, Paris, France
E-mail: [nicolas.hidalgo, erika.rosas, luciana.arantes, olivier.marin, pierre.sens]@lip6.fr
† Universidad Técnica Federico Santa María, Valparaíso, Chile
E-mail: xavier.bonnaire@inf.utfsm.cl

***Abstract*—Traditional DHT structures provide very poor support for range queries, since uniform hashing destroys data locality. Several schemes have been proposed to overcome this issue, but they fail to combine load balancing, low message overhead, and low latency in search operations.**

In this article we present DRing, an efficient layered solution that directly supports range queries over a ring-like DHT structure. We improve load balancing by using only the nodes that store data, and by updating neighbour information through an optimistic approach. DRing produces low overhead and low latency in environments where queries significantly outnumber data insertion operations. We analyze DRing through simulation and show that our solution does not rely on data distribution.

***Keywords*-Peer-to-Peer, DHT, Information Retrieval, Range Queries.**

I. Introduction

P2P networks are autonomous, self-organised and highly scalable systems with the potential to grow up to millions of nodes. Distributed Hash Tables (DHTs) provide the infrastructure in order to build large scale, decentralised applications over P2P networks. The most desirable properties one expects of a DHT are high data availability, fault tolerance, scalability and load balancing.

DHTs usually achieve load balancing and scalability by using a uniform hash that maps data to a node in a common name-space. They are particularly efficient in exact match queries. P2P overlays, such as Chord [1], Pastry [2] and Kademlia [3], can perform a lookup operation in $O(\log(N))$ hops, where N is the size of the network.

Range queries, on the other hand, are not supported efficiently since the use of uniform hashing destroys data locality. Yet range queries are required in a wide variety of distributed applications like, music or movie storage, P2P persistent games, scientific computation, data mining, and many types of large scale distributed databases. A range query retrieves all the objects with values within a given range. For example: "find all the computers with memory capacity between 1GB and 3GB" or "find all the movies between years 2000 and 2011".

Since the appearance of DHTs, several schemes have been proposed as an attempt to support range queries [4], [5], [6], [7]. Among these, approaches that build an index over the DHT preserve the properties of the underlying overlay. Unfortunately, none of these solutions manage to provide load balancing, low message overhead, and low search latency simultaneously while preserving the scalability of the DHT.

In this paper we present DRing, an efficient approach that aims at improving range query searching in environments where queries significantly outnumber data insertion operations. Our solution supports range queries whilst achieving good load balancing among the nodes, a very low search latency, and low message traffic overhead.

Our main contribution focuses on the search process rather than on the mapping of data onto nodes. The main property of DRing is that it performs independently of the data distribution and can perform range queries in the order of $O(\log(m))$ steps, where m is the number of nodes that store data.

The rest of this paper is organised as follows. Section II briefly describes Chord and the indexing structure over which we build our solution. Section III presents the structure and operations of DRing. In Section IV we present simulation results, discussing cost and performance results. Finally, Section V and Section VI give an overview of some important related work and our concluding remarks respectively.

II. Background

Our solution is based on the the Prefix Hash Tree (PHT) indexing structure [4] and is build on top of the Chord overlay [1]. In this section we briefly describe both systems.

A. Chord

DHT networks are self-organising structured peer-to-peer overlay networks in which any data can be located within a bounded number of routing hops. Systems like Chord [1], Pastry [2] or Tapestry [8] logically organise the nodeId space of nodes into a ring.

Our work uses Chord [1], a ring-like structured DHT. Every node in Chord is assigned with a unique nodeID in a m-bit space using the hash function SHA-1. Each node maintains a successor and a predecessor in the identifier circle. Any key k, in the same name-space as the node's identifiers, is assigned to the first node whose identifier is equals or follows k. In [1], they define as $successor(k)$ the node that is responsible for the key k.

To achieve efficient and scalable key location in the ring, the nodes in Chord store a *finger table*, which is a routing table with at most m entries. The ith entry in the table at node n contains the address of $successor(n + 2^{i-1})$. At each step of the routing process, a node forwards the key to the node that is the closest successor of the key. Using this small amount of information, Chord can route gracefully a key through a sequence of $O(\log(N))$ other nodes towards the destination [1].

Although we describe and implement our approach with Chord [1], the use of any other DHT is straight-forward for ring-like structures like Pastry [2] or Tapestry [8].

B. Indexing Structure

Data in DRing is distributed following the PHT approach [4]. PHT is a binary prefix tree (binary trie) indexing data structure over DHT-based P2P networks. Keys of objects to be indexed are within the domain $\{0,1\}^D$, where D is the length of the string. Notice that such an assumption can be made without loss of generality [4].

The left branch of a node is labeled 0 and the right branch is labeled 1. Each node n of the trie is identified with a chain of P bits (prefix) produced by the concatenation of the labels of all branches in the path from the root to n. PHT builds a prefix tree in which objects are stored at *leaf nodes*. Hence, an object with key k is stored at a leaf node with a label that is a prefix of k.

The trie is completely distributed among the peers in the network. This is achieved by *hashing* the prefix labels of the PHT nodes over the underlying DHT identifier space. As a consequence, each node of the trie will have an assigned node in the DHT. Figure 1 illustrates an example of mapping. We denote *internal nodes* those nodes that belong to the PHT trie which are not leaves. On the other hand, the nodes that do not belong to the PHT trie are denoted *external nodes*.

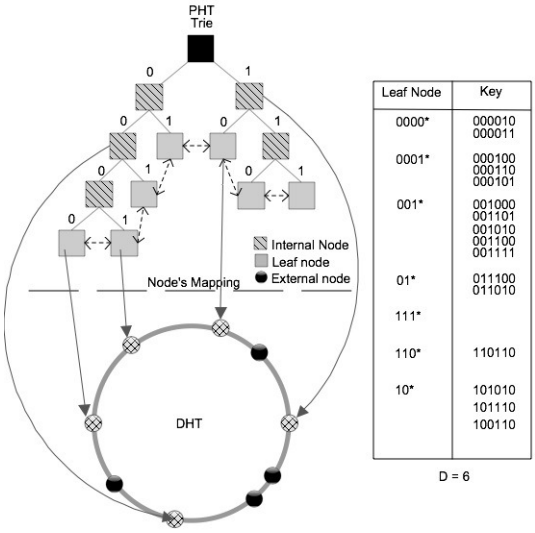


Fig. 1. PHT indexing structure: Mapping trie nodes over the DHT.

Classic searches in PHT are performed following a linear or binary approach. Linear search starts looking for the node that corresponds to the smallest prefix possible of a given key k. Then, until a leaf node is reached, a new DHT lookup operation with a one-bit longer prefix is performed. This search method produces a number of *DHT-lookups* in the order of D. Binary search is a half-interval process that starts by querying a middle prefix of D. If the prefix corresponds to an internal node of the PHT, the search discards the lower half of the interval and continues querying a new middle prefix of the remaining interval. If the prefix corresponds to an external node, the search discards the upper half of the interval. This search method produces a number of *DHT-lookups* in the order of $\log(D)$ and returns the searched leaf node.

In order to improve the performance of range queries, PHT maintains a double list which links all leaf nodes (Threaded leaves), as shown in Figure 1 with dashed lines.

III. DRing: Building a Double Ring

DRing is a two layer structure that effectively supports range queries. Search in DRing exploits previous search information about the nodes that store data. When performing a query, such information is used to contact directly a second ring layer thus reducing search latency.

A. DRing Architecture

Data are uniformly distributed among the nodes of the DHT without an specific order. DRing keeps a second ring structure on top of the overlay which aims at improving performance of range queries. This second overlay, called *Leaf Ring*, comprises only *leaf nodes*, i.e., nodes that store data. The identifier of a node in this second ring is its respective identifier in the prefix trie. Organizing the prefix labels of the leaf nodes into a ring structure improves sequential data search, and therefore, easily provides range queries.

In the *Leaf Ring*, each node references its successor and predecessor leaf in the trie. Figure 2 presents the proposed architecture, where nodes in grey represent leaf nodes.

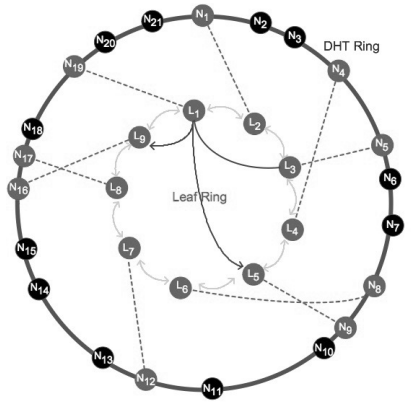


Fig. 2. DRing double structure: *Leaf Ring* over the overlay ring

Additionally, each node of the *Leaf Ring* stores a *Leaf Table* that keeps references to other leaf nodes. Figure 3 shows an

example of a *Leaf Table*. At each entry i, the node stores the reference to a node that is at a *distance* of 2^i. The maximum number of entries of the *Leaf Table* is the maximum object key size D. Each entry is composed by a prefix label of the leaf node, as well as a static reference (IP Address). Maintaining the prefix label helps in dynamic environments, where static references become useless. When a static reference fails, a lookup over the DHT can be performed in order to contact the leaf node by hashing the stored prefix.

A node stores information about past queries it issued which allows to directly access the *Leaf Ring*, avoiding thus internal levels of the trie-structure (see Section III-C). Within the second layer, search is performed as the underlying DHT structure: using a greedy technique that forwards the search to the node which is the numerically closest successor to the key.

LEAF TABLE
Node L

	Distance from L	Prefix	IP Address
1	2^1	0010010	190.168.0.15:99
2	2^2	0110	200.1.19.25:99
⋮			
i	2^i	11101	81.48.20.2:99

Fig. 3. A *Leaf Table* of a node

A node iteratively fills its *Leaf Table* the first time it join the *Leaf Ring*. To complete the entry 2^{i+1} of the *Leaf Table*, a node X asks the node at a distance of 2^i for the ith entry of its *Leaf Table*. For example, to find the node that is at a distance 2^1, X can obtain the information from its successor (since it is at a distance 2^0) and to find the node at a distance 2^5 has to contact the node at a distance 2^4 which provides information about the 4th entry of its table.

The part of our approach described above performs efficiently if the trie is balanced. However, such an assumption is not realistic. Join and leave operations induce a global change in the tables of the nodes in *Leaf Ring*, that grows linearly with the number of nodes. This is the result of taking into account distances instead of static partition of the namespace which, on the other hand, is not efficient since when the trie is unbalanced, the load is also unbalance.

In order to solve this issue and repair the *Leaf Tables* of nodes an optimistic approach is used which consists in only updating tables when a range query is performed (Section III-D). We argue this is enough to achieve a good performance in scenarios where queries highly outnumber data insertion operations.

B. Split Operation

Each split operation in the logical PHT trie, induces two joins and one leave operation in the *Leaf Ring*.

Starting from the root node in the PHT trie, when node X achieves its maximum storage capacity B, the trie splits into two children. The left child will have the prefix label of its parent concatenated with 0 and the right child with 1. The information stored in X is distributed among both children. The left child will be the successor and predecessor of the right one (in the beginning there are only two leaf nodes that store data).

Whenever a node splits, this node, denoted parent, must inform its children about the identifier of its predecessor and successor nodes: the predecessor of the left child is its parent's predecessor and its successor is its right brother; similarly, the predecessor of the right child is its left brother and the successor is its parent's successor. The children *Leaf Tables* are thus initialised with the information provided by their parent node and are further updated in an optimistic way as described in Section III-D.

Figure 4 shows an example of a split operation in the *Leaf Ring*. Node with identifier 01 reaches its storage capacity and is replaced by two other nodes with prefixes 010 and 011. The information of the node with prefix 01 is moved to its children in order to update the successor and predecessor links.

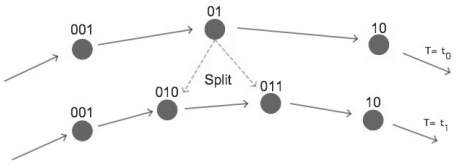


Fig. 4. A split operation in DRing

In case of concurrent splits of neighbor nodes of the *Leaf Ring*, the obtained information that a child gets from its parent might be out of date. Thus if the parent's successor, Y, is no longer in the *Leaf Ring*, the right child will contact the left child of Y. Notice that the prefix label of Y's left child can be obtained by the concatenation of Y's prefix label with '0'; similarly, if the parent's predecessor, Z is no longer in the *Leaf Ring*, the left child will contact Z's right child. The identifier of the latter is composed by the concatenation of Z's prefix label with '1'.

C. Range Query Strategies over DRing

A range query corresponds to an interval of data to be searched. It has the form $R_q = [L, U]$, where L is the lower bound and U is the upper bound. To collect the data in the range, one of the bounds is searched in the network. Then, the search will follow the successor or predecessor links of the *Leaf Ring* in order to retrieve all the data contained in the range query.

Each node in the DHT maintains a *Leaf List*, that contains the leaf nodes that have been previously contacted. These *contact nodes* are the access to the *Leaf Ring* and become the starting point of the search.

The node can start the range query search either by the lower or the upper bound. Such a decision is based on the common prefix with the nodes in its *Leaf List*, i.e., the node starts the search from the bound that has the greatest common prefix.

The size of the *Leaf List* is a parameter λ of the system. Each of its entries store a static reference and the prefix label that identifies the leaf node in the trie (and in the *Leaf Ring*). In the case the access to the first fails, a DHT lookup operation will find the new contact node by applying the hash function over the prefix label of the entry. When the *Leaf List* is full, the replacement strategy removes an entry following a least recently used (LRU) policy.

If a node has never performed any query yet, its *Leaf List* is empty: it knows no *contact node*. In order to fill its *Leaf List*, a node first asks for the *Leaf List* of its neighbors. However, if such an information is not available either, it applies one of the PHT search mechanisms (linear or binary) [4].

Other complex queries are also directly supported by DRing. For instance *Min/max queries* look for the smallest and largest value of the indexed data. DRing carries them out with a single lookup operation. *K-NN queries* return the k nearest data values to a given key. The successor and predecessor links of the *Leaf Ring* allow to find the required data efficiently.

D. Optimistic Table Maintenance

We propose to use the information obtained when a range query is performed to optimistically repair the routing tables, i.e., the entries are updated using the information from neighbors only when a range query takes place.

In the case of Chord [1], the data corresponding to leaf node with prefix A is stored in the $successor(SHA(A))$. Chord will route to the node in the overlay with the identifier which is the successor of the searched key. Since DHT nodes can crash or leave the network, Chord can replicate data on several nodes in the numerical vicinity of $SHA(A)$ so as to avoid information loss. Therefore in dynamic environments, the node responsible for given key can change. When a leaf node leaves the network or crashes, its static reference is no longer valid. In this case, Chord falls back on the new current node whose identifier is the successor of $SHA(A)$ as the holder of the data. Given the prefix label, this new node can easily be contacted through a DHT lookup operation and the static reference to its IP address is updated.

Every split of a leaf node of the trie produces inconsistency in the *Leaf Tables*. Similarly, this information is only updated when a range query takes place. Upon receiving the query, the node looks into its *Leaf Table* and forwards a message to the i reference which is the closest to the searched node, also including its table reference $i+1$ in the message. The node that receives the message checks if this reference corresponds to its own at row i in its table. If such is not the case, it passes on its entry reference to the sender which will then update its table.

Evaluation results (see Section IV) show that this update mechanism provides good performance, producing low overhead, even when the distribution of the data is highly unbalanced.

IV. Simulations & Analysis

In this section we discuss the performance of DRing by comparing it with PHT searches[4]. Our simulations were conducted on top of Peersim simulator [9].

A. Traffic Message Overhead

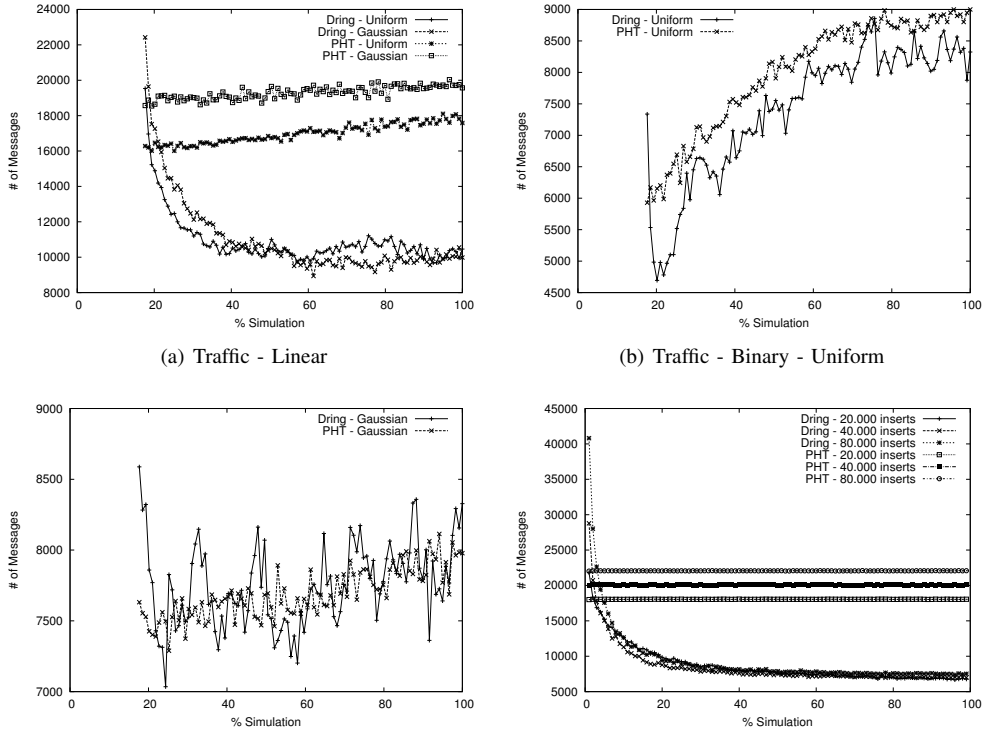
Traffic message overhead is directly related with latency. We measure the number of messages generated by range queries over our system and compare it with the PHT [4] linear and binary search. To this end, we use both the Uniform and Gaussian data distributions which respectively generate a balanced and unbalanced trie-structures. When there is no known contact node in DRing, the default search can be the linear or binary search.

In our experiments, we consider a network composed of $10,000$ peers. Each node stores at most 50 objects. Each simulation starts by inserting $20,000$ objects in the network, which covers the first 20% of the simulation. In the rest of the simulation, the system generates a range query request and an object insertion operation with a probability of 0.9 and 0.1 respectively. The presented results also include the maintenance messages but the latter represent only 0.1% of the total traffic of range queries messages.

Figure 5(a) shows the message traffic when linear search is performed using DRing and PHT. We consider both the Uniform and Gaussian distributions. We can observe that DRing highly decreases the number of messages generated by the search process. In the case of the Uniform distribution, the number of messages is reduced in more than 40% and the improvement reaches near 50% for the Gaussian distribution. In the case of PHT, there is a degradation of performance when the data distribution is Gaussian. Since the latter generates unbalanced tries which store clustered information in the deeper levels of the trie, PHT search suffers from high latencies. On the other hand, DRing directly accesses leaf nodes without going through internal nodes, as PHT search does. As a result, DRing search is more efficient and its performance is independent of data distribution.

Binary search results are presented in Figure 5(b) and 5(c). DRing and PHT perform similarly in the case of binary search. For both distributions the amount of messages is around $8,000$ at each step of simulation, when the Leaf Tables are full. However, we must point out that binary search has a major drawback: if an internal node fails or leaves the system, it might be impossible to locate a given leaf node and the search has to restart generating extra message traffic [4].

Figure 5(d) presents simulation results for different sizes of the trie structure. The generated tries are the results of $20,000$, $40,000$ and $80,000$ insertion operations of data objects in the network. Data insertions make the trie structure grow. The number of leaf nodes that store data is approximately 500, 1000 and 2000 leaf nodes respectively. The results were obtained using linear search and Uniform data distribution. The insert operations were performed before the beginning of the simulation. PHT linear search latency increases with the number of levels of the trie: one more level implies one more message. As the trie is a hierarchical structure, the number of messages will increase in the same amount if we double the



(a) Traffic - Linear

(b) Traffic - Binary - Uniform

(c) Traffic - Binary - Gaussian

(d) Traffic - Data Insertion Operations

Fig. 5. Traffic Comparison

number of insert operations, as is shown in Figure 5(d). In the case of DRing, its structure allows that a query to be routed in $O(\log(m))$ steps, where m is the number of leaf nodes, which greatly improves the results of a linear search. The grow of the trie slightly degrades performance, but which is almost unobservable in Figure 5(d). After approximately 20% of the simulation, the nodes in DRing fill its respective Leaf Tables and the routing process is thus improved.

Maintenance operations in DRing increase with the number of split operations in the trie structure. Figure 6 shows how the number of maintenance messages grows in a simulation when 20%, 15%, and 10% of the total of messages correspond to data insertion operations. DRing performs optimally in an environment where range queries are higher than data insertion operations since such difference makes DRing to fill its tables in an optimistic way. However, if the latter is equal or higher to the former, the nodes in DRing can not update its respective tables and the performance is therefore degraded.

In highly dynamic churn environments, where nodes go in and out of the system frequently, the performance will also be affected. However, DRing uses the indexing trie-structure of PHT [4], which makes easy to find the new node which correspond to a key. With a good replication scheme exploiting the neighbours of every leaf node, the dynamism problem can be mitigated.

B. Load Balancing

One of the main advantages of the DRing approach, is that the traffic is balanced among the leaf nodes of the trie.

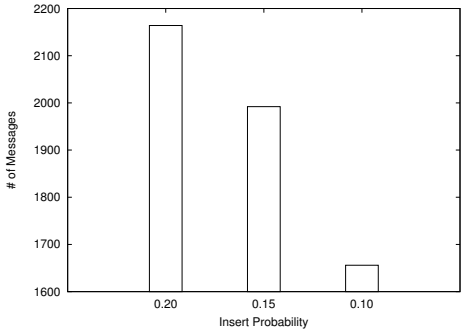


Fig. 6. Maintenance operations in DRing

Generally, search over tree-based approaches traverse part of the trie structure to find data. The upper levels of the trie are highly overloaded, since queries are distributed among a small number of nodes which introduces bottlenecks. DRing diverts queries from the upper levels directly to the leaf nodes, and, therefore, search is performed only at the leaf level. As a consequence, messages are distributed to a greater number of nodes.

A linear PHT search, for example, sends messages to each level of the trie, producing bottlenecks where the number of nodes in the level is small (closer to the root). Notice that at the level e of the trie there is 2^e nodes. The worst case is the first level, where only 2^1 nodes answer queries. On the other hand, at lower levels the messages are distributed

among a higher number of nodes. DRing avoids bottlenecks by distributing these messages at the lowest level.

C. Fault Resilience

In PHT [4] linear and binary searches, nodes are queried about their position in the trie, and each step is used to compute the next one. If a node crashes or leaves the network, the search process will also fail. Moreover, the crash of a node at an upper level of the trie impacts heavily on the search of its sub-trees.

Our approach aims at reducing the traditional hierarchical search of tries, while maintaining the good properties of assigning one prefix label to a node in the DHT. It minimizes access to PHT internal structure as much as possible: most queries are directly diverted to leaf nodes. Internal nodes avoidance not only improves search performances but also reduces the potential impact of faulty nodes.

V. Related Work

Many solutions that support range queries exploiting DHT have been proposed in the literature. They are basically divided into two classes: *Overlay-dependent* and *Over-DHT*.

Overlay-dependent indexing solutions adopt either a DHT-free indexing approach which re-designs its own overlay, or a DHT-modification approach in order to provide data locality. MAAN [10] and Mercury [5] are some examples. Generally they present load balancing and traffic message overhead issues because they do not maintain the DHT properties.

Our approach can be categorized as an *Over-DHT* index, since it builds an index over the DHT. The created index preserves data locality providing support for complex queries. The advantages of these solutions are their portability to any DHT, their easy implementation and the fact that they preserve the DHT properties. We have focused on this class of solutions due to space limitations.

An *over-DHT* solutions generally relies in tree-based indexing structure [11], [7], [4], [6]. Distributed Segment Tree (DST) [11] is a trie-based indexing solution which replicates data over the internal nodes of the trie. To process a range query in DST, the query range is decomposed in several sub-ranges each corresponding to an internal node. Due to data replication, high-levels of the structure can be easily overloaded becoming bottleneck. To address this issue, authors in [7] proposed a Range Search Tree (RST). RST extends DST by introducing a novel data structure called Load Balancing Matrix (LBM) in order to improve load balance among nodes. The main problems of these two solutions are the maintenance cost, the data lost and load balance.

LIGHT [6] is a solution that stores data in all nodes of the indexing structure using a new naming function. It avoids part of the maintenance overhead produced when making a copy of the data in the children nodes. To search over the data, they introduce a new distributed data structure: the leaf bucket. A leaf bucket stores data information and summarises the partition tree structural information. However, load balance in search is still an issue.

VI. Conclusion

In this paper we have presented DRing, a new approach to support range queries in ring-like DHTs systems. DRing builds a second ring overlay with nodes that store the data (leaf nodes). We have also proposed an efficient method to access DRing using the information obtained from past range queries which allows to easily locate a node to start the search over DRing.

Simulation results show that our system outperforms PHT [4] linear search, reducing the traffic of messages in more than 40% in environments where queries are higher than data insertion operations. DRing balances the load of nodes, avoiding bottleneck in the upper levels of the indexing structure.

Exploiting different data distributions, simulation results have confirmed that the performance of our approach is maintained, even in the presence of skewed data.

Range queries is still a major concern when the number of attributes grows significantly. Our future work is to perform simulations indexing multi-attributed data.

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, pp. 17–32, February 2003.

[2] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, ser. Middleware '01. London, UK: Springer-Verlag, 2001, pp. 329–350.

[3] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. London, UK: Springer-Verlag, 2002, pp. 53–65.

[4] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, "Prefix hash tree: An indexing data structure over distributed hash tables," in *In Proceedings of ACM PODC, St. Johns, Canada, July 2004*, 2004.

[5] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," *SIGCOMM Comput. Commun. Rev.*, vol. 34, pp. 353–366, August 2004.

[6] J. X. Yuzhe Tang, Shuigeng Zhou, "Light: A query-efficient yet low-maintenance indexing scheme over dhts," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, pp. 59–75, 2010.

[7] J. Gao and P. Steenkiste, "An adaptive protocol for efficient support of range queries in dht-based systems," in *Proceedings of the 12th IEEE International Conference on Network Protocols*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 239–250.

[8] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.

[9] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris, "The Peersim simulator," `http://peersim.sf.net`.

[10] M. Cai, M. Frank, J. Chen, and P. Szekely, "Maan: A multi-attribute addressable network for grid information services," in *Proceedings of the 4th International Workshop on Grid Computing*, ser. GRID '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 184–.

[11] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed segment tree: Support of range query and cover query over dht," in *IPTPS '06*, 2006.