Journal of Zhejiang University SCIENCE A
ISSN 1009-3095 (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Using bidirectional links to improve peer-to-peer lookup performance

JIANG Jun-jie[†1], TANG Fei-long[1], PAN Feng[1], WANG Wei-nong[2]
([1]*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200030, China*)
([2]*Network Center, Shanghai Jiao Tong University, Shanghai 200030, China*)
[†]E-mail: jiang.jj@gmail.com

Received Mar. 9, 2005; revision accepted May 27, 2005

Abstract: Efficient lookup is essential for peer-to-peer networks and Chord is a representative peer-to-peer lookup scheme based on distributed hash table (DHT). In peer-to-peer networks, each node maintains several unidirectional application layer links to other nodes and forwards lookup messages through such links. This paper proposes use of bidirectional links to improve the lookup performance in Chord. Every original unidirectional link is replaced by a bidirectional link, and accordingly every node becomes an anti-finger of all its finger nodes. Both theoretical analyses and experimental results indicate that these anti-fingers can help improve the lookup performance greatly with very low overhead.

Key words: Distributed hash table (DHT), Peer-to-peer, Lookup performance
doi:10.1631/jzus.2006.A0945 **Document code:** A **CLC number:** TP393.02

INTRODUCTION

Peer-to-peer computing has become a popular distributed computing paradigm. Efficient resource lookup is essential for such systems.

As we know, most of the deployed peer-to-peer systems are unstructured. Napster is based on a central directory and was in popular use during early days. Soon after, Gnutella and KaZaA were deployed widely. However, all these popular unstructured peer-to-peer systems suffer from unscalability.

Fortunately, there are new kinds of peer-to-peer networks—the so-called structured peer-to-peer networks such as CAN (Ratnasamy *et al*., 2001) and Chord (Stoica *et al*., 2003), which are based on distributed hash table (DHT). In a large-scale DHT system, a lookup generally cannot be resolved by just one node, so that a lookup always reduces to the routing of the lookup message.

Chord is a based on DHT representative peer-to-peer lookup service, in which every node maintains a finger table as its routing table. There is a unidirectional application layer link from a node to its each finger node. Due to the dynamics of peer-to-peer networks, a heartbeat mechanism is generally used to perceive the churn and detect the availability of each link. Such heartbeat messages are considered to be the dominating maintenance overhead in Chord.

In this paper, we propose to replace each unidirectional application layer link by a bidirectional link. That is, for each original directed edge in the Chord topology graph, a reverse edge is added so that every node in Chord becomes an anti-finger of its each finger node. Then each node should maintain an anti-finger table in addition to its finger table, but the additional maintenance overhead is very low because a single heartbeat message could be used to maintain a couple of links in opposite directions. The performance analyses and experimental results in this paper show that such anti-fingers can help improve the lookup performance greatly.

CHORD OVERVIEW

In Chord, both data objects and nodes are as-

signed an m bits identifier by using a consistent hashing such as SHA-1. A node's identifier is obtained by hashing the node's IP address and service port number while a data object's identifier is produced by hashing itself or its name. We will use the term "node id" and "key" to refer to the identifier of a node and a data object respectively. The node id of node x is denoted by $id(x)$. Sometimes, we also denote a node using its node id. Consequently, Chord defines a name space as a sequence of m bits and arranges the name space on a scaled virtual ring modulo 2^m, which is called the Chord ring.

Along the Chord ring, all the identifiers including node ids and keys are ordered. Key k is assigned to the first node whose node id is equal to or follows k clockwise along the Chord ring and the node is called the successor node of k, denoted by $succ(k)$. Also, the successor node of a node x is the first node clockwise from $id(x)$ along the Chord ring, and is denoted by $succ(x)$.

In Chord, each node maintains a routing table, called the finger table and each routing table entry is called a finger of the node. The ith finger of node x, denoted by $x.finger(i)$, contains the identity of the first node s, that succeeds x by at least 2^{i-1} along the Chord ring clockwise, namely $s=succ(id(x)+2^{i-1})$, $1\leq i\leq m$. The finger table of each node may contain m fingers at most and in fact, the finger table size is $\log_2 N$ with high probability, where N is the number of nodes in the network.

A Chord ring with m=3 is shown in Fig.1. There are four nodes in the network—0, 1, 3 and 6. In addition, there are four data objects, whose keys are 1, 2, 6 and 7 respectively. The four data objects are assigned to their keys' successor nodes, i.e., nodes 1, 3, 6 and 0 respectively. In other words, the four data objects are located at nodes 1, 3, 6 and 0 respectively.

Fig.1 also shows the finger tables of these nodes. For example, we have the node id of the ith finger of node 0, $id(0.finger(i))=id(succ(0+2^{i-1}))$, $1\leq i\leq 3$. Therefore, the node ids of these fingers are 1, 3 and 6 respectively.

Chord just provides one operation: to lookup the node responsible for a given key, i.e., the successor node of the key. Therefore, a lookup for the key can be resolved as long as the lookup message is routed to the node.

The Chord lookup algorithm is similar to binary search. As stated above, each node in Chord maintains a finger table consisting of m fingers at most. To lookup a given key k, a node will check its finger table and forward a lookup message to the finger that is closest to but does not overshoot k clockwise. And then the finger will do the same in an iterative or recursive manner. After several iterations or recursions, the lookup message will ultimately arrive at the node that immediately precedes k. Then the node will return the address of its successor node, which is also the successor node of k. The lookup is resolved. It is clear that in Chord, all the lookup messages are passed clockwise along the Chord ring.

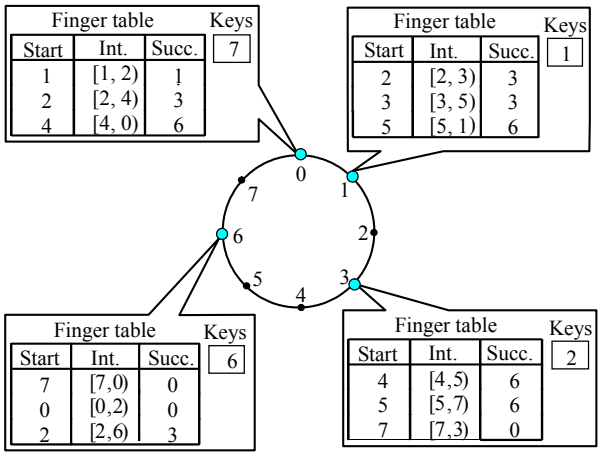

Fig.1 An example of Chord ring

DESIGN OF BICHORD

BiChord construction

Two reasons inspire us to propose BiChord. The first reason is that a single heartbeat message between each pair of nodes can be used by the nodes to perceive the arrival or departure of each other. Such a message can be utilized to maintain a bidirectional link. The second reason is that in Chord, all the lookup messages are passed clockwise along the Chord ring and we think it is inefficient, because to lookup the keys located near but preceding the node, the lookup messages will have to traverse almost the whole Chord ring.

The principle of BiChord is quite simple. In BiChord, a node needs to maintain a so-called anti-finger table in addition to the finger table. When there

is a link in Chord, we add a reverse link that links the same two nodes but in the reverse direction. So if one node A is a finger of the other node B, then node B becomes an anti-finger of node A. There are no changes to the responsibility of data objects. All the data objects are still located at the successor nodes of their keys. Therefore each node in BiChord maintains: (1) finger table; (2) successor list; (3) anti-finger table. The former two are the same as in Chord while the last one includes the information about all the anti-fingers of the node.

Fig.2 gives an example of BiChord. This example is the BiChord version of the example in Fig.1. In Chord, node 0 is the 3rd finger of node 3 and the 1st and 2nd fingers of node 6. So nodes 3 and 6 are included into the anti-finger table of node 0. The original unidirectional links from nodes 3 and 6 to node 0 are both replaced by a pair of symmetrical links, i.e., a bidirectional link.

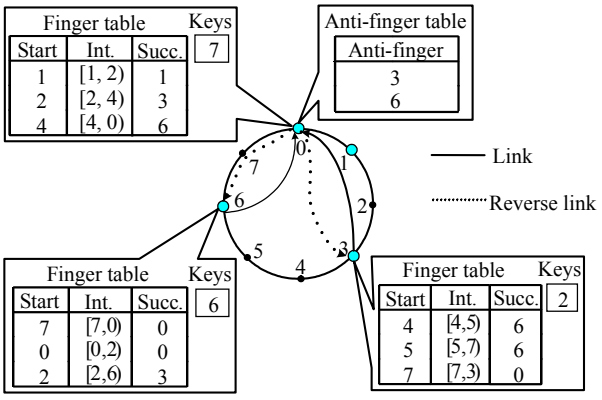

Fig.2 An example of BiChord

BiChord lookup algorithm

We take the finger table and the anti-finger table as a whole and call it the routing table. In the routing table, all the entries are ordered. To lookup a given key k, a node will check its routing table to find a certain table entry, namely one of its fingers or anti-fingers, whose identifier is closest to k among all the entries. Then the lookup message is forwarded to this node. Ultimately, the lookup message will arrive at the node closest to the key k among all the nodes in the network. Since we have not changed the responsibility of data objects, this node must be the predecessor or successor node of key k. If the node is the successor node of k, the lookup is resolved. Or else, this node must be the predecessor node of k. Then it can return the address of its successor node, which is also the successor node of k. The lookup is resolved likewise.

The pseudocode of BiChord lookup algorithm is shown in Fig.3. Remote calls and variable references are both preceded by the identifier of remote node, while local variable references and procedure calls both omit the identifier of local node. The mechanisms to deal with the join and stabilization remain unchanged from that in Chord.

```
// ask node n to find the successor of k
n.find_successor(k) {
    if (k∈(id(predecessor), id(n)])
        return n;
    else
        if (k∈(id(n), id(successor)])
            return successor;
        else {
            n'=closest_node(k);
            return n'.find_successor(k);
        }
}

// search the local routing table for the closest
// node to k, R denotes the routing table
n.closest_node(k) {
    x=R(1);
    for i=1 to ||R||
        if (|R(i)–k|<|x–k|)
            x=R(i);
    return x;
}
```

Fig.3 BiChord lookup algorithm

PERFORMANCE ANALYSIS

The two main performance metrics we discuss here are routing table size and lookup path length. Moreover, we give a precise proof on the routing table size in BiChord.

Theorem 1 Each node maintains a finger table with at most m entries and with high probability (whp), the size of finger table is $O(\log N)$, where m is the length of identifier and N is the number of nodes in the network.

Proof Since there are no differences between the construction of finger table in BiChord and in Chord, the number of entries in the finger table of each node also remains unchanged. The finger table size in BiChord is the same as that in Chord and it follows that whp, the size of finger table in Chord is $O(\log N)$ and is m at most (Stoica *et al*., 2003).

Theorem 2 With high probability, each node maintains an anti-finger table with $O(\log^2 N)$ entries, where N is the number of nodes in the network. Also, the expected anti-finger table size is $O(\log N)$.

Proof The expected distance between two successive nodes is $2^m/N$ on the Chord ring, and whp, the distance is $L=O((2^m/N)\times\log N)$.

We consider a node n. There are L continuous identifiers between node n and its predecessor node p, i.e., $|id(n)-id(p)|=L$, whp.

If node n is a finger of node x, we have $n=succ(id(x)+2^{i-1})$, $1\le i\le m$. That is to say, $id(x)+2^{i-1}$ is in the range from $id(p)$ to $id(n)$. So $id(x)$ is in the range from $id(p)-2^{i-1}$ to $id(n)-2^{i-1}$. It's clear that for a particular i, $id(x)$ locates in this range with probability $|(id(n)-2^{i-1})-(id(p)-2^{i-1})|/2^m$. That is, node n is a finger of node x with probability $|id(n)-id(p)|/2^m$. As stated above, whp, $|id(n)-id(p)|=L=O((2^m/N)\times\log N)$. Thus node n is a finger of node x with probability $O(\log N/N)$ for any node x and a particular i. Thus whp, for this particular i, there are $O(\log N)$ nodes that finger node n because there are N nodes totally in the network.

Since whp a node has $O(\log N)$ fingers, there are $O(\log^2 N)$ unique nodes with finger node n, whp, and so node n maintains an anti-finger table with $O(\log^2 N)$ entries.

As shown in Theorem 1, whp, every node maintains a finger table with $O(\log N)$ entries. So the sum of the finger table size is $O(N\times\log N)$ and on average, each node is a finger of $O(\log N)$ nodes. Thus the expected anti-finger table size is $O(\log N)$.

Theorem 3 If in a BiChord network, every node maintains a correct finger table and a correct anti-finger table, the BiChord lookup algorithm is convergent.

Proof During each step of BiChord lookup algorithm, if current node cannot find a routing table entry closer to the desired key than itself, the node must be the predecessor or successor node of the key and the lookup is resolved because if not so, the predecessor or successor node of this node must be closer to the key.

Otherwise, the lookup message will be forwarded to a finger or anti-finger of the current node that is closest to the desired key among all the routing table entries. Obviously, such a next hop node is closer to the desired key than current node. In such a case, the lookup message is routed to a closer node to the desired key at each step. Ultimately, the lookup message will arrived at the node preceding or succeeding the key, namely the predecessor or successor node of the key and the lookup is resolved.

Since the anti-finger table brings much uncertainty into the routing process, it's rather difficult to give an exact analytical result on the lookup path length in BiChord. But the lookup efficiency is improved distinctly. In Chord, each node maintains information about a small number of other nodes (fingers) and knows more about the nodes closely following it along the Chord ring than the nodes farther away, whereas in BiChord, each node maintains not only such information but also information about more nodes (anti-fingers). A node knows many things about the nodes near it at sides clockwise and counterclockwise instead of only the clockwise side. Intuitively, the fingers and anti-fingers of each node partition the Chord ring finer and by these fingers and anti-fingers, a node will locate the region of a desired key more accurately at each step. Thus the nodes will resolve a lookup within fewer steps.

EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of BiChord by simulation. The simulator implements the BiChord lookup algorithm shown in Fig.3. For comparison, we also implement Chord lookup algorithm.

Routing table size

The routing table size is an important performance metric. Although the anti-finger table brings very low additional overhead, we still investigate the anti-finger table size to evaluate the performance of BiChord thoroughly. The finger table size is examined during the experiment too. From Theorem 1 and Theorem 2, with high probability, the finger table and anti-finger table sizes are $O(\log N)$ and $O(\log^2 N)$ re-

spectively, and the expected finger table and anti-finger table sizes are both $O(\log N)$, where N is the total number of nodes in the network.

To understand the routing table size in practice, we simulated a network with $N=2^k$ nodes. We varied k from 3 to 14 and conducted a separate experiment for each value of k. We measured the finger table and anti-finger table sizes of every node during each experiment.

Fig.4a plots the average sizes of finger table and routing table as a function of k. As expected, they both increase logarithmically with the number of nodes. Fig.4b plots the probability density function of the finger table and anti-finger table sizes for a network with 2^{12} nodes (k=12).

Fig.4a confirms that in BiChord, both the average finger table size and the average routing table size are $O(\log N)$. Fig.4b shows that the finger table sizes of most peer nodes are around $\log N$ while the distribution of anti-finger table sizes is more dispersed.

Lookup path length

The routing performance of BiChord mostly depends on the lookup path length. We also simulated a network with $N=2^k$ nodes and 100×2^k data objects here. We varied k from 3 to 14 and conducted a separate experiment for each value of k. During each experiment, every node picked up a random set of keys to lookup using Chord and BiChord lookup algorithm respectively, and we measured each lookup path length of the two algorithms.

Fig.5a plots the average lookup path length of the two lookup algorithms as a function of k. Fig.5b plots the probability density function of lookup path length in the case of a 2^{12} nodes (k=12) network.

Fig.5a indicates that BiChord has great improve-

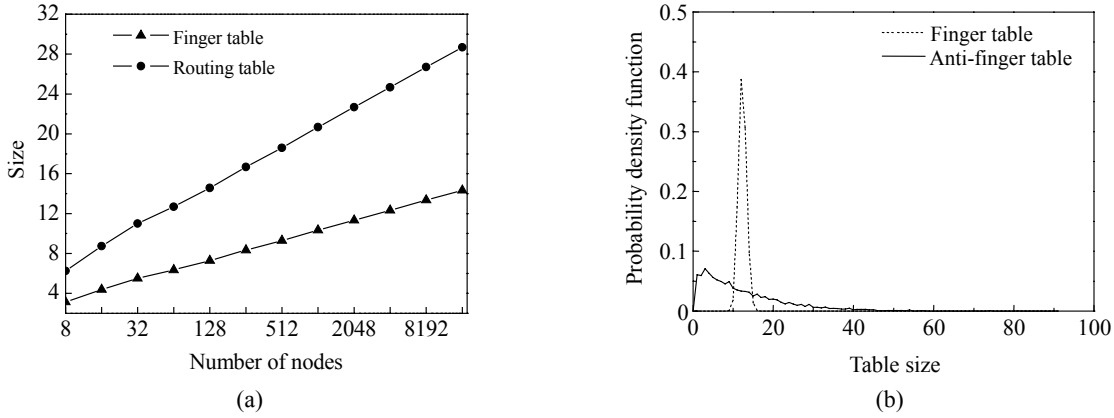

Fig.4 Routing table size in BiChord. (a) Average sizes; (b) The probability density function of the finger and anti-finger table sizes in the case of a 4096 nodes network

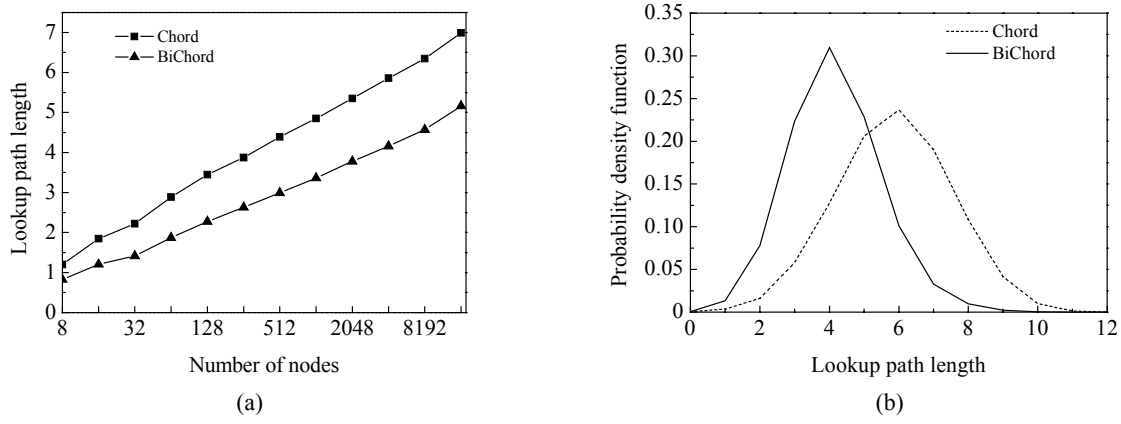

Fig.5 Lookup path length in Chord and BiChord. (a) Average lookup path length; (b) The probability density function of lookup path length in the case of a 4096 nodes network

ment over Chord in average lookup path length. Fig.5b shows more plainly that the average lookup path length in BiChord is lower than that in Chord.

We also counted the number of anti-fingers that act as lookup forward nodes during each BiChord lookup and calculate the percentages of anti-fingers in lookup forward nodes. Fig.6 plots such percentages for each network size.

Fig.6 shows that about half of the lookup forward nodes are achieved by anti-fingers and confirms that the anti-fingers contribute much to the improvements of lookup efficiency.

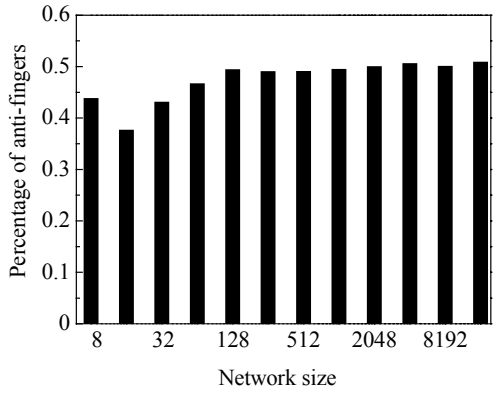

Fig.6 Percentages of anti-fingers in lookup forward nodes

RELATED WORK

Chord provides an efficient peer-to-peer lookup service based on DHT with provable correctness and performance guarantee. By now, Chord has brought up many novel distributed application systems. We also have implemented an experimental system for distributed text retrieval based on Chord (Jiang and Wang, 2004). However, there is still some space for improvement in Chord lookup efficiency.

There are two main ways to improve the lookup efficiency in Chord. One is to optimize the logical topology of Chord such as the denser finger technique (Zhuang and Zhou, 2003). The other is to utilize the underlying network topology information to reduce the total routing latency such as geographic layout and proximity neighbor selection (PNS) techniques (Ratnasamy *et al*., 2002).

The denser finger technique places fingers of node x at points $id(succ(id(x)+(1+1/d)2^{i-1}))$ on the Chord ring, $(1 \le i \le m' \wedge (1+1/d)^{m'} \le 2^m)$ and d is a tunable integer parameter. The number of fingers kept by each node is now d times of that in original Chord and the maximum lookup path length is reduced to $1/\log(1+d)$ of the original length. However, the average lookup path length is $\log N/[(1+d)\log(1+d)-d\log d]$. In our another work, we presented ChordPlus lookup algorithm (Bai *et al*., 2005) which generalizes Chord lookup algorithm to M-ary lookup. It also gets some improvements in lookup path length.

However, work remains to route lookup messages in one way (clockwise) like in Chord. S-Chord was proposed using symmetry to improve lookup efficiency in Chord (Mesaros *et al*., 2003). That is, each node maintains fingers at both its sides and these fingers of node x are placed at points $id(succ(id(x)+4^{i-1}))$ and $id(pred(id(x)-4^{i-1}))$ $(1 \le i \le m)$. S-Chord shows the improvements in routing performance by experiments. BiChord is rather different from the idea of S-Chord. In S-Chord, each link is still unidirectional like in Chord, but in BiChord, we add a reverse link to the topology graph for each original unidirectional link at very low additional maintenance overhead and achieve significant improvements in routing performance.

Exploiting the underlying network topology information has been considered for use in Chord too (Dabek *et al*., 2001).

CONCLUSION

Chord is a scalable peer-to-peer lookup service for Internet applications. The simplicity, provable correctness and performance make it an attractive substrate for distributed applications. However there still remains some space for improvement in the lookup efficiency in Chord.

This paper proposes BiChord, an improved lookup service based on Chord. BiChord utilizes the existing finger table in Chord and the heartbeat mechanism to construct an anti-finger table at very low additional overhead. By using the finger table and anti-finger table, the lookup performance is improved greatly. Theoretical analyses and experiment results both confirm such improvements. Another byproduct is that the fault-tolerance is enhanced due to more routing table entries and the relaxed routing selection policy.

References

Bai, H.H., Jiang, J.J., Wang, W.N., 2005. ChordPlus: a scalable, decentralized object location and routing algorithm. *Journal of System Engineering and Electronics*, **15**(4): 772-779.

Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I., 2001. Wide-area Cooperative Storage with CFS. Proceedings of the 18th ACM Symposium on Operating Systems Principles. Chateau Lake Louise, Banff, Canada, p.202-215.

Jiang, J.J., Wang, W.N., 2004. Text-Based P2P Content Search Using a Hierarchical Architecture. Proceedings of the 7th International Conference of Asian Digital Libraries. Shanghai, China, p.429-439.

Mesaros, V., Carton, B., van Roy, P., 2003. S-Chord: Using Symmetry to Improve Lookup Efficiency in Chord. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03). Las Vegas, Nevada, USA, p.1752-1760.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S., 2001. A Scalable Content-addressable Network. Proceedings of ACM SIGCOMM 2001. San Diego, CA, p.161-172.

Ratnasamy, S., Shenker, S., Stoica, I., 2002. Routing Algorithms for DHTs: Some Open Questions. Proceedings of the 1st International Workshop on Peer-to-Peer Systems. Cambridge, MA, USA, p.45-52.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M.F., Dabek, F., Balakrishnan, H., 2003. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, **11**(1): 11-32.

Zhuang, L., Zhou, F., 2003. Understanding Chord Performance and Topology-aware Overlay Construction for Chord. Project Report, available at http://www.cs.berkeley.edu/~zl/doc/chord_perf.pdf.