SFDHT: A DHT Designed for Server Farm

Lichun Li†, Chunhong Zhang‡, Wei Mi†, Yonghui Zhang†, Tao Ma†, Yang Ji†, Xiaofeng Qiu†
School of Information and Communication Engineering
Beijing University of Posts and Telecommunications
Beijing, P.R. China
Email: †{lilichun, miwei1985, yhzhang07, abcdmatao, quxiaofeng}@gmail.com
‡{zhangch, jiyang}@bupt.edu.cn

Abstract—DHT (Distributed Hash Table) algorithms are very efficient for distributed data storage and retrieval. As one kind of P2P overlay, DHT overlay also has the advantages of high reliability, high scalability and low cost. DHT has not only been applied to form user nodes’ overlays, but also been proposed to form DHT-based server farms, such as DHT-based SIP server farm, HSS server farm, DNS server farm, CDN server farm, etc. However, seldom DHT algorithms consider server farm’s stringent requirement on system capacity, and only a handful of DHT algorithms take server into consideration. This paper presents our DHT algorithm called SFDHT for high throughput DHT server farm. Compared with existing DHTs, SFDHT considers the characters and requirements of DHT server farm and maximizes system capacity. SFDHT is a one-hop DHT with novel built-in load balancing solution. The proposed load balancing solution produces much less overhead that existing solutions do. Both theoretical analysis and simulation results show that SFDHT can reduce overhead, balance load and improve system capacity.

I. INTRODUCTION

In recent years, DHT (Distributed Hash Table) algorithms, such as Chord [1], CAN [2] and Pastry [3], have been proposed. DHT algorithms are very efficient for distributed data storage and retrieval. As one kind of P2P overlay, DHT overlay also has the advantages of high reliability, high scalability, self-organized, self-configured and self-maintained. Due to above advantages, DHT has not only been applied to form common DHT overlays, but also been proposed to form DHT-based server farms, such as DHT-based SIP server farm [4], [5], DHT-based HSS server farm [6], DHT-based DNS server farm [7]–[9], DHT-based CDN server farm [10], etc. DHT makes data storage and retrieval in server farm more efficient, reliable and scalable, as well reduces costs to deploy and maintain server farm.

However, seldom DHT algorithms consider server farm’s stringent requirement on system capacity, and only a handful of DHT algorithms take server into consideration. System capacity (i.e. maximum throughput) is the most important performance indicator in DHT server farm. Each node in DHT server farm is supposed to store massive amount of data and perform massive amount of DHT put/get/delete operations. Thus, the DHT algorithm must optimize maximum throughput of DHT put/get/delete operations in DHT server farm. DHT server farm’s system capacity can be maximized by two following ways.

• Reducing overhead by adopting one-hop DHT instead of multi-hop DHT in DHT server farm. Less routing and maintenance overhead means more server resources can be used in data storing and retrieving, which increases the maximum throughput. Compared with other types of DHTs, the sum of routing overhead and maintenance overhead that one-hop DHT produces in server farm is lower, which will be shown in Section IV. Unfortunately, existing one-hop DHTs all have some little flaws, which will be discussed in Section II.

• Improving load balancing solution. If load distribution is not balanced, some servers may be overloaded while many servers are still underloaded. Obviously, system capacity can be higher if the load is balanced among servers. However, existing load balancing solutions all have the problem of high maintenance overhead or data migration overhead, which will be discussed in Section II. These overhead will be higher if the requirement on load balancing is more stringent. Unfortunately, DHT server farm has very stringent requirement on load balancing.

Considering above ways to increase system capacity and the characters of DHT server farm, we design a DHT algorithm called SFDHT for high throughput DHT server farm. SFDHT is a one-hop DHT with a built-in novel load balancing solution. In SFDHT, each physical node has one node ID and multiple partition IDs. Node IDs are used for maintenance. Partition IDs are used to partition hash space into segments and assign segments to nodes.

Compared with existing DHTs, SFDHT considers more about the characters and the system capacity requirement of server farm. As a DHT algorithm designed for server farm, SFDHT has two advantages over existing DHTs. Firstly, SFDHT is designed with a built-in load balancing solution, which overcomes the high overhead problem in existing solutions. Load balancing solution in SFDHT doesn’t increase maintenance overhead, and barely increases data migration overhead. Secondly, SFDHT overcomes the flaws in existing one-hop DHTs, which will be discussed in Section II.

The rest of this paper is structured as follows. The related work is presented in Section II. Section III describes the DHT designed for server farm. We evaluate performance of SFDHT in Section IV. Finally, we conclude and present future work in Section V.
II. RELATED WORK

A. One-hop DHT

The readers of this paper is assumed to be familiar with DHT. So we skip describing DHT mechanism and survey one-hop DHT directly. Several one-hop DHTs have been proposed. They are mainly heterogeneous in their membership information (i.e. node joining/leaving information) dissemination/broadcast mechanisms. In [11], some central nodes called configuration service nodes are responsible for collecting and disseminating entire membership information to all nodes. Gossip-based group membership protocols, such as SWIM [12], can be used to update routing table. However, gossip-based group membership protocol disseminates messages randomly, which results in many duplicate messages and long dissemination time. [13] [14] separately uses a static and a dynamic dissemination hierarchy to broadcast membership changing events. Some nodes are responsible to collect, merge and dispatch events. To wait for other events to merge, messages are not sent immediately when detecting events in [13], [14]. However, intervals between events in server farm are usually long. The attempt of merging events only increases the time of dissemination. Leong et al. [15] present a protocol that uses a novel token passing mechanism. A token is a message that contains network events for updating routing table. When a node receives a token, it can simply pass the token to its predecessor, or generate q secondary tokens and send them to other nodes in serial, which results in long latency.

Though these DHTs all have designed broadcast/dissemination mechanisms for updating routing table, they all have some drawbacks. [12]–[14] don’t consider how to update routing table when broadcasting fails, and can not make sure that new joining nodes can receive broadcast messages. Broadcast in [11], requires that central nodes (configuration service nodes) must be alive anytime.

Compared to existing one-hop DHTs, our one-hop DHT is designed with the consideration of the server farm environment and load balancing. Our one-hop DHT also overcomes the drawbacks in existing one-hop DHTs. Compared with [11], our DHT doesn’t require central nodes. The membership information disseminating in our DHT is fully distributed. Compared with [12]–[15], our DHT needs a shorter period of time to disseminate membership information to all nodes. Compared with [12]–[14], our DHT makes sure that new joining nodes can receive broadcast messages and considers the situation when broadcast fails.

B. Load Balancing in DHT

If no load balancing solution is applied, each node owns a hash space segment of random length, which results in a very skew load distribution. Load distribution is more skew when node capacities are heterogenous.

Existing load balancing solutions can be classified into two types: ID manipulating solution and virtual node solution (also known as virtual server solution). [16], [17] are typical ID manipulating solutions, while [18], [19] are typical virtual node solutions.

ID manipulating solutions balance hash space segments owned by nodes through elaborately assigning and reassigning node IDs. ID manipulating solution brings in too much additional overhead if load balancing requirement is stringent. To balance load, many node IDs need to be reassigned when a node joins or leaves overlay. This results in migrating a lot of data. It also increases maintenance overhead because it requires many messages to update routing table due to the changes of node IDs.

In virtual node solutions, each physical node runs multiple virtual DHT nodes. So the load of a physical node is determined by the amount of all the hash space segments owned by its virtual nodes. Each physical node runs a number of virtual nodes proportional to its capacity. Virtual nodes can also be created, deleted and transferred dynamically based on the changing load distribution. But that will bring in some additional data migrating overhead. Virtual node solution can make load distribution very even if virtual node number per physical node is big enough. Unfortunately, more virtual node number per physical node means more maintenance overhead. As multiple virtual DHT nodes run in a physical node, the DHT maintenance overhead increases multiple times in a physical node.

SFDHT adopts a novel load balancing solution. Each physical node has one node ID and multiple partition IDs. Node IDs are used for maintenance. Partition IDs are used to partition hash space into segments and assign segments to nodes. The same as virtual node solution, our solution balances load by assigning each node multiple hash space segments of random length. However, our solution doesn’t increase maintenance overhead, and barely increases data migration overhead.

III. SFDHT DESIGN

![Fig. 1. Node ID and partition ID spaces](image-url)
A. Node ID and Partition ID

Existing DHTs use node ID for both maintenance and hash space assignment. SFDHT separates the second role from node ID’s roles. In SFDHT, each node has a unique node ID and multiple partition IDs. Partition IDs in the overlay are all unique. The node ID is only used for maintenance, and partition IDs are used for hash space assignment. These IDs are generated randomly by nodes. As shown in Fig. 1, node IDs and partition IDs are mapped into two ring-shaped spaces respectively. The ring containing node IDs indicates the overlay’s topology. Partition IDs and data object keys share the same hash space. The ring containing partition IDs determines how the hash space is partitioned into segments and how these segments are assigned to nodes. The node with more partition IDs owns more segments. The node with larger capacity should generate more partition IDs.

B. Data Storage and Hash Space Assignment

In DHTs, each original data object is stored in its root node, which is “closest” to the data object key. The first replication of a data object is usually stored in the second “closest” node. So the second closest node will automatically become the new closest node if the original closest node leaves overlay. If there are more replications, they are stored in a set of closest nodes.

Several “close” rules have been proposed in existing DHTs, while “close” rule in SFDHT is a little different from existing DHTs. In existing DHTs, data object keys and node IDs share the same hash space. The “close” rules in existing DHTs is used to measure the distance between a node ID and a data object key. We adopt these “close” rules to measure the distance between a partition ID and a data object key. The distance between a node and a data object key is minimal distances among the distances between the key and the node’s partition IDs. In other words, the closest node in SFDHT is the node, which has the closest partition ID to the data object key. As the second closest partition ID may be owned by the root node too, the replication is stored in the second closest node, not always the node owns the second closest partition ID. An example is shown in Fig. 1. Data object k1’s closest partition ID is a1, which belongs to node A. Then node A is the root node of the data object k1. Replication of k1 is stored in node E instead of node A.

Let us see the data storage distribution from the view of hash space assignment. In existing DHTs, with certain “close” rule, the whole hash space is partitioned into segments based on node IDs. Each hash space segment is assigned to a node, which is the root node of all keys in the hash space segment. In SFDHT, the whole hash space is partitioned into segments based on partition IDs. Each partition ID has a corresponding hash space segment. As each node has multiple partition IDs, each node owns multiple hash space segments.

C. Routing Table and Routing

Every node maintains a routing table. As shown in TABLE I, a routing table contains all partition IDs and their associated node IDs and addresses. For simplification, the node IDs and partition IDs shown in this paper are only 16-bit numbers.

We describe routing process with the routing table shown in TABLE I. Suppose a node, say node A, owns the routing table. If node A wants to put/get/remove a data object with key 0x75ad. Node A first lookups its routing table and finds out partition ID 0x8000 is closest to 0x75ad. Then node A sends a put/get/remove request to the node owns the partition ID, whose node ID is 0x4444 and address is 10.0.0.3:2000.

D. Maintenance

Every node maintains a node list containing all alive nodes’ node IDs and a routing table containing all alive nodes’ partition IDs. When a node joins the DHT overlay, it get the node list and routing table from its predecessor (the next node in the node ID ring in clockwise direction).

To update the node list and routing table, membership changing events need to be broadcasted to all nodes in the overlay. The joining node broadcasts its joining. If a node leaves the overlay gracefully, the leaving node will broadcast its leaving before it leaves. If a node fails, its successor will detect that and broadcast its leaving.

To detect the failure of nodes, each node sends keep-alive messages to its predecessor (the next node in the node ID ring in counter-clockwise direction). Each node also replies ACK messages to its successor when receiving keep-alive messages from its successor. Fig. 1 shows the keep-alive mechanism.

Two mechanisms are needed to update node list and routing table in case the broadcast fails. Some nodes may not receive the broadcast messages if one or more broadcasting nodes fail in the middle of broadcast. Unaware nodes (the nodes failing to receive broadcast messages) can detect the joining/leaving event during routing. If a node is found unreachable during routing, unaware nodes delete the node from node list and all its routing entries. If a joining event is missed, unaware nodes will send the joining node’s successor some requests, which are supposed to be sent to the joining node. The joining node’s successor finds out that it is not responsible for these requests, and redirects those requests to the joining node. Then the unaware nodes find out the joining node and update node list and routing table.

E. Broadcast mechanism for maintenance

Fig. 2 shows the application-level broadcast mechanism in SFDHT used to update node list and routing table. The nodes
that send broadcast messages to other nodes are referred to as broadcast nodes in this paper. The node that starts a broadcast is referred to as starting node. As discussed above, starting node could be a joining node, leaving node or failing node’s successor.

The broadcast can be one level if node number is small. If node number is large, one-level broadcast may overburden the starting node and takes a long time. Then multi-level broadcast should be used. The multi-level broadcasting can be viewed as a recursive process of one-level message sending.

A broadcast message includes node ID space range, membership changing type and the joining/leaving node’s information (node ID, partition IDs, capacity). Membership changing type indicates the membership changing node is joining or leaving. Node ID space range indicates that the message should be sent to all nodes in the node ID space range. Capacity attribution is shared for load balancing, which will be discussed later.

The behavior of a node in the broadcast is described here. Suppose a node is responsible to broadcast a message in a node ID space range. At the beginning of broadcasting, the starting node is the only broadcast node, which is responsible for the broadcast in the whole node ID space. If there are no nodes other than itself in that node ID space range, the responsible node does not need to send messages. Otherwise, it broadcasts in that node ID space range following two steps.

- First, broadcasting node divides its responsible node ID space range into some segments. Among these segments, there is one invalid segment, which ranges from the broadcasting node to its predecessor. Rest segments are valid segments. Node number of each valid segment is the same or almost the same. Each valid segment has a head node, which is first node in the segment in counterclockwise direction.
- Second, broadcasting node sends head nodes the broadcast messages in parallel. In the message sent to each head node, the node ID space range has been set as the range of the head node’s segment. Then each head node is responsible for the broadcast in its segment.

With node ID space range in the broadcast message, the new joining nodes can receive the broadcast message even if the broadcasting node(s) of former broadcast level(s) is not aware of the new joining nodes. As shown in Fig. 2, node A and B join the overlay after node P starts broadcasting. Node A and B can still receive the broadcast message because node S and E respectively detects the joining of node A and B.

If broadcasting node detects a head node fails, it chooses the successor of failing node as the new head node. The range of failing node’s segment enlarges accordingly. For example, in Fig. 2, after detecting node F fails, broadcasting node P chooses node G as the new head node. Node ID space segment [F,S) is extended to [G,S). G will receive extra broadcast messages. But that can make sure that any new joining node in node ID space segment [F,C) will receive broadcast message.

### F. Load Balancing

In server farm, data object number is much larger than node number. So the proportion of hash space a node owns is nearly equal to the proportion of data the node is responsible for. Thus, hash space assignment determines load distribution.

To evaluate the balancing degree of load distribution precisely, we need to define a mathematical way for evaluation. Suppose nodes are indexed from 1 to \( N \). The capacity of node \( i \) is \( C_i \). Each node’s capacity can be estimated by the node itself or operator in the same standard. Then the proportion of hash space that node \( i \) should owns is \( C_i / \sum_{k=1}^{N} C_k \). Let \( P_i \) be the proportion of hash space that node \( i \) actually owns. So \( C_i / \sum_{k=1}^{N} C_k \) is closer to 1 means that load distribution is more balanced. We define node i’s LAF (Load Assignment Factor) \( L_i \) as \( P_i / C_i / \sum_{k=1}^{N} C_k \). The node with highest LAF will overload, if actual system’s load is more than \((\max_{i=1}^{N} L_i)^{-1}\) times of the optimal system capacity (the system capacity when fully load balanced). Then some put/get/remove requests will be processed with high latency or can not be processed. Thus, the smaller \( \max_{i=1}^{N} L_i \), the higher the system capacity. To assure QoS and system capacity, we can set a threshold \( LT \) for LAF and prevent any node’s LAF becoming higher than the threshold.

### Partition ID generation

Each node owns a number of random partition IDs proportional to its capacity. Specifically, the node with capacity \( C_i \) has \( d \times C_i \) partition IDs, where \( d \) is a parameter shared in the server farm to control the partition ID number. Therefore, the node with larger capacity generates more partition IDs and owns more proportion of hash space. Partition ID generation requires sharing capacity attribute, which is done by maintenance. Capacity information is included in the node list maintained by each node. To make sure node joining won’t break the load balance, partition IDs are generated with the consideration of current load distribution status. Suppose nodes that already in the overlay are indexed from 1 to \( N \) and the joining node is
node \( x \). When node \( x \) joins overlay, it first gets current routing table, node list from bootstrap peer. Then the node \( x \) runs the following algorithm to generate partition IDs.

\[
\text{GeneratePartitionID}(\text{routing table}, \text{node list}, d)\\
1) \text{Node } x \text{ calculates all nodes’ hypothetic LAFs } L'_1 \ldots L'_N \text{ and } L'_x \text{ under the supposition that it uses its generated partition IDs if there are any. (At the beginning, node } x \text{ has no generated partition ID). }\\
2) \text{Node } x \text{ picks the node with the highest LAF, say node } y \text{ (the node with the highest LAF could be node } x \text{ itself). }\\
3) \text{From the key space segments owned by node } y, \text{ node } x \text{ picks the one with max length, say key space segment } z.\\
4) \text{Node } x \text{ generates a random partition ID in the key space segment } z, \text{ which is longest segment owned by the node } y.\\
5) \text{If node } x’s \text{ partition ID number is less than } d \times C_x, \text{ go back to step 1. Otherwise, end the algorithm. }
\]

**Partition ID migration.** To restore the load balance broken by node leaving, partition IDs are transferred from heavily loaded nodes to lightly loaded nodes according to current load distribution status. When any node leaves overlay, each node except the leaving node runs the algorithm bellow. In order to sustain the ratio of partition ID number and capacity, the algorithm minimizes the number of transferred partition IDs.

\[
\text{TransferPartitionID}(\text{routing table}, \text{node list}, d, LT)\\
1) \text{Node } i \text{ calculates its LAF } L_i. \text{ If } L_i > LT, \text{ continue the algorithm. Otherwise, end the algorithm. }\\
2) \text{Node } i \text{ sorts the lengths its key space segments, and picks some partition IDs with biggest key space segment length for transferring. Partition IDs are picked one by one from the top until transferring picked IDs can make LAF lower than LT. }\\
3) \text{Node } i \text{ calculates all nodes’ LAFs } L_1 \ldots L_N \text{ and negotiates with the nodes with lowest LAFs to determine which node(s) can accept partition ID. }\\
4) \text{Node } i \text{ transfers picked partition IDs and associate data to the the node(s) willing to accept partition ID. }
\]

Even the data distribution is balanced, hot spots may break the balance of DHT operations. Some hot data may be queried at a rate extremely higher than other data, which may cause the access rate on servers unbalanced. Cache mechanism can address the hot spot issue. Each server maintains a cache storing data which have been recently gotten from other servers. This can reduce the load of the node storing hot data.

**IV. PERFORMANCE EVALUATION**

This section compares routing overhead and maintenance overhead between SFDHT and multi-hop DHT in server farm context, and evaluates the load distribution and load balancing overhead. Notations used in this section is listed in TABLE II.

**TABLE II**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>physical node number. Normally, node number in a server farm ranges from 2 to tens of thousands. Thus, ( N ) ranges from 2 to ( 2^{16} ) in the performance evaluation.</td>
</tr>
<tr>
<td>( R )</td>
<td>Rate of DHT put/get/remove operations (times per second per node). Each server is supposed to handle hundreds or even thousands of requests per second. So the values of ( R ) are set between 100 and 5000 in the performance evaluation.</td>
</tr>
<tr>
<td>( U )</td>
<td>Membership changing rate, i.e. the rate of node joining/leaving (times per second)</td>
</tr>
<tr>
<td>( M )</td>
<td>MTBF (Mean Time Between Failures). Usually a server can run without failure for months. So the MTBF values in the performance evaluation we pick are ( 10^3 ), ( 10^4 ) and ( 10^5 ) hours.</td>
</tr>
<tr>
<td>( MPN )</td>
<td>Minimal partition ID number in SFDHT, i.e. the partition ID number in the weakest node.</td>
</tr>
<tr>
<td>( MVN )</td>
<td>Minimal virtual node number in virtual node solution, i.e. the virtual node number in the weakest node.</td>
</tr>
</tbody>
</table>

**A. Routing and Maintenance Overhead**

Routing overhead and maintenance overhead are compared separately between SFDHT and typical multi-hop DHTs (e.g. Chord and Pastry) first, then we compare overall overhead (the sum of routing overhead and maintenance overhead) between SFDHT and all multi-hop DHTs.
of the manipulated key, routing to the root node requires zero message. The probability is \(1/N\) if every node owns equal proportion of hash space. So the routing overhead is about \(2R(1−1/N)\) messages per node per second.

We compute the routing overhead of SFDHT and typical multi-hop DHT and present them in Fig. 3. From Fig. 3, we can see SFDHT’s routing overhead is smaller than multi-hop DHT’s overhead when node number is larger than 2.

Maintenance overhead includes keep-alive overhead and topology update overhead. However, keep-alive overhead can be omitted in server farm. In server farm, frequency of DHT routing is very high and DHT node number is relative small. So usually all DHT neighbors are accessed again before keep-alive timers are due.

Topology update overhead are evaluated with the number of messages involved with updating routing table when nodes join or leave overlay. Topology update overhead in different multi-hop DHT varies from \(O(\log^2 N)\) to very low value. We take \((2 \times U \times \log N)/N\) to compare with the topology update overhead in SFDHT. In SFDHT, topology update message (i.e. broadcast related message) number per node per second is \(2 \times (N−1) \times U/N\).

Load distribution is also compared between partition ID and virtual node solution. The same as virtual node solution, our solution balances load by assigning each node multiple hash space segments of random length. But our solution adopts a novel algorithm to generate partition ID, while virtual node IDs are generated randomly. To compare hash space segment generating, load distribution are compared without applying partition ID migration or virtual node migration. And this comparison assumes that no node leaves DHT server farm.

To evaluate the effect of load balancing, we compare the load distribution in SFDHT and one-hop DHT without load balancing solution.

Load distribution is evaluated with following considerations:

- To evaluate the effect of load balancing, we compare the load distribution in SFDHT and one-hop DHT without load balancing solution.
- Load distribution is also compared between partition ID solution and virtual node solution. The same as virtual node solution, our solution balances load by assigning each node multiple hash space segments of random length. But our solution adopts a novel algorithm to generate partition ID, while virtual node IDs are generated randomly. To compare hash space segment generating, load distribution are compared without applying partition ID migration or virtual node migration. And this comparison assumes that no node leaves DHT server farm.
- ID manipulating solution can distribute load evenly as the other two solutions, and it is not practical due to overhead issue. So the load distribution with ID manipulating solution is not compared here.

B. Load Distribution

Load distribution is evaluated with following considerations:

- To evaluate the effect of load balancing, we compare the load distribution in SFDHT and one-hop DHT without load balancing solution.
- Load distribution is also compared between partition ID solution and virtual node solution. The same as virtual node solution, our solution balances load by assigning each node multiple hash space segments of random length. But our solution adopts a novel algorithm to generate partition ID, while virtual node IDs are generated randomly. To compare hash space segment generating, load distribution are compared without applying partition ID migration or virtual node migration. And this comparison assumes that no node leaves DHT server farm.
- ID manipulating solution can distribute load evenly as the other two solutions, and it is not practical due to overhead issue. So the load distribution with ID manipulating solution is not compared here.

In a realistic server farm, usually there is only one or a few types of servers. Therefore, in the simulation, the capacities are the same or constrained in a few values. The simulation runs.
under different capacity settings, and the results are similar. So we only show the figures of homogenous capacity setting here.

Fig. 5 and Fig. 6 show the maximum and minimum LAF. Fig. 7 and Fig. 8 show the empirical CDF (Cumulative Distribution Function) of LAF. As shown in these figures, LAFs are much closer to 1 and maximum LAF is reduced a lot when using our partition ID solution, which means our partition ID solution can increase system capacity a lot. As shown in Fig. 6 and Fig. 8, load distribution is evener if MPN or MVN is bigger. Compared with virtual node solution, our load balancing solution distributes load evener when MPN is equal to MVN. So, to make load as even as our solution do, virtual node solution has to create more key space segments or migrate more data using virtual node migration.

C. Increased Overhead Caused by Load Balancing

Virtual node solution, ID manipulating solution and our partition ID load balancing solution all can meet the stringent load balancing requirement by DHT server farm. However, three kinds of load balancing solutions bring in different overhead to meet the requirement.

To meet the stringent load balancing requirement by DHT server farm, virtual node solution has to increase DHT node number a lot, which results in increasing maintenance overhead a lot. Suppose each physical node runs 100 virtual nodes. Then the DHT node number is 100 times of physical node number. So the keep-alive messages increase 100 times, and a broadcast requires 100 times of messages too. If a physical node fails, 100 virtual nodes’ failing will be broadcasted, which requires 100 times of broadcasts and 10000 times of broadcast messages.

To meet the stringent load balancing requirement by DHT server farm, ID manipulating solution increases migrating data and broadcast messages a lot. Suppose load distribution in a server farm is satisfied for now. A node’s joining or leaving will break the balance. ID manipulating solution requires changing a lot of node IDs to rebuild balance again. A lot of broadcast messages are required to inform the changing of node IDs. And a lot of data has to migrate among nodes. Partition ID solution barely increases data migration overhead, and the increased overhead is much less than the overhead increased by virtual node solution. In partition ID solution and virtual node solution, partition ID migration and virtual node migration additionally migrate data from nodes with LAFs higher than threshold. Partition ID migration only happens when node leaving, while virtual node migration may happen when node joining or leaving. Suppose the LAF threshold is 1.05. Then, as shown in Fig. 6 and Fig. 8, partition ID solution doesn’t need to migrate partition ID if no node leaves,
and virtual node solution still needs to migrate non-ignorable amount of data if MVN is not big enough. From Fig. 9 and Fig. 10 we can see that partition ID migration seldom happens if MPN is no less than 100, and only a very small percentage of data owned by a node is migrated when partition ID migration happens. From Fig. 9 and Fig. 10 we can see that partition ID migration seldom happens if MPN is no less than 100, and only a very small percentage of data owned by a node is migrated when partition ID migration happens.

Our load balancing solution doesn’t increase maintenance overhead, and barely increase data migrating overhead. Compared with other solutions, our solution has less data migrating overhead. Clearly, our solution produces much less overhead than the other solutions do. Our load balancing solution only requires larger memory to store routing table, which is still feasible for servers. Suppose a routing table entry requires a space of about 50 bytes (160 bits for node ID, 160 bits for partition ID, 6 bytes for IP and 2 bytes for port). Suppose a server farm has $10000 \times 100$ partition IDs owned by 10000 servers. A routing table will takes up about 50M bytes space, which can be afforded by a server absolutely.

V. CONCLUSION AND FUTURE WORK

This paper has presented our DHT algorithm called SFDHT, which is designed for high performance DHT server farm. SFDHT is a one-hop DHT with novel built-in load balancing solution. The load balancing solution and other design details of SFDHT have been described in this paper. The performance of SFDHT has been evaluated by theoretical analysis and simulation. Performance evaluation shows that SFDHT has low overhead and the load is well balanced. The overhead of load balancing is also much lower than that of existing load balancing solutions. DHT server farm’s maximum throughput is increased due to overhead reducing and load balancing.

We have proposed load balancing solution based on partition ID. To address hot spot issue, we still need to design cache mechanism in details, and propose other mechanisms. We are also implementing SFDHT and will test its performance in the future.

### REFERENCES


