Implementation of chord P2P protocol using bidirectional finger tables

Submitted by :
Devendra Kalia, Pravesh Ramachandran, Yang Li, Shashank Chaudhary
Code Link : http://www.cse.msu.edu/~chaudh34/

1. INTRODUCTION [1][2][3]

Peer-to-peer is a network communications model in which each node has equal participation in the network and either node can initiate a communication session to carry out a transfer or sharing of data or files. Sometimes in peer to peer networks each node acts as a client and a server. It means it can act both as a source and a recipient. In recent usage, peer-to-peer has come to describe applications in which users can use the Internet to exchange files with each other directly or through an intermediate server. In a P2P model, however, there is no centralized server, but rather an interconnected network of peers. Each node can request data or files from any other node at any other point in time. The concept of P2P networks is that there is no centralized server here, all the data and the information is stored in a distributed manner on all nodes.

In structured peer-to-peer networks, connections in the overlay are fixed. They typically use distributed hash table-based (DHT) indexing, such as in the Chord system (MIT). Unstructured peer-to-peer networks do not provide any algorithm for organization or optimization of network connections.

There are three models of unstructured P2P which have been defined as of now. In the first type of unstructured P2P network, i.e. pure peer-to-peer systems the entire network consists of only peers with equal potential. Hence there is only one routing layer, since there are no special or high priority nodes with any special infrastructure function. Hybrid peer-to-peer systems allow such infrastructure nodes to exist, it means they allow special nodes with high priority which are termed as supernodes. The third kind of unstructured P2P systems are centralized peer-to-peer systems, in which there is a central server is used for indexing functions and to load and establish the whole system of unstructured nodes. The first prominent and popular peer-to-peer file sharing system, Napster, was an example of the centralized model. Gnutella and Freenet, on the other hand, are examples of the decentralized model. Kazaa is an example of the hybrid model.

We will primarily deal with structured P2P networks. Structured P2P network employ a globally consistent protocol to ensure that any node can search and route any other node that contains the required file, even when the file is located at some distant point in the network. Such a guarantee necessitates a more structured pattern of links containing path details. Distributed hash table(DHT) is a structured P2P network, in which a variant of consistent hashing is used to assign storage details of each file

Distributed Hash Tables

Distributed hash tables are distributed systems that provide lookup to a hash table. Every hash table consists of (key,value) pair and any node can efficiently retrieve the value associated with a key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. DHTs can be used to build big networks because they are distributed evenly across all nodes. This in a way increases the number of nodes that can be in a network.

LITERATURE STUDY

Chord protocol [1]

A major issue with peer to peer applications is that they are not able to efficiently locate the nodes containing a particular data item. To solve this chord protocol has been developed which is a distributed lookup protocol. Its primary function is to map a given key onto a node. Chord is a scalable protocol.

Chord provides fast distributed computation of a hash function mapping keys to nodes related to them. Chord uses consistent hashing to provide keys to nodes. Whenever a new node enters the system the keys are evenly distributed to all the nodes thereby maintaining a

well distributed load. Since a chord node stores information about some of the other nodes located close to it, So chord protocol is scalable. All this information is stored in a distributed manner, so each node receives the hash value from other nodes.

Now the primary concern is that how do we uniquely identify every node. An m-bit identifier is used in such a case wherein every node's IP address is hashed into an m-bit identifier and a key is hashed into an m-bit identifier.
This hashing is done using a hash function. As described earlier the primary function of a hash function is to produce an m-bit identifier. This process is known as consistent hashing.

Consistent hashing performs the function of assigning keys to the nodes in the following manner. All the identifiers are arranged in an identifier circle modulo 2^m. Now the keys are assigned to the nodes. This is done by comparing the identifier of a key with the identifier of a node. The key gets assigned to a particular node whose identifier's value is more than that of the key. The selected node will be termed the successor of the the assigned key. It is also denoted as successor(k), which means successor of key k. Assume that identifiers are represented in a circle of numbers starting from 0 to 2^m-1, then successor(k) will be the first node when we traverse in a clockwise direction from k.

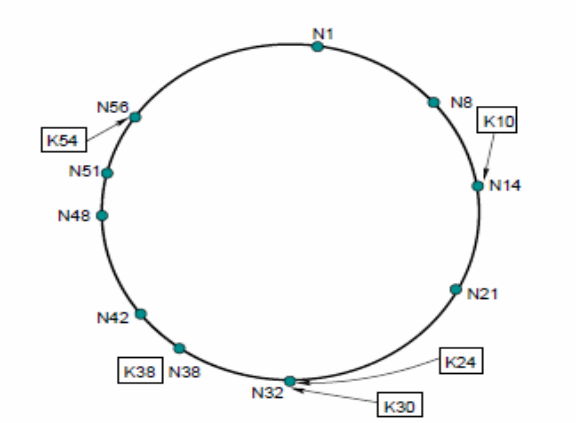


Figure1. An identifier circle (ring) consisting of 10 nodes storing five keys [1]
"Figure 1 shows a Chord ring with m = 6. The Chord ring has 10 nodes and stores five keys. The successor of identifier 10 is node 14, so key 10 would be located at node 14. Similarly, keys 24 and 30 would be located at node 32, key 38 at node 38, and key 54 at node 56" [1].

Chord lookup algorithm [1]

Generally in chord protocol lookup is performed when each node tries to ask its successor about the key. The basic concept in chord protocol is that each node has an entry about its successor. So it has to go on traversing the chord ring one node after the other to search for the key. Now this is a time taking process. So there is a method to make the chord protocol scalable. Chord protocol maintains a finger table to search for the key in a more time efficient manner.

Let us assume that the number of bits in the identifier be m. A routing table is maintained by every node which contains a maximum of m entries. This table maintained by each node is called the finger table. The finger table contains a number of entries, where the ith entry contains a mapping to the node's first successor f which succeeds the node let us say n by at least 2^{i-1} on the identifier circle, i.e., f = successor($n+2^{i-1}$). We call node f the ith finger of node n. It is denoted as n.finger[i]. The finger table can be under stood more clearly from the figure shown below. Basically it has two entries, one for the identifier and the other for the IP Address of a particular node.

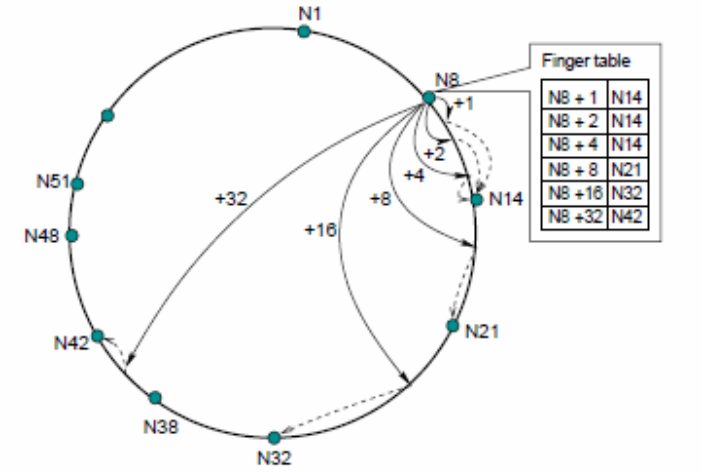


Figure2. Finger table entries for each node in chord ring [1]

"The Figure 2 shows the finger table for node N8. The first finger of node 8 points to its successor which is node 14, because node 14 is the first successor which succeeds $(8+2^0)$ mod 2^6 = 9. Similarly, the last finger of node 8 points to node 42, as node 42 is the first node which is the successor of $(8 + 2^5)$ mod 2^6 = 40" [1]. It is evident from the figure that the first finger for every node is its first successor.

More insight can be gained on the finger tables from the table given below. It gives a clear idea of the definition of fingers and the respective successors and predecessors for each node.

Notation	Definition
finger[k]	first node on circle that succeeds $(n + 2^{k-1}) \bmod 2^m$, $1 \leq k \leq m$
successor	the next node on the identifier circle; *finger*[1].*node*
predecessor	the previous node on the identifier circle

Finger Table definition [1]

The working of chord protocol is quite simple. Whenever a new node enters in the system, it requests another node to find a successor for it. When a new successor is found for it, it is then assigned to it as its successor. Now the successor also knows that the new node is its predecessor. But the previous predecessor of the new node's successor does not know whether a new node as been added to the system. For this each node is automatically refreshed after a certain amount of time. During this node each node asks its successor and predecessor about any changes to know whether a new node has been added or removed from the existing system. This is where it comes to know that the configuration has changed. So the previous predecessor of the new node's successor knows about its successor's new predecessor. Then it stores the new nod as its successor. The new node which was unaware of its predecessor now realizes that it has this node as its predecessor. This is how the system is managed.

Whenever a node fails or randomly leaves the system the keys stored on that node are evenly distributed to other nodes. But to maintain the order each node maintains separate table where it stores the entries of next n nodes succeeding it or preceding it. It keeps on sending messages in a sequential order to every other node in the table until it gets a reply from a node. The node which sends a reply to it first in order to what is stored in the entries is stored as its new successor and that node stores it as its successor.

Limitations in chord protocol

Since the existing protocol maintains a finger table in which the search is done in a sequential manner in the clock wise direction there is an issue with the time taken to search a node. Consider a case where a key to be searched is located on the node which is at the end of the ring when scanned in a clockwise direction. This lookup would take a lot of time, thereby reducing the efficiency of the chord protocol. So it becomes very important to reduce the amount of time needed to lookup for a node in a less amount of time. This is an important aspect which must be taken into consideration because the primary aim of any protocol is the efficient and fast lookup of nodes containing keys.

2. OBJECTIVES

The project aims at addressing some of the shortcomings in the original CHORD algorithm. Specially the way it lookups for a particular key in the CHORD ring. It only searches it in clockwise direction. This single direction search not only increases the time for finding the particular key, but more routing messages needs to be passed in the P2P network. Searching in both the direction will reduce the time required to find a particular key in the ring. This is even more efficient when the key lies in the searching nodes predecessor. In the normal algorithm the whole CHORD ring would be traversed, but in bidirectional lookup this will be faster and will require lesser hops.

3. DESIGN

This project tries to overcome some of the limitations with CHORD protocol stated in previous sections. The project implements a bidirectional finger table, which would reduce the lookup time for a particulra node. In original CHORD all the lookup messages are passed in a clockwise manner along the CHORD ring, which is inefficient. For example to lookup a key located near but preeceding the node, the lookup messages will have to traverse almost the whole CHORD ring.

At the core of the project is to implement a anti finger table for each node along with this already existing finger table. Simply said the finger table stores the successors and their mapped keys for a particular node, the anti finger table will have the list of predecessor nodes of that particular node. As finger table links to nodes in clockwise direction, the anti finger table will link the nodes in anti-clockise direction. For example when there is a link in finger table of original CHORD form node A to node B, we add a reverse link from node B to node A. These reverse links which are in anticlockwise direction form a revrse finger table for that node. Having said this there should be no change in the way the data objects are stored. All the data objects are still located at the successor of their keys.

So each node in modified CHORD protocol maintains:

- Finger Table
- Successor List
- Anti finger Table

While the previous two are the same as they were in the original CHORD protocol.

3.1 Example

In the given figure Node 0 has predecessors node 6 and node 7 both of these nodes have node 0 in their finger

table. So the anti-finger table of node 0 will contain the fingers of node 6 and node 7 which are its predecessor The original unidirectional links from node 6 and 7 to node 0 are both added by a symmetric link.

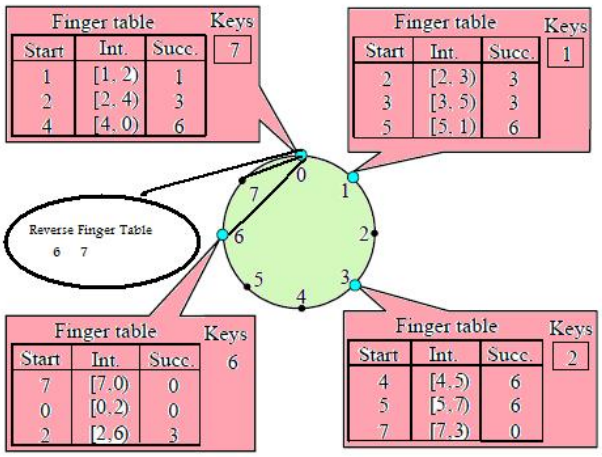


Figure Anti Finger Table

3.2 Lookup Algorithm

The lookup algorithm needs to be modified as we want to preserve the key and data mappings intact. The way original CHORD works is that it looks for a particular key's successor. But while using the anti-finger table, we store the node's predecessors so we cannot use the same algorithm to get the keys. Whenever we are using anti-finger table, the keys will not be at the keys successor but its predecessor. This is important to maintain the key-data mappings of the original protocol.

Using anti-finger in some cases we won't be required to go through the entire lookup routing procedure. We can check the anti finger table to lookup if we have a node in it whose value is greater than key. Then we can use that node to retrieve the data associated with that key.

If a particular key was not found in that nodes anti-finger table entry, we can go through the normal lookup procedure as we did in the original CHORD algorithm. But as we will also be searching in the anti-clockwise direction as we do in clockwise direction, we will find the key earlier than earlier algorithm as we are searching it in both the directions.

4. IMPLEMENTATION

The two main implementation metrics we discuss here is the original clockwise Chord routine based on OpenChord, and the modification of program to fit the feature of bi-directional routine. Moreover, we add some features for each program to weight the performance of each routine algorithm.

The OpenChord platform is an opensource platform for Chord algorithm simulation. We use MyEclipse 8.0 to develop the, the source code can be compiled to a .bat file, we can launch the command line to see the effects of modification made in the source code.

Since the basic construction of the finger table remains the same as original Chord, we do not need to modify the declaration of finger table definition. The number of entries in the finger table is m at most, and the size of finger table is O(logN), where N is the total number of nodes in the network.

Also, each nodes maintains an anti-finger table, which has m entries stored at most, the size of anti-finger table is O(logN). In the BiChord lookup algorithm, when a node send a search message to look up the desired key, if there is no entries in the finger table and anti-finger table which is closer than itself, the node is therefore the predecessor or successor of the key. Otherwise, algorithm will check the routing table(fingers and anti-fingers), then the lookup message will be forwarded to the next hop node which is closer than the current node. The lookup operation is iteratively executed until it finds the node that is preceding or succeeding the desired key.

Since the scale of simulator is small (normally less than 50 nodes), and the simulator is based on Java Virtual Machine(JVM), the lookup speed through the Chord circle is relatively high. If we set a timer to weigh the performance of lookup operation in each algorithm, the best accuracy we can achieve is millisecond. However, after several attempts, we find the timer is not the best way to calculate the cost because it's inaccurate. So we facilitate the hop counter here to calculate the steps of looking up. Since the hops will not be effected by the hardware configuration, the hops counter can explicitly shows the efficiency of the algorithm in this case.
The pseudo code for the bi-directional lookup algorithm is as following :

```
Node findPredecessor(key,n){
Node pred=n.getPredecessor();
if (pred==null)
   return n; //n is the current node
else if (key.isInInterval(pred.ID, n.ID)  //check if the key is between the pred and current node
        return n;
         else {
        Node n'=getClosestPrecedingNode(key) //if not, track the closest preceding node and lookup again
        return findPredecessor(key,n')  // recursively find the predecessor of node n'
```

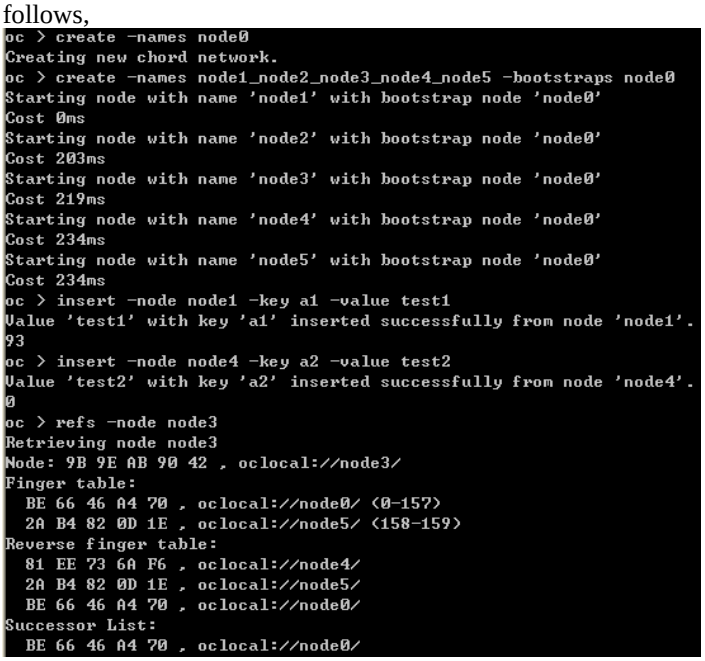
```
        }
}

Node findSuccessor(key,n){
  Node succ=n.getSuccessor();
  if (succ==null)
     return n; //n is the current node
  else if (key.isInInterval(n.ID, succ.ID)  //check if the key is between the current node and successor's node
        return n;
        else {
        Node n'=getClosestPrecedingNode(key) //if not, track the closest preceding node and lookup again
        return findSuccessor(key,n') // recursively find the predecessor of node n'
        }
}

Set<Serializable> retrieve_R(key){
  hops_R=0;  //initialized the hops counter in anti-finger table direction
  whild(!retrieved){
     Node responsibleNode_R=null;
     responsibleNode_R = findPredecessor(id);
     hops_R+=1; //while not retrieve the desired key, add the hop counter by 1
     try{
       result_R = responsibleNode_R.retrieveEntries(id); // get the responsibleNode to fetch the entry
        retrieved = true; //if successfully get the value, set retrieved state to true
     }catch(Exception e){}
      continue;
     }
  }
  if(result_R !=null) values1.add(entry.getValue()); // add the lookup result to the valueset
  final_hopsR=hops_R; //get the hop counter for the current lookup operation
  return values1;
}

Set<Serializable> retrieve(key){
  hops=0;  //initialized the hops counter in finger table direction
  whild(!retrieved){
     Node responsibleNode=null;
     responsibleNode = findSuccessor(id);
     hops+=1; //while not retrieve the desired key, add the hop counter by 1
     try{
       result = responsibleNode.retrieveEntries(id); // get the responsibleNode to fetch the entry
        retrieved = true; //if successfully get the value, set retrieved state to true
     }catch(Exception e){}
      continue;
     }
  }
  if(result !=null) values.add(entry.getValue()); // add the lookup result to the valueset
  final_hops=hops; //get the hop counter for the current lookup operation
  return values;
}
```

It can be seen in the pseudo code that in our modified version of Chord, we set a hop counter for look up operation in clockwise direction and inverse-clockwise direction. When key retrieve operation is executed, system will return a hop value for either searching routine. The screenshot of the program interface is shown as follows,



Create a network and add nodes, insert values

5. EVALUATION

In this section, we evaluate the performance of Bi-Directional Chord by using OpenChord simulator. The algorithms used in this experiment are original OpenChord Chord algorithm and the modified Bi-Directional Chord algorithm. The Chord algorithm is used for comparison.

Experimental Test Results :

Routine Table Size

Number of entries in finger table and anti-finger table can help estimate the total size of routine table. As is mentioned in the implementation section, the number of entries in finger table should be less or equal to the number of nodes m in the network. In the experiment, we simulated a network in OpenChord with [1,2,4,8,16,32,48] nodes respectively. In each size of network, we record the number of nodes, the number of entries in finger table, the number of entries of anti-finger table, the lookup hops in each direction and the total number of entries in routine table.

As expected, the number of entries in routine table is less or equal to 2m, where m is the number of nodes in the network. And the size of the routine table is approximately the total entries number of whole network.

Look Up Hops

To weigh the performance of routine table look up, we pick up a same searching key *(a1)* to search from the same node *(node0)* to execute the look up operation in Chord and BiChord algorithm. We record the hops counter for each size of network tested. The less hops means the algorithm retrieves the node with key in shorter steps.

The test data table and their evaluation description are given as follows :

First of all we have tried to compute the value of number of hops in the existing chord protocol, So the table for that is given below

Number of nodes	Finger table entries	Finger table size	Hops
1	0	0	1
4	2	8	1
8	3	24	1
16	4	64	1
32	5	160	2
48	6	288	2

Table 2. Evaluation table for chord protocol

Now for the new bidirectional finger table evaluation is based on the table given below

Number of nodes	Finger table entries	Finger table size	Reverse finger table entries	Reverse finger table size	Hops	Hops(R)	Total entries
1	0	0	0	0	1	1	0
4	1	4	2	8	1	1	3
8	2	16	3	24	1	1	5
16	4	64	3	48	1	1	7
32	5	160	5	160	2	1	10
48	6	288	5	240	2	2	11

Table 3. Evaluation of bidirectional finger table

As it is evident from both the tables that the new protocol which is implemented using the bidirectional finger table takes a lesser number of hops, when we look at the number of nodes value = 32. But it is not possible to practically evaluate this with a less number of nodes. To get the optimal results we have to test this in an environment where the number of nodes are comparable to a practical network.

6. CONCLUSION

This report proposes bi-directional lookup algorithm based on the OpenChord simulation platform. During designing the algorithm we construct an anti-finger table. From the experiment we know that the lookup performance is slightly improved. We also gave some analysis to show the improvement of the lookup efficiency. In the future work we need to improve the algorithm to implement the function of automatically choosing the search direction by keeping a check on the lookup hops.

7. ACKNOWLEDGMENTS

Our sincere gratitude to Dr. Guoliang Xing invaluable support right through the making of our project without which it would not have been possible to successfully implement this project. We would also like to thank our peers in class for their valuable inputs at each stage of our project. Last but not the least we thank you again for giving us the opportunity to explore the practical aspect of implementing and simulating our protocol.

8. REFERENCES

[1] Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications Ion Stoicay , Robert Morrisz, David Liben-

Nowellz, David R. Kargerz, M. Frans Kaashoekz, Frank Dabekz

[2] Using bidirectional links to improve peer-to-peer lookup performance JIANG Jun-jie†1, TANG Fei-long1, PAN Feng1, WANG Wei-nong2 (1Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200030, China)

[3] Wikipedia : http://en.wikipedia.org

[4] http://compnetworking.about.com/od/p2ppeertopeer/.../p2pintroduction