平成 24 年度修士論文

ピアツーピアアプリケーションのためのBi-Chord オーバレイネットワークの性能向上に関する研究

提出日	平成 24 年 2 月 3 日
所属	名古屋工業大学 大学院 工学研究科 創成シミュレーション工学専攻
指導教官	松尾 啓志 教授
	平成 20 年 4 月入学　22413581 番
氏名	THANGARAJA VIJITHA

EFFICIENT BI-CHORD OVERLAY FOR PEER TO PEER APPLICATIONS

A PROJECT REPORT

Submitted by

THANGARAJA VIJITHA

in partial fulfillment for the award of the degree

of

MASTER OF SCIENCE

in

SCIENTIFIC AND ENGINEERING SIMULATION

NAGOYA INSTITUTE OF TECHNOLOGY

NAGOYA, JAPAN

FEB 2012

NAGOYA INSTITUTE OF TECHNOLOGY
NAGOYA,JAPAN

BONAFIDE CERTIFICATE

Certified that this project report **EFFICIENT BI-CHORD OVERLAY FOR PEER TO PEER APPLICATIONS** is the bonafide work of **THANGARAJA VIJITHA** who carried out this project work under my supervision.

SIGNATURE :

SUPERVISOR : **HIROSHI MATSUO**

Professor,

Department of Scientific And Engineering Simulation,

Nagoya Institute of Technology,

Nagoya, Japan.

Contents

Tables

Figures

1. INTRODUCTION

1.1 Research Background

PEER-TO-PEER (P2P) networks have received tremendous interest in recent years among both Internet users and computer networking professionals. Peer-to-peer computing or networking is a distributed application architecture that partitions the tasks or workloads among peers. Peers are equally privileged, equipotent participants in the application. They are said to form a Peer-to-peer network of nodes. Peer-to-peer technology has become very popular for various applications such as file sharing or live streaming. Today, a large fraction of the internet traffic is due to peer-to-peer applications.

Chord protocol has been suggested as a mandatory underlying overlay technology [1]. In this overlay, each peer maintains a finger table that stores a few successors' connections. Chord routes the message by sending/forwarding messages to the next successor, step by step, until the destination. However, as a protocol originally designed for background downloading applications, Chord owns several disadvantages. Firstly, Chord lookup protocol is based on clockwise lookup. It causes high delay when communicating with peers that are in the anti-clockwise direction. Secondly, Chord uses consistent hashing (e.g. SHA-1, etc) to partition a keyspace so that each peer is responsible for roughly the same

load of resources. However, physically close peers might be assigned with different IDs that are far away from each other in the overlay, and therefore causes longer latency when connected. Thirdly, Chord is implemented in either iterative or recursive style. However, recursive routing might increase the hop number; and iterative routing might be not efficient in traversing NAT . Fourth, Chord lacks of cache mechanism to preserve the useful information for future session establishment. A few attempts have been made to solve the weakness of Chord in P2P communication systems. [2, 3] propose a system model that physically close peers in the overlay are assigned with close peer IDs because most frequently communicated peers are those who are geometry related each other.

In this paper, we study several approaches that could further reduce the communication delay in P2P communication systems. After that, we evaluate these approaches based on the comparison in the aspects: number of hops, and message flow; we get the conclusion that the proposed improved Bi-Chord approach might be a better way for P2P communication systems.

1.2 Research Objective

Main objective of this research is to reduce the response time taken for the lookup service to find the node in the peer-to-peer system. Evaluate the results with the existing protocol in terms of response time, memory and message flow.

1.3 Organization

This thesis is organized as follows. Chapter 2 presents the background of this research, which introduces distributed hash table and about the problems that can occur in the configuration of chord polices.

In chapter 3 introduce the related works of this research. We have did an extensive literature survey regarding the configuration of chord and has been presented in this chapter.

Chapter 4 shows the experimental evaluation of the system with the other methods.. It also explains about the difference between the chord and bi-chord.

Chapter 5 concludes some possible future directions for this work.

2.BACKGROUND

Distributed hash tables are a class of decentralized distributed systems that provide a lookup service similar to hash table. Key, Value pairs are stored in DHT, and any participating node can sufficiently retrieve the value associated with a given 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. This allows DHTs to scale to extremely large numbers of and to handle continual node arrivals, departures and failures.

Chord was proposed in 2001 by Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, and was developed at MIT. Chord is a protocol and algorithm for a peer-to-peer distributed hash table. Chord specifies how keys are assigned to nodes, and how a node can discover the value for a given key by first locating the node responsible for the key.

2.1 Overlay network

An overlay network is a computer network which is built on the top of another network. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. For example, distributed systems such as cloud computing, peer-to-peer networks, and client-server applications

are overlay networks because their nodes run on top of the Internet. Overlay networks have also been proposed as a way to improve Internet routing, such as through quality of service guarantees to achieve higher-quality streaming media. an overlay network can be incrementally deployed on end-hosts running the overlay protocol software, without cooperation from ISPs. The overlay has no control over how packets are routed in the underlying network between two overlay nodes, but it can control, for example, the sequence of overlay nodes a message traverses before reaching its destination.

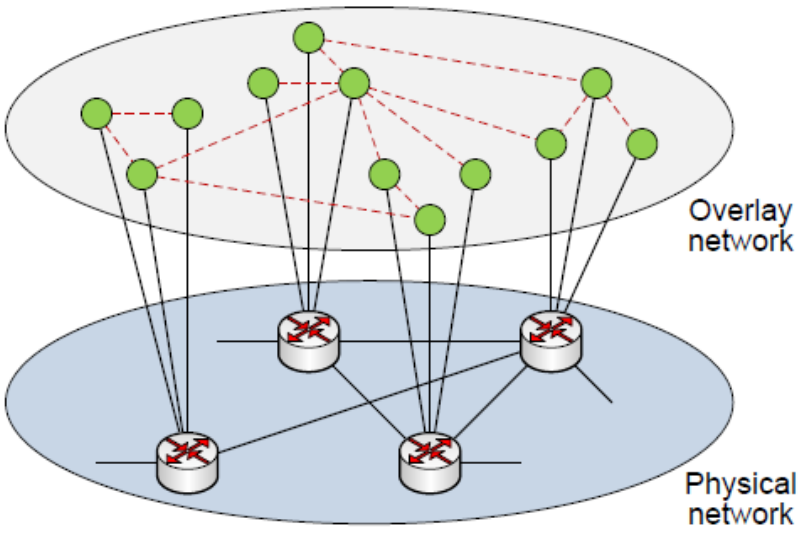

Fig 2.1 Overlay network

The peer-to-peer overlay network consists of all the participating peers as network nodes. There are links between any two nodes that know each other. If a participating peer knows the location of another peer in the peer-to-peer network, then there is a directed edge from the former node to the latter in the overlay

network. Based on how the nodes in the overlay network are linked to each other, we can classify the peer-to-peer networks as unstructured or structured.

2.1.1 Structured System

Structured P2P networks employ a globally consistent protocol to ensure that any node can efficiently route a search to some peer that has the desired file, even if the file is extremely rare. Such a guarantee necessitates a more structured pattern of overlay links. By far the most common type of structured P2P network is the distributed hash table (DHT), in which a variant of consistent hashing is used to assign ownership of each file to a particular peer, in a way analogous to a traditional hash tables assignment of each key to particular array slot.

2.1.2 Unstructured systems

An unstructured P2P network is formed when the overlay links are established arbitrarily. Such networks can be easily constructed as a new peer that wants to join the network can copy existing links of another node and then form its own links over time. In an unstructured P2P network, if a peer wants to find a desired piece of data in the network, the query has to be flooded through the network to find as many peers as possible that share the data.

R

2.2 Distributed Hash Table

A distributed hash table (DHT) is a reliable, scalable, wide area data storage system that frees programmers from many of the complications of building a distributed system [4]. DHTs store blocks of data on hundreds or thousands of machines connected to the Internet, replicate the data for reliability, and quickly locate data despite running over high latency, wide area links.

The DHT addresses problems of locating data and replicating it for reliability, which are common to many distributed systems, without additional work by the application. The DHT provides a generic interface, which makes it easy for a wide variety of applications to adopt DHTs as a storage substrate: put stores data in the system under a key; get retrieves the data. Distributed hash tables fill a gap in the design space of storage systems. DHTs occupy a middle ground between small systems with strong guarantees on the service they provide (such as distributed file systems) and large unorganized, best effort systems (such as the world wide web or file sharing systems). DHTs are able to operate over a large and previously unoccupied area of the design space; the existence of a single, practical system that operates throughout this regime will make it easier to write new distributed applications. In filling this design space gap, DHTs attempt to combine the two strands of systems research that inspired DHTs and provide the best features of both. Inspired by small, LAN-based systems (which we will call transparent distributed systems), DHTs provide probabilistic guarantees on

the success of a get or put operation. At the same time DHTs can operate on the same scales as large systems designed to run on the wide area.

Functions of DHT:

1. put(key, value)
2. get(key)

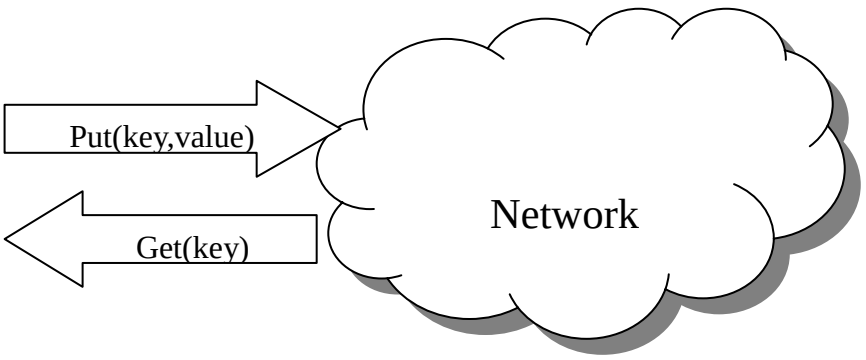

Fig 2.2 Functions of DHT

2.3 DHT properties

DHTs characteristically emphasize the following properties:

1. Decentralization
2. Fault tolerance
3. Scalability

2.4 Chord

2.4.1 Overview

Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. Chord assigns keys to nodes with *consistent hashing* [4], which has several desirable properties. With high probability the hash function balances load (all node receive roughly the same number of keys). Also with high probability, when an Nth node joins or leaves the network, only a *O*(1/N) fraction of the keys are moved to a different location this is clearly the minimum necessary to maintain a balanced load. Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A Chord node needs only a small amount of "routing" information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with other nodes. In an N-node network, each node maintains information about only *O*(log N) other nodes, and a lookup requires *O*(log N) messages.

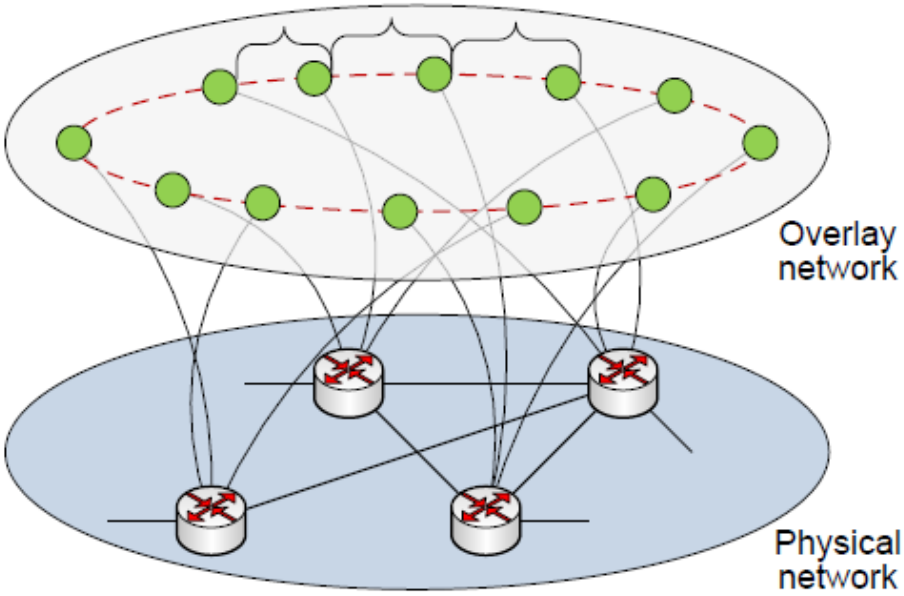

Fig 2.3 Overlay network with Chord ring

2.4.2 Consistent Hashing

The consistent hash function assigns node and key an *m*-bit *identifier* using SHA-1 [5] as a base hash function. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. We will use the term "key" to refer to both the original key and its image under the hash function, as the meaning will be clear from context. Similarly, the term "node" will refer to both the node and its identifier under the hash function. The identifier length m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible. Consistent hashing assigns keys to nodes as follows. Identifiers are ordered on an *identifier circle* modulo 2^m. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the *successor node* of key k,

denoted by *successor*(k). If identifiers are represented as a circle of numbers from 0 to 2^{m-1}, then *successor*(k) is the first node clockwise from k. In the remainder of this paper, we will also refer to the identifier circle as the *Chord ring.*

Consistent hashing is designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hashing mapping when a node n joins the network, certain keys previously assigned to *n*'s successor now become assigned to n. When node n leaves the network, all of its assigned keys are reassigned to *n*'s successor. No other changes in assignment of keys to nodes need occur.

2.4.3 Simple key location

Lookups could be implemented on a Chord ring with little per-node state. Each node need only know how to contact its current successor node on the identifier circle. Queries for a given identifier could be passed around the circle via these successor pointers until they encounter a pair of nodes that straddle the desired identifier; the second in the pair is the node the query maps to.

2.4.4 Scalable Key location

Let *m* be the number of bits in the key/node identifiers. Each node *n* maintains a routing table with up tomentries (we will see that in fact only O(log *n*) are distinct), called the *finger table*. The ith entry in the table at node n contains the identity of the *first* node s that succeeds n by at least 2^{i-1} on the identifier circle, i.e., s = *successor*$(n+2^{i+1})$, where $1 \leq i \leq m$ (and all arithmetic is modulo 2m). We call node s the i^{th} *finger* of node n, and denote it by *n.finger*[i] (see Table 2.1). A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Note that the first finger of n is the immediate successor of n on the circle; for convenience we often refer to the first finger as the *successor*.

Table 2.1 Notation

Notation	**Definition**
Finger[k]	First node on circle $(n+2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
Successor	The next node on the identifier circle
Predecessor	The previous node on the identifier circle

2.4.5 Address the problems

Chord simplifies the design of peer-to-peer systems and applications based on it by addressing these difficult problems:

*1.Load balance***:** Chord acts as a distributed hash function, spreading keys evenly over the nodes; this provides a degree of natural load balance.

*2.Decentralization***:** Chord is fully distributed: no node is more important than any other. This improves robustness and makes Chord appropriate for loosely-organized peer-to-peer applications.

*3.Scalability***:** The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible. No parameter tuning is required to achieve this scaling.

*4.Availability***:** Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change.

*5.Flexible naming***:** Chord places no constraints on the structure of the keys it looks up: the Chord key-space is flat. This gives applications a large amount of flexibility in how they map their own names to Chord keys.

2.5 Chord Overlay

In Chord overlay, peers and resources construct a ring, as shown in Figure 2.4. In the ring, peers and resources are represented by an integer Node ID/Resource ID. Each peer stores a certain amount of <*id*, *value*> pairs, in which

id is the peer/resource ID, *value* is the peer address information or the data storage. Peer/resource ID is assigned by consistent hashing [6], e.g. SHA-1 algorithm. For instance, the peer ID can be produced by hashing the IP address of the particular peer; and the resource ID can be generated by hashing the data value. The Resource ID is stored in the first peer, whose ID>= Resource ID (see Figure 2.4).

Each peer contains a routing table, called Finger table, for storing the routing information records. The Finger table records $\log N$ successors where N is the number of peers in the overlay (see Figure 2.4). Suppose the space size of overlay is 2^m, for some integer m and the i-th successor ID of a peer with ID P is:

$$Succid(i) = (P + 2^{i-1}) \bmod 2^m \quad (0 < i \leq m)$$

Each peer contacts periodically its successors for updating the Finger table. It also contacts the predecessor that is the previous peer in the identifier circle. This is useful when a peer leaves the ring and asks the previous peer to update its Finger table. Chord routes the message by sending messages to the next successor that is nearest to the destination identifier. The total cost is no more than $\log N$ hops and .log N in average where N is the number of peers in the overlay [7].

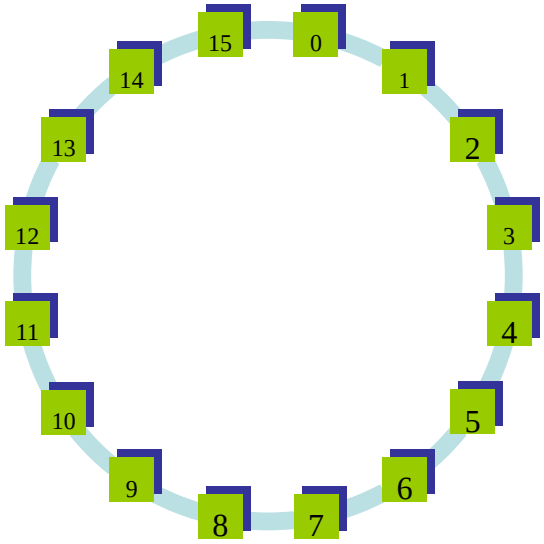

Fig 2.4 : Chord ring

Chord also defines the advertisement function about joining/leaving procedure for peers. The advertisement function would tell the corresponding successor and predecessor to update their finger table.

Table 2.2 :Finger Table

i	Successor
0	4
1	5
2	7
3	11

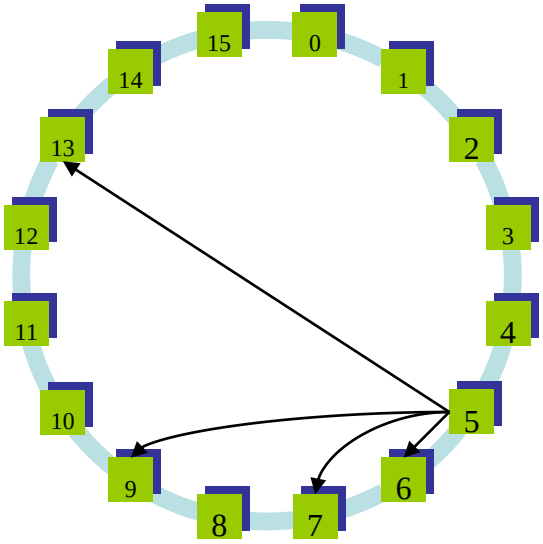

Fig 2.5: Direct connection

2.4.1 Successor function

Since each node maintains information about only a small subset of the nodes in the system, evaluating the successor function requires communication between nodes at each step of the protocol. The search for a node moves progressively closer to identifying the successor with each step. A search for the successor of *f* initiated at node *r* begins by determining if *f* is between *r* and the immediate successor of *r*. If so, the search terminates and the successor of *r* is returned. Otherwise, *r* forwards the search request to the largest node in its finger table that precedes *f*; call this node *s* . The same procedure is repeated by *s* until the search terminates.

Pseudocode to find the successor

// ask node n *to find the successor of* id

n.find successor(id)

if (id ϵ (n, *successor*])

return *successor*;

else

// forward the query around the circle

return *successor*:*find successor*(id);

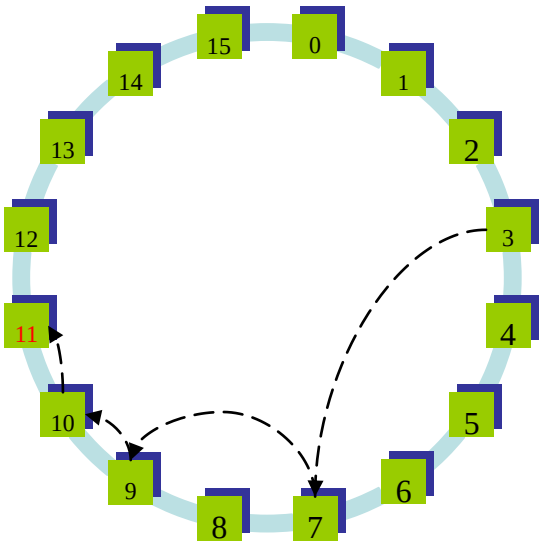

Fig 2.6: Message flow

The algorithm is outlined above in *recursive* form: if a search request requires multiple steps to complete, the step is initiated by the $(n-1)^{th}$ behalf of the initiator. The successor function may also be implemented *iteratively*. In an iterative implementation, the initiating node is responsible for making requests for finger table information at each stage of the protocol. Both implementation styles offer advantages: an iterative approach is easier to implement and relies less on intermediary nodes, while the recursive approach lends itself more naturally to caching and server selection.

2.4.2 Node join

In order to ensure that lookups execute correctly as the set of participating nodes changes, Chord must ensure that each node's successor pointer is up to date. It does this using a "stabilization" protocol that each node runs periodically in the background and which updates Chord's finger tables and successor pointers. Figure 6 shows the pseudocode for joins and stabilization. When node n first starts, it calls *n.join(n0)*, where *n0* is any known Chord node, or *n.create()* to create a new Chord network. The *join*() function asks *n0* to find the immediate successor of n. By itself, *join*() does not make the rest of the network aware of *n*. Every node runs *stabilize*() periodically to learn about newly joined nodes. Each time node *n* runs *stabilize*(), it asks its successor for the successor's predecessor *p*, and decides whether *p* should be *n*'s successor instead. This would be the case if node *p* recently joined the system. In addition, *stabilize*() notifies node *n*'s successor of *n*'s existence, giving the successor the chance to change its predecessor to *n*. The successor does this only if it knows of no closer predecessor than *n*.

create a new Chord ring.

n.create()

predecessor = **nil**;

successor = n;

join a Chord ring containing node *p*.

n.join(n')

predecessor = **nil**;

successor = *p .find successor*(n);

called periodically. verifies *p*'s immediate successor, and tells the successor about n.

n.**stabilize**()

x = *successor*.*predecessor*;

if (x ϵ (n. *successor*))

successor = x;

successor.*notify*(n);

p thinks it might be our predecessor.

n.notify(p)

if (*predecessor* **is nil or** *p* ϵ (*predecessor*. n))

predecessor = *p*;

called periodically. refreshes finger table entries.

next stores the index of the next finger to fix.

n.fix_fnngers()

next = *next* + 1;

if (*next* > *m*)

next = 1;

finger[*next*] = *find successor*(n + 2^{next-1});

called periodically. checks whether predecessor has failed.

n.check predecessor()

if (*predecessor* **has failed**)

predecessor = **nil**;

Each node periodically calls *fix fingers* to make sure its finger table entries are correct; this is how new nodes initialize their finger tables, and it is how existing nodes incorporate new nodes into their finger tables. Each node also runs check predecessor periodically, to clear the node's predecessor pointer if *n.predecessor* has failed; this allows it to accept a new predecessor in notify.

As a simple example, suppose node n joins the system, and its ID lies between nodes n_p and n_s. In its call to *join*(), *n* acquires n_s as its successor. Node n_s, when notified by *n*, acquires *n* as its predecessor. When n_p next runs *stabilize*(), it asks ns for its predecessor (which is now *n*); n_p then acquires *n* as its successor. Finally, n_p notifies *n*, and *n* acquires n_p as its predecessor. At this point, all predecessor and successor pointers are correct. At each step in the process, n_s is reachable from n_p using successor pointers; this means that lookups concurrent with the join are not disrupted. Figure 7 illustrates the join procedure, when *n'*s ID is 26, and the IDs of n_s and n_p are 21 and 32, respectively. As soon as the successor pointers are correct, calls to *find successor*() will reflect the new node. Newly-joined nodes that are not yet reflected in other nodes' finger tables may cause *find successor*() to initially undershoot, but the loop in the lookup algorithm will nevertheless follow successor (*finger*[1]) pointers through the newly-joined nodes until the correct predecessor is reached. Eventually *fix*

fingers() will adjust finger table entries, eliminating the need for these linear scans.

One simple solution is to use two directional lookup mechanism in the search of peer/resource, called Bi-Chord [8]. In this solution, suppose the overlay space is 2^m, each peer stores *m* successor and *(m–1)* predecessor records in its Finger table. The request is sent/forwarded to one of the successors/predecessors that is clockwisely closest to the target and then forwarded step-by-step until the destination is reached.

Suppose the space size of overlay is 2^m, for some integer *m* and the *i*-th predecessor ID of a peer with ID *P* is:

$$Precid(i) = (i + n - 2^{i-1}) \bmod 2^m \quad (0 < i \leq m-1)$$

Table 2.3 Predecessor list for node 3

I	Predecessor
0	2
1	1
2	14

Figure 5 depicts the Bi-chord with successor and predecessors. Solid line represents the connection to the successor where as the dotted line represents the connection with the predecessor. Figure 5 shows the connections of a peer with identifier 5. It holds four connections with its successors (peer 6, peer 7, peer 9,

and peer 13) and three connections with its predecessors (peer 1,peer 3 and peer 4).

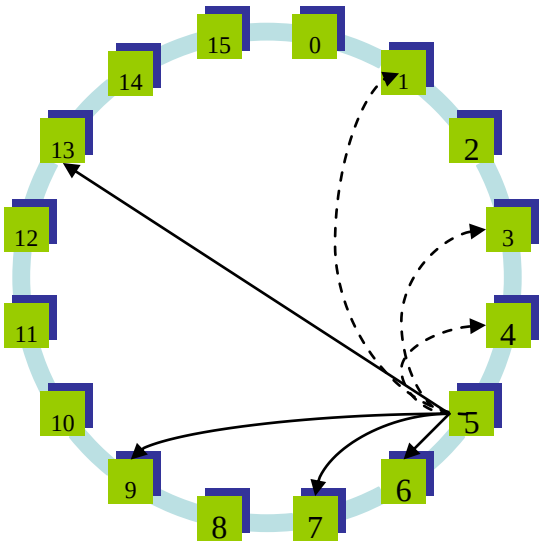

Fig 2.7: Bi-chord

For searching a peer, for instance peer 2, peer 5 firstly send the request to the peer 1; peer 1 , after using its own finger table, forwards the request to the destination peer 2, as represented in Figure 6. This approach reuses most of the Chord lookup algorithm and routing style. According to the algorithm in section 2, it takes (log(*N*)-1) / 2 in average before the message reaches the destination, where N is the number of peers in the overlay.

This approach reuses the most of the Chord lookup algorithm and routing style. According to the algorithm it takes [log (N-1)]/2 in average before the message reaches the destination, where N is the number of peers in the overlay.

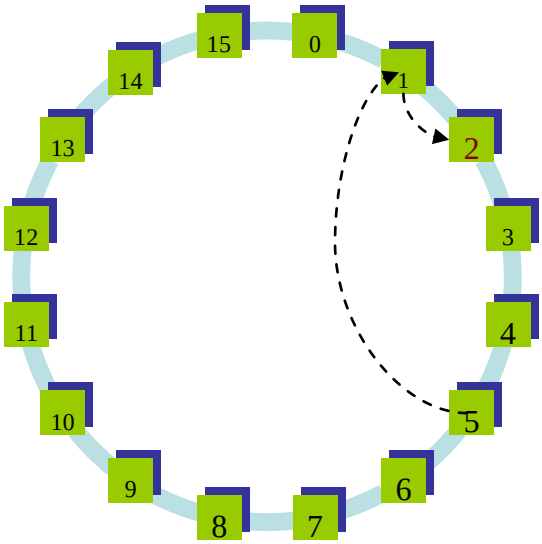

Fig 2.8: Message forwarding in Bi-chord

2.4.3 Path Length

Chord's performance depends in part on the number of nodes that must be visited to resolve a query. Path length is *O*(logN), where N is the total number of nodes in the network. Each node in an experiment picked a random set of keys to query from the system, and we measured each query's path length.

Consider a node making a query for a randomly chosen key. Represent the distance in identifier space between node and key in binary. The most significant (say i^{th}) bit of this distance can be corrected to 0 by following the node's i^{th} finger. If the next significant bit of the distance is 1, it too needs to be corrected by following a finger, but if it is 0, then no i -1^{st} finger is followed—instead, we move on the the i-2^{nd} bit. In general, the number of fingers

we need to follow will be the number of ones in the binary representation of the distance from node to query.

Table 2.4 Path length

Chord	*O* log(N)
Bi-Chord	log(N-1)/2

3. APPROACH

In this chapter the proposed complex bi-chord lookup method is explained with routing style of chord with some modifications and compare message flow and response time with the existing algorithm.

3.1 Complex and efficient Bi-Chord

This algorithm may be implemented to further enhance Bi-Chord efficiency. In this approach, each peer maintains the same finger table as Bi-Chord (Figure 8). The major difference is that each peer transmits the P2P request to one of its successors/predecessors that is as close as possible to the destination, independent of the clockwise or anti-clockwise direction. Each peer chooses either its successor or predecessor for routing messages, on the distance basis.

Suppose peer A wishes to initiate/forward a P2P message to the destination B, it chooses the shorter distance of

(a) One of its predecessors closest to peer B.

(b) One of its successors closest to peer B.

(c) If these two choices have equal path lengths, the message will follow the rule b.

Figure 3.1 shows an example of the proposed lookup initiated from peer 3 and ended in peer 12.The message is firstly routed to peer 10 (peer 3's successor), then to peer 13 (successor of peer 10), and finally to peer 12 (predecessor of 13). The average path length of this method is

$$\log(N) / 2 - \log(N / 2\pi) + 1$$

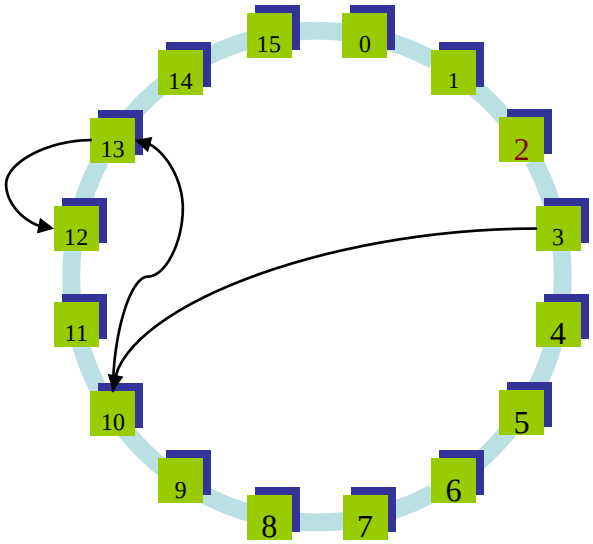

Fig 3.1 Lookup in proposed method

3.2. Cache Entry Record

We can also use a cache entry record approach to improve performance indirectly. The concept is as following: P2P peer in the overlay maintains a cache that records the communication history details (see Table 3.1), including the previous communicated peer identifier, the corresponding public IP address, port, etc. For searching the destination peer, source peer first check its cache entry

record. If the destination peer (peer identifier, public IP address, port, etc) is in the table, the session might be established directly. Otherwise, the source peer will execute the lookup algorithm described above.

In the stable overlay where peers do not change the identifier and the public endpoints frequently, the cost is only one hop. However, in the unstable overlay where peers change their identifier/IP frequently, this might even cost worse delay. It takes at most (log N +1) hops (e.g. Bi-Chord lookup) before reaching the destination.

Table 3.1 Cache entry record

id == 3	Identifier	Public endpoint: port
1	A	215.239.168.1:1980
2	B	159.250.16.2 :2000
. .	. .	. .
N	S	128.39.169.2 : 9000

3.3 Impact of Node Joins on Lookups

In this section, we consider the impact of node joins on lookups. We first consider correctness. If joining nodes affect some region of the Chord ring,

a lookup that occurs before stabilization has finished can exhibit one of three behaviors.

The common case is that all the finger table entries involved in the lookup are reasonably current, and the lookup finds the correct successor in O(logN) steps. The second case is where successor pointers are correct, but fingers are inaccurate. This yields correct lookups, but they may be slower. In the final case, the nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail.

The higher-layer software using Chord will notice that the desired data was not found, and has the option of retrying the lookup after a pause. This pause can be short, since stabilization fixes successor pointers quickly. Now let us consider performance. Once stabilization has completed, the new nodes will have no effect beyond increasing the N in the O(logN) lookup time. If stabilization has not yet completed, existing nodes' finger table entries may not reflect the new nodes. The ability of finger entries to carry queries long distances around the identifier ring does not depend on exactly which nodes the entries point to; the distance halving argument depends only on ID-space distance. Thus the fact that finger table entries may not reflect new nodes does not significantly affect lookup speed. The main way in which newly joined nodes can influence lookup speed is if the new nodes' IDs are between the

target's predecessor and the target. In that case the lookup will have to be forwarded through the intervening nodes, one at a time. But unless a tremendous number of nodes joins the system, the number of nodes between two old nodes is likely to be very small, so the impact on lookup is negligible. Formally, we can state the following result. We call a Chord ring *stable* if all its successor and finger pointers are correct.

Theorem : If we take a stable network with N nodes with correct finger pointers, and another set of up toN nodes joins the network, and all successor pointers (but perhaps not all finger pointers) are correct, then lookups will still take *O*(logN) time with high probability.

Proof : The original set of fingers will, in *O*(logN) time, bring the query to the old predecessor of the correct node. With high probability, at most *O*(logN) new nodes will land between any two old nodes. So only *O*(logN) new nodes will need to be traversed along successor pointers to get from the old predecessor to the new predecessor. More generally, as long as the time it takes to adjust fingers is less than the time it takes the network to double in size, lookups will continue to take *O*(logN) hops. We can achieve such adjustment by repeatedly carrying out lookups to update our fingers. It follows that lookups perform well so long as ($\log_2$ N) rounds of stabilization happen between any N node joins.

3.4 Failure and Replication

The correctness of the Chord protocol relies on the fact that each node knows its successor. However, this invariant can be compromised if nodes fail. For example, in Figure 4, if nodes 14, 21, and 32 fail simultaneously, node 8 will not know that node 38 is now its successor, since it has no finger pointing to 38. An incorrect successor will lead to incorrect lookups. Consider a query for key 30 initiated by node 8. Node 8 will return node 42, the first node it knows about from its finger table, instead of the correct successor, node 38. To increase robustness, each Chord node maintains a *successor list* of size r, containing the node's first r successors. If a node's immediate successor does not respond, the node can substitute the second entry in its successor list. All r successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of r. Assuming each node fails independently with probability *p*, the probability that all r successors fail simultaneously is only p^r. Increasing r makes the system more robust. Handling the successor list requires minor changes in the pseudocode in Figures 5 and 6. A modified version of the *stabilize* procedure in Figure 6 maintains the successor list. Successor lists are stabilized as follows: node n reconciles its list with its successor s by copying *s*'s successor list, removing its last entry, and prepending *s* to it. If node n notices that its successor has failed, it replaces it with the first

live entry in its successor list and reconciles its successor list with its new successor. At that point, n can direct ordinary lookups for keys for which the failed node was the successor to the new successor. As time passes, *fix fingers* and *stabilize* will correct finger table entries and successor list entries pointing to the failed node. A modified version of the *closest preceding node* procedure in Figure 5 searches not only the finger table but also the successor list.

4.EVALUATION

In this chapter the modeling of the complex and efficient bi-chord is done. The results are compared with existing algorithms.

4.1 Lookup

The results are obtained for each node in the network and tried for all the combination of keys. For example if the network sine is N then the search is done for N-1 time for each and every node. Response time is the average of the search results time. Fig 4.1 shows the search results done for the three approaches. Based on the Chord lookup protocol described in Chapter 2, the average complexity of Chord lookup algorithm is .log(*N* / *S).* Bi-Chord provides fairness in bi-directional peer/resource lookup.

Thus, the complexity is .log(*N* / *S*). Proposed method further improves the lookup efficiency. According to the revision from previous research (in Chapter 3), the average complexity is: log(*N* / *S*) / 2 - log(*N* / 2p*S*) +1. We compare three lookup algorithms by setting different S value (e.g. we set S = 8 and S = 16 for example), as represented in Figure 15. X axis represents the peer number in the overlay and Y axis represents the number of hops in average. We get the information that firstly, Bi-Chord and proposed lookup algorithms are much more

efficient than the original Chord lookup; secondly, the higher value of S, the smaller number of hops; thirdly, proposed approach provides better result than Bi-Chord.

Table 4.1 Response time for the lookup method

No.of.Peers	Chord	Bi-chord	Proposed Method
16	0	0	0
32	0.0000375	0.0000375	0
64	0.0005391	0.0005984	0
128	0.000625	0.000625	0.000313
256	0.0030469	0.0012109	0.000703
512	0.0095313	0.0062695	0.005723
1024	0.0349414	0.0223633	0.018457
2048	0.210249	0.1986768	0.17063

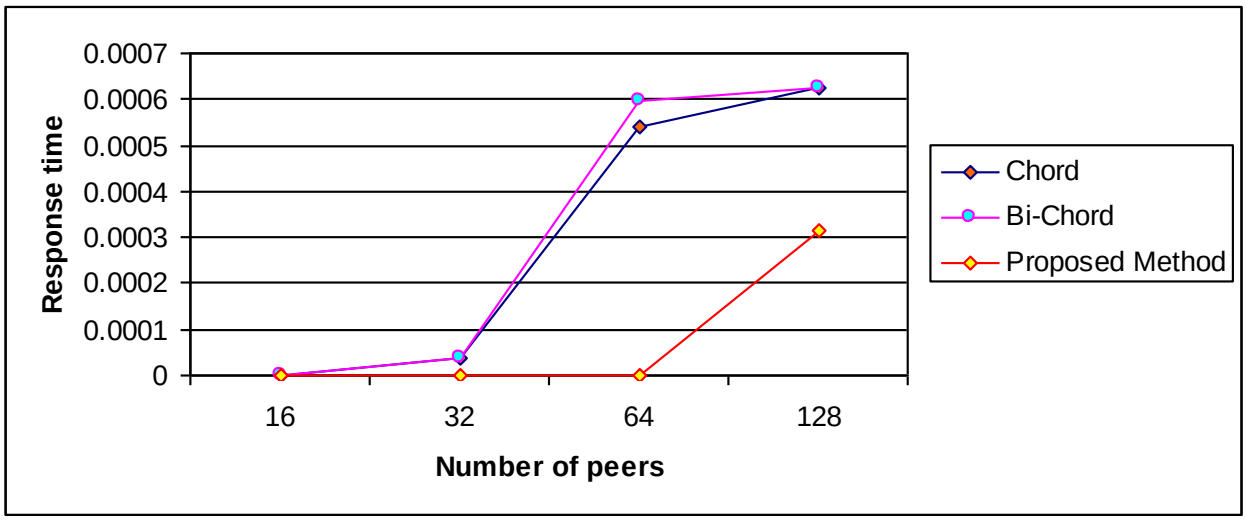

Fig 4.1(b) Experimental results

The graph explicitly shows that the response time for a lookup service in chord is long. Bi-chord gives better results compared to Bi-chord ie the response time is less than the chord. As expected the proposed complex and efficient method gives better results. i.e the response time for the lookup service is comparatively less compared to the existing one. For the number of peers 16, three methods give the same result. The response time is near to zero, it is very less. From the size 32, the proposed method give better results, the graph is linearly increased according to size. The proposed method utilizes the time efficiently. Figure 4.2 shows the result for maximum number of peers used in the network. According to figure 4.2, if the network size is increased the response time also will increase, but the proposed method give comparatively good result.

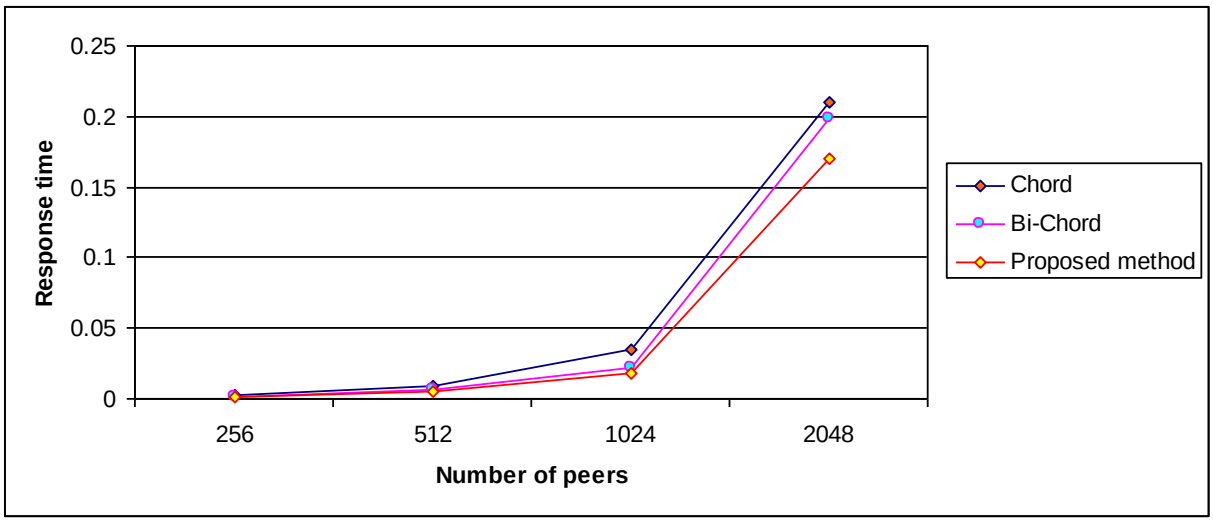

Fig 4.1(b) Experimental results

4.2 Message flow

For the chord it will take *O*log(N) message flow. If the node receive the request it will forward the request to the closest successor if the node did not find the successor in the finger table of the node. Message forwarding continues until the destination is found. For a node, it will take maximum *m-1* message flow. Path length for chord *O*log N is proved from the result. Fig 4.3 shows the evaluation of message flow. The x-axis denotes the number of bits used for the id whereas y-axis denotes the message flow in average.

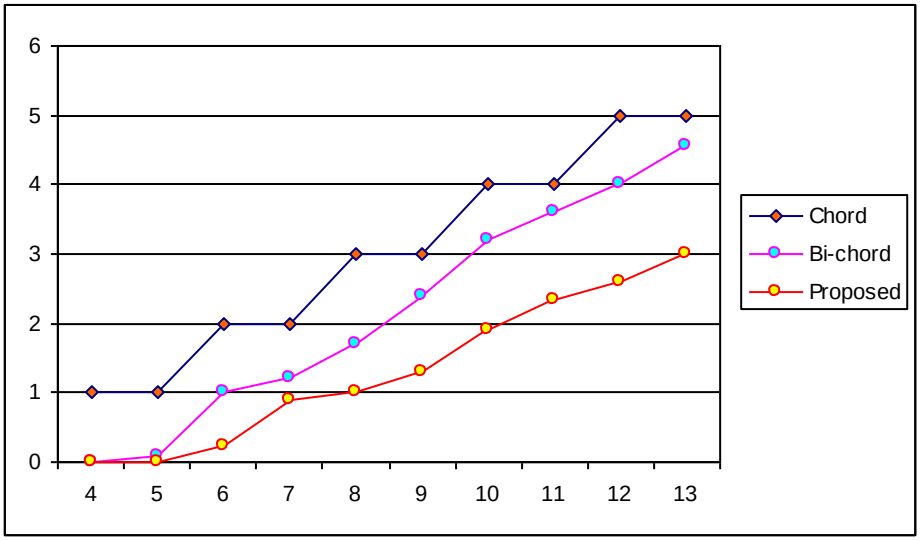

Fig 4.2 Message flow

Figure 4.2.b show the message flow result in the worst case, ie the path length of the lookup service is long. According to the graph, proposed method worst case message flow is less in average.

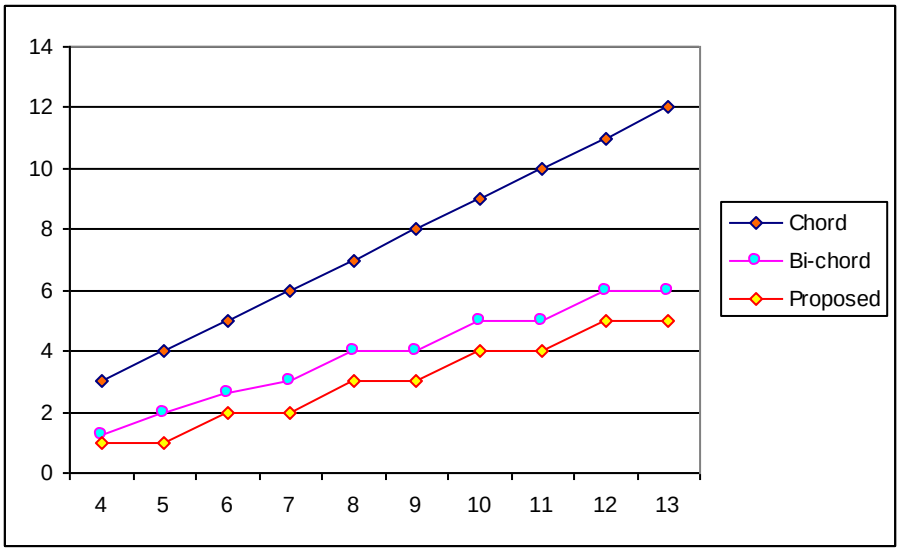

Fig 4.3 Worst case Message flow

4.3 Maintenance cost

Chord algorithm maintenance cost is less because it has only *m* successor list of finger table, so it is easy to maintain the chord algorithm with minimal cost. Algorithm checks the first and last entry of the finger table at the beginning. If the key is smaller than the first entry then it will give first key as the result. Whereas the key is bigger than the last entry then it provides the last key as a result. Otherwise it will check the each entry of the finger table, when it found the closest preceding node it will stop the lookup service. This lookup is easy to implement and maintain.

In Bi-chord each node keeps the finger table with *m* successors and *m-1* predecessor. Searching a key involves the both successor and predecessor entries. When a node leaves the network the node have to update the finger table entries.

Bi-chord also verify that whether predecessor or successor list have the closest preceding node, it will take more cost to maintain the finger table as well as the lookup service.

In the proposed method, the algorithm have to calculate the distance with both successor and predecessor. Moreover it have to check whether the distance is fell under 2^m and if there are more than one distance fall under the 2^m then it have to find the minimum distance to proceed further. If the node whose distance fall under 2^m does not contain the key values then the lookup service select second minimum distance and it will continue the lookup until found the destination. Total cost of the this method is comparatively bigger than the bi-chord approach.

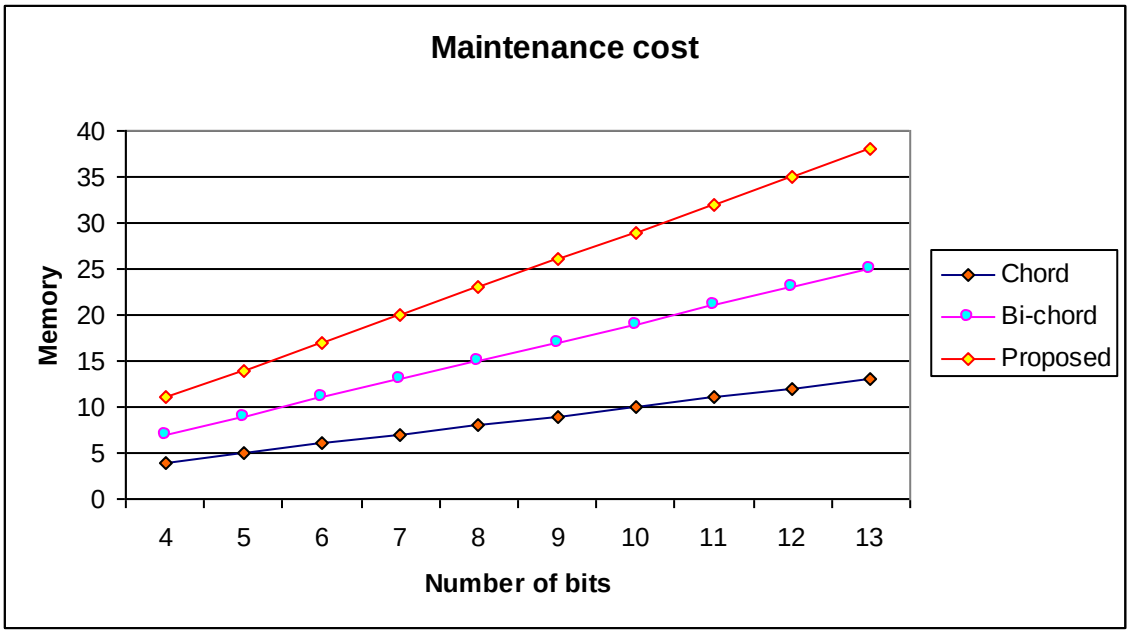

Fig 4.4 Maintenance cost

5. CONCLUSION

The proposed algorithm works well for the overlay network peer to peer applications. . The lookup in this method is very east to find the peer in the network. However it follows the routing style of chord algorithm it will take less time to search the node with maximum memory utilization. It can solve any queries to find the node. But it will take more memory compared to existing algorithm. If there is alternative for the optimized memory usage to reduce the memory capacity, this method will be the best lookup service to find the peer in the network.

Acknowledgment

At first, I would like to take this opportunity to thank Japan Ministry of Education, Culture Sports, Science and Technology (MEXT) for awarding me the Japanese Government Monbukagakkusho Scholarship. I am much honored to be the recipient for this award. Receiving this scholarship motivates me to maintain a peaceful life and providing me confidence and willingness to help me achieve my goals.

I would like to take this opportunity to thank my respected professor Dr. Hiroshi Matsuo for giving me this wonderful opportunity. I would also like to record my gratitude to him for his supervision, advice, and guidance from the very early stage of this research as well as giving me extraordinary experiences through out the work. Whenever I go to meet my professor with depression, I will come back with motivation because of him. I am very lucky to work with my professor.

At the foreign student's centre of Nagoya Institute of Technology, I would like to express my gratitude to all the staff members for helping me with various aspects from daily life in Japan to learning the Japanese language itself. I would

like to thank especially Associate Professor, Mrs. Yamamoto, International centre for automotive research, for her kind support and helping in my Japanese Language. I would also like to thank Imai san for helping me to lead life in Japan in my entire study curriculum.

My special thanks go to my uncle Dr. Arulkumaran who introduced about Nagoya Institute of Technology to pursue my higher studies. I would like to thank my cousin Mrs.. Subana Thanasegaran for being introduced into my lab.

It is a pleasure to express my gratitude wholeheartedly to Dr. Venkatesan, Mr. Sridharbabu's family for their kind hospitality during my stay in Japan.

I would like to thank my labmates Mr. Hatta, Mr. Yoshida, Mr. Tsubasa , Mr. Sobue , Mr. Mizuno , Mr. Kumazaki , Mr. Asai , Mr. Ikegaya for their moral support towards my Japanese language, job hunting and research in Japan.

My parents deserve special mention for their inseparable support and prayers. I would like to sincerely thank My Father Mr. Thangaraja who put the fundament my learning character, showing me the joy of intellectual pursuit ever since I was a child and My Mother Mrs. Manonmani Thangaraja is the one who sincerely raised me with her caring and gentle love. I would like to express my

gratitude to my beloved sisters for being so affectionate and loving. Though very faraway they have stood by me and encouraged me in all my ventures. I would like to thank my friends Mr. Ganapathy and Mr. Thirumal Azhagan for their support and they always suggest me a good solution for any problems.

Finally, I would like to thank everybody who was important to the successful realization of thesis, as well as expressing my apology that I could not mention personally one by one.

References

[1] Ion Stoica, Robert Morris , David Liben-Nowell, David R.Karger, M.Frans Kaashoek, Frank Dabek , Hari Balakrishnan : *Chord: a scalable peer-to-peer lookup protocol for internet applications.*IEEE/ACM Transactions on Networking., 2003.

[2] Guangyu Shi, Y.L., Jian Chen, Hao Gong, Hongli Zhang, *T2MC: A Peer-to-Peer Mismatch Reduction Technique by Traceroute and 2-Means Classification Algorithm*, in *7th International IFIP-TC6 Networking Conference*, May 5-9, 2008: Singapore.

[3] Huang, L., *Location and Discovery of Subsets of Resources.* Internet-Draft , July,2008:

[4] Frank Dabek , *A Distributed Hash Table* Submitted to the Department of Electrical Engineering and Computer Science on Nov 2005,MIT.

[5] KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed

caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM*

[6]*Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654–663. FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[7] David, K., et al., *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web*, in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997, ACM: El Paso, Texas, United States.

[8] Jani, H. and C. Gonzalo, *Evaluation of DHTs from the viewpoint of interpersonal communications*, in *Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*. 2007, ACM: Oulu, Finland.

[9] Junjie Jiang, R. Changyong Liang and Weinong Wang, *BiChord: An Improved Approach*
for Lookup Routing in Chord. Lecture Notes in Computer Science 2005: p. http://www.springerlink.com/content/egj6fpw0w8e136tm/.

[10] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger Robert Morris, Ion Stoica, Hari Balakrishnan at MIT Laboratory for Computer Science, *Building Peer-to-Peer SystemsWith Chord, a Distributed Lookup Service*.

[11] Zhongmei Yao, and Dmitri Loguinov, *Understanding Disconnection and Stabilization of Chord* , IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 22, NO. 4, APRIL 2011

[12] http://:www.slideshare.net/did2/chorddht?from=share_email_logout3

[13] Mudhakar Srivatsa and Ling Liu, *Mitigating Denial-of-Service Attacks on the Chord Overlay Network:A Location Hiding Approach*, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 20, NO. 4, APRIL 2009

[14] http://p2pfoundation.net/Distributed_Hash_Table

[15] http://www.infoanarchy.org/en/Distributed_Hash_Table