

COEN 241 Term Project  
Chord Protocol Enhancement and Analysis  
for  
Peer to Peer System

Submitted By : Team 6

Shraddhaben Padariya,

Nidhi Shah,

Deepak Rengasamy Thirumeni Palanivelu,

Kushal Gosalia

Instructor :

Prof. Ming Hwa Wang

Santa Clara University

# Preface

This project analyses the efficiency of the chord protocol, identifies the problem faced in the chord protocol and proposes a more efficient method which can overcome the drawback faced in the chord protocol.

# Acknowledgement

We would like to thank Dr. Ming-Hwa Wang for teaching us about the topics we used in this project.

<b>1. Introduction</b>	<b>5</b>
1.1. Objective	5
1.2. What is the problem	6
1.3. Why this is a project related to this class.	6
1.4. Why other approach is no good	6
1.5. Why you think your approach is better	6
1.6. Statement of the problem	6
1.7. Area or scope of investigation	7
<b>2. Theoretical bases and literature review</b>	<b>7</b>
2.1. Theoretical Background Of The Problem	7
2.2. Related research to solve the problem	8
2.3. Advantage/Disadvantage of those research	9
2.4. Your solution to solve this problem	9
2.5. Where your solution different from others	9
2.6. Why your solution is better	10
<b>3. Hypothesis (or goals)</b>	<b>10</b>
3.1. Positive/negative hypothesis	10
<b>4. Methodology</b>	<b>10</b>
4.1. How to generate/collect input data	10
4.2. How to solve the problem	10
4.2.1. Algorithm design	11
4.2.2. Language Used	16
4.2.3. Tools Used	16
4.3. How to generate output	16
4.4. How to test against hypothesis	17
<b>5. Implementation</b>	<b>17</b>
5.1. code (refer programming requirements)	17
5.2. Flowchart	18
<b>6. data analysis and discussion</b>	<b>22</b>
6.1. output generation	22
6.2. output analysis	24
6.3. compare output against hypothesis	25
6.4. abnormal case explanation (the most important task)	27
<b>7. conclusions and recommendations</b>	<b>27</b>
7.1. summary and conclusions	27
7.2. recommendations for future studies	28
<b>8. Bibliography</b>	<b>28</b>
<b>9. appendices</b>	<b>29</b>

9.1. program source code with documentation	29
9.2. input/output listing	29

# 1. Introduction

## 1.1. Objective

The main objective of this paper is to analyse the traditional chord protocol and to address the problem associated with it by offering a new scalable modified chord protocol which can run more efficiently in a dynamic distributed environment.

## 1.2. What is the problem

Native Chord Protocol is an efficient lookup algorithm in a peer to peer system and also handles efficiently when a node joins or leaves the system. However, The native chord protocol in some cases for instance where it has to find a key which resides in the neighbour of preceding node. The path taken to find the key in the neighbouring preceding node is quite long since it has to iterate the whole ring even though the node which holds the key it resides next to the node which requested for a lookup of the key.

## 1.3. Why this is a project related to this class.

This project aims to optimize the lookup algorithm in P2P System, which is one of the core part of the P2P system. By improving on the lookup time we can increase the overall running efficiency of the P2P System.

## 1.4. Why other approach is no good

The approach used by the existing chord protocol results in a high lookup latency when the key we are searching for lies in the second half of the ring. Doing this way would increase the network traffic in the system by involving more machine to lookup a key which ultimately increases the lookup time of a key.

## 1.5. Why you think your approach is better

In our approach, we are making two changes in chord algorithm to improve search efficiency. First, we will make modify the chord protocol in such a way that it will be able to iterate in both clockwise and anticlockwise direction. Each node contains two finger table; regular finger table and anti-finger table. When a search request comes to any node it checks weather receiver node is in  $2^{(m-1)}$  from self. If yes, then it gives it to regular finger table to handle else it will give it to anti-finger table to handle. Second, we are using cache to make algorithm more search efficient for highly popular data by serving the request in a constant time. When a search data request comes, node checks if data key is in cache the then it find appropriate receiver ip and port and send request

directly to that receive else query request goes through finger table. Backend process automatically update cache periodically to make it more cost-effective.

## 1.6. Statement of the problem

Chord Protocol is used to provide efficient lookup of a key with a complexity of  $O(\log n)$ . However, In some cases we analysed that the search time and the number of nodes involved in finding a key can be reduced by using an additional data structure - anti finger table and revolving in both clockwise and anticlockwise direction. This was possible at the cost of extra space. Also, we have made it more efficient by implementing the Least Recently Used(LRU) cache on top of our modified algorithm.

## 1.7. Area or scope of investigation

In this project, we focus our attention on improving the efficiency of the lookup latency in chord protocol.

# 2. Theoretical bases and literature review

## 2.1. Theoretical Background Of The Problem

Chord is a protocol and set of algorithms which enable fast and efficient look up operation in Distributed Hash Tables (DHTs) for peer to peer systems. A Distributed Hash Table has key-value pairs and assigns keys to different nodes (computers) in the network. A node will store all the key values for which it is responsible. Chord will define a way to implement lookup of these values and map the node when a value (keyword) is specified. It also gives instructions on how a node will discover the value for given key, by first locating the node where the key is stored.

The lookup operation using Chord protocol increases the speed and efficiency by the following method. Whenever a keyword is specified, the protocol converts it to binary equivalent of the key. Once that number is available, the protocol puts the assigned key into the successor node of the hashing id. Consider an example: If the keyword to find is "Happy Birthday", whose binary equivalent is 24. Now, in the Chord ring if the successor node in clockwise direction is 32, the protocol will save that keyword in node 32. So, when lookup takes place, it knows exactly where the key is stored looking at the binary equivalent of the key.

### **Consistent Hashing**

The core operation of Chord is to provide fast distributed computation of hash function to map keys to nodes responsible for storing it. Chord uses consistent hashing which has some desirable unique properties to assign data key to nodes. Chord uses SHA-1 hash function to assign unique identifier to all nodes which are part of a distributed system and to all data keys. Because of this hashing mechanism, whenever a Nth node join the system or existing one leave the system then

only  $O(1/N)$  fraction of keys are moved to different location which clearly shows the minimum requirement to maintain a balanced load among all nodes.

Consistent hashing maps data keys to nodes as follows : All nodes with their unique identifiers are represented as an ordered list arranged in a circle modulo  $2^m$  known as Chord Ring. Whenever a new data request (key K) comes ,then Key K would assign to the first node whose identifier is equal to or follows L in identifier space. This node is called the successor node of Key K ,denoted by  $successor(k)$ . If nodes along with their unique identifiers are represented on a circle numbered from 0 to  $2^m - 1$ , then  $successor(k)$  is the first node from clockwise from K.

### Properties of chord

1. Decentralization  
Chord algorithm would work for fully distributed system as it consider all nodes equal. This will improves its robustness and makes it appropriate for loosely organized peer-to-peer applications.
2. Availability  
Chord automatically adjusts its internal lookup tables to reflect newly joined nodes or the one which left the system to ensure to that key would always found despite continuous state of changes.
3. Scalability  
Chord protocol avoids the need to store information all of all nodes by storing only limited  $\log(N)$  nodes information. Such a way, higher scalability can be achieved.
4. Load balance  
Chord uses consistent hashing (SHA-1 hash function) to distribute all the data keys evenly among participating nodes to maintain load balance among them.
5. Flexible Naming  
Chord does not limit any restriction regarding the structure of keys it looks up.

## 2.2. Related research to solve the problem

To improve chord protocol, we read few papers related to chord protocol enhancement and below is facet of all those papers.

First paper which we read was 'Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications' in which chord protocol concept has been introduced first. This paper gives an overview about the concept,consistent hashing,finger table concept etc. It has also included the working mechanism of chord protocol and how it run efficiently in a dynamic environment by periodically updating its routing information.



Second paper 'Preserving Receiver's Anonymity for Circular Structured P2P networks' was about improving chord protocol using tunneling and public key/private key encryption to secure data transmission. In that new approach, when any node in chord gets data request it calculates hash value of data to search on which node data resides. It sends query packet to destination node that includes data to search, public key of self and ID of node(final) to which destination will send query result data (encrypted using public key of sender). So in such way it preserves sender's anonymity. To get data from final node to sender node tunnel mechanism is used. Tunnel can have one or more nodes included. Data passed through tunnel is encrypted so no other node in tunnel can read data. When sender node receives data it decrypts the query result using its private key and show results to user.

Another important paper that we reviewed was titled "Research and Improvement in Chord". The paper begins by explaining the working of the Chord algorithms and protocol. Then the authors move on to propose a solution to increase the lookup efficiency in the Chord ring. According to the authors, all the nodes in the P2P system are named using a random ID generator assigned to it using a Hashing algorithm. The proposal was instead of a random ID for each node, there should be a structured way to name the nodes in the network. They divide the structured ID in two parts: 1)Prefix - includes the physical location of the node in the network. 2)Suffix - includes the random ID assigned to it using Hashing algorithms. They provide an analogy to the way administrative districts are divided - Province, District, City, Random. Naming the nodes this way, increases the lookup speed in the DHT used within the network.

### 2.3. Advantage/Disadvantage of those research

In paper 'Preserving Receiver's Anonymity for Circular Structured P2P networks', author used tunnel and encryption mechanism. It makes data transmission very secure but use of tunnel to return query result to sender will increase hop counts and indirectly increase network traffic. As number of node in tunnel increase, network traffic increase. It will also make network less scalable because of that problem. In this paper , data security is achieved at cost of extra network traffic.

For the paper "Research and Improvement in Chord Protocol", the authors suggest using Structured node IDs instead of Random ID generated for each node using Hashing Algorithm. The advantage of using this method is that it improves the lookup efficiency and reduces the stretch rate. Stretch Rate is defined as the ratio of logical/physical position of node to the smallest distance between source and destination nodes. On the other hand, a major disadvantage of using structured node IDs is that each node in the network has to be named with four parts, which increases the memory used and is also tedious to assign unique IDs.

### 2.4. Your solution to solve this problem

Our Solution to the problem, is to implement the chord ring lookup bidirectional. In order to implement this solution we are making use of two finger tables. One of them is used to search nodes in the anti clockwise direction of the chord ring while the other searched the node in clockwise direction.

## 2.5. Where your solution different from others

Our solution differs from others in the sense that, all the papers we studied and read had either implemented the traditional algorithm directly or had ideas to improve the speed of lookup in P2P network using Chord Protocol but none of the paper which we read had addressed the improvement of search efficiency. Our solution is unique because the implementation of two finger tables has not been tried in these papers and also we are making the chord ring bidirectional. Each node maintains two finger tables, one is a normal finger table while the other is anti-finger table. According to our solution, whenever a key is specified, the protocol will take that value, compare it to half the number of nodes on the system and if the key is more than that it will implement lookup in the anti clockwise direction using the anti finger table. If not, then it will search in the finger table of the node and save the assigned key to successor in clockwise direction.

## 2.6. Why your solution is better

This is a better solution because, it increases the speed and efficiency of lookup operation in a Distributed Hash table of P2P system networks. Moreover, each node in the system is only required to save the information of its successor and its precursor. Thus it also requires minimal memory usage. We have also provided a solution to use the chord ring bidirectionally instead of unidirectional search. This reduces the number of hops to store or lookup a key assigned to the node.

# 3. Hypothesis (or goals)

## 3.1. Positive/negative hypothesis

We are storing two finger tables(regular finger table, anti- finger table) at each node of network, making chord network bidirectional will decreases the number of hop counts in search operation.

We will also provide caching feature to make search more efficient by storing the result of data container node ip and port of most frequently searched query request. So use of cache in each node will decrease search time.

# 4. Methodology

## 4.1. How to generate/collect input data

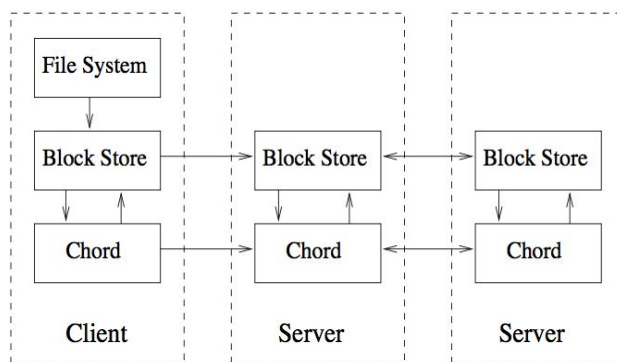
We will take all input data request from text file at regular interval from each host. We will log the latency time and number of hop counts need to traverse to serve each data requests in a log file.

## 4.2. How to solve the problem

As mentioned in section 2.1, Chord is a structured look up protocol usually used for peer to peer distributed system. The main goal of chord is to provide a way to look up the location of nodes based on the unique key (identifiers). This means it is used to locate the node on which the desired data item has been stored. It provides this without using any centralized topological network and tries to provide good load balancing, scalability and availability. This section would describe topological structure of traditional chord algorithm and then it will discuss the problems associated with it and propose a new scalable modified chord algorithm to improve search efficiency.

### 4.2.1. Algorithm design

#### Topological structure of peer to peer system using chord

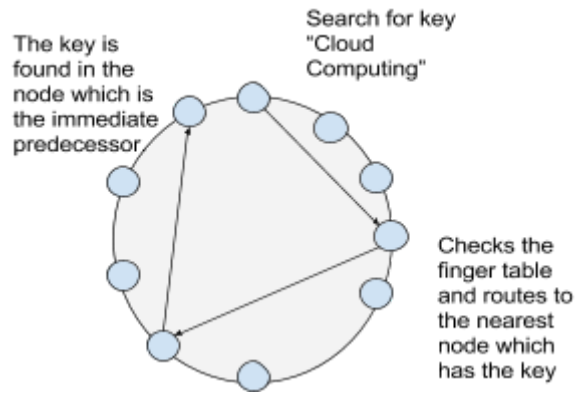


The fundamental structure of any typical peer to peer system using chord has been shown in above figure. The highest level is responsible for implementation of application specific functions such as file system meta-data. The next level (block store) implements a general purpose distributed hash table that multiple application use to insert and retrieve data blocks identified by unique key. This layer also takes care of storing, caching and replication of blocks. Last layer implements chord protocol and it specifies how to find the location of keys, how new nodes join the system, and how to recover from failure of existing nodes.

#### Drawbacks of Native Chord Protocol

The chord protocol maintains the finger table in such a way that the lookup of a key is done in a sequential manner. The ultimate aim of the chord protocol is to efficiently reduce the lookup time of the key's value.

For example, Consider a case where the key is resided in the node at the end of the ring. So to search for the key, Chord algorithm has to iterate in clockwise direction and the last node in the ring is the node which has the key. This ultimately reduces the lookup efficiency.



From the diagram we understand that even though the key is in the neighbour node the chord protocol iterates the whole ring to look up the key.

### Modified Chord Protocol:

#### Proposed new Lookup algorithm by using anti-finger table

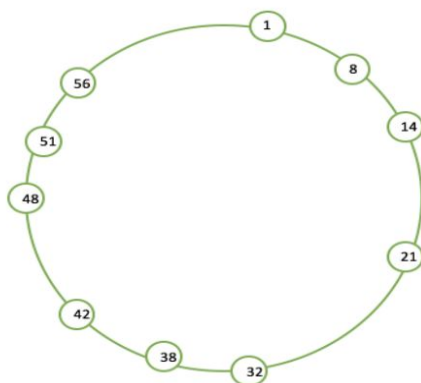
##### Idea :

To overcome the above mentioned problem we come up with an algorithm which efficiently looks up the key in both direction i.e., both clockwise and anticlockwise direction.

For example , To lookup a key located near but preceding the node, the native chord protocol lookup will traverse almost the whole chord ring but in our proposed method we can just traverse in anti clockwise direction which will reduce the distance travelled to find the key and the number of messages is reduced between the peers to lookup a key.

##### Construction of anti finger table :

To traverse in anti clockwise direction we should know the node's predecessors. So to maintain the node's predecessors we make use of one more data structure to store the node's predecessors. We call this data structure as anti finger table. As finger table contains links to nodes in clockwise direction, the anti finger table should contains link to the nodes in anti-clockwise direction.



Finger Table			Anti finger table				
1	Key	Span	Node	1	Key	Span	Node
		2 [2,3)	8			0 [63,0)	1
		3 [3,5)	8			63 [61,63)	1
		5 [5,9)	8			61 [57,61)	1
		9 [9,17)	14			57 [49,57)	1
		17 [17,33)	21			49 [33,49)	51
		33 [33,2)	38			33 [0,33)	38
8	Key	Span	Node	8	Key	Span	Node
		9 [9,10)	14			7 [6,7)	8
		10 [10,12)	14			6 [4,6)	8
		12 [12,16)	14			4 [0,4)	8
		16 [16,24)	21			0 [56,0)	1
		24 [24,40)	32			56 [40,56)	56
		40 [40,9)	42			40 [7,40)	42
32	Key	Span	Node	32	Key	Span	Node
		33 [33,34)	38			31 [30,31)	31
		34 [34,36)	38			30 [28,30)	32
		36 [36,40)	38			28 [24,28)	32
		40 [40,48)	42			24 [16,24)	32
		48 [48,0)	48			16 [0,16)	21
		0 [0,33)	1			0 [31,0)	1
38	Key	Span	Node	38	Key	Span	Node
		39 [39,40)	42			37 [36,37)	38
		40 [40,42)	42			36 [34,36)	38
		42 [42,46)	42			34 [30,34)	38
		46 [46,54)	48			30 [22,30)	32
		54 [54,6)	56			22 [6,22)	32
		6 [6,39)	8			6 [37,6)	8
14	Key	Span	Node	14	Key	Span	Node
		15 [15,16)	21			13 [12,13)	14
		16 [16,18)	21			12 [10,12)	14
		18 [18,22)	21			10 [6,10)	14
		22 [22,30)	32			6 [62,6)	8
		30 [30,46)	32			62 [46,62)	1
		46 [46,15)	48			46 [13,46)	48
21	Key	Span	Node	21	Key	Span	Node
		22 [22,23)	32			20 [19,20)	21
		23 [23,25)	32			19 [17,19)	21
		25 [25,29)	32			17 [13,17)	21
		29 [29,37)	32			13 [5,13)	14
		37 [37,53)	38			5 [53,5)	8
		53 [53,22)	56			53 [20,53)	56
42	Key	Span	Node	42	Key	Span	Node
		43 [43,44)	48			41 [40,41)	42
		44 [44,46)	48			40 [38,40)	42
		46 [46,50)	48			38 [34,38)	38
		50 [50,58)	51			34 [26,34)	38
		58 [58,10)	1			26 [10,26)	32
		10 [10,43)	14			10 [41,10)	14
48	Key	Span	Node	48	Key	Span	Node
		49 [49,50)	51			47 [46,47)	48
		50 [50,52)	51			46 [44,46)	48
		52 [52,56)	56			44 [40,44)	48
		56 [56,0)	56			40 [32,40)	42
		0 [0,16)	1			32 [16,32)	32
		16 [46,15)	21			16 [47,16)	21
51	Key	Span	Node	51	Key	Span	Node
		52 [52,53)	56			50 [49,50)	51
		53 [53,55)	56			49 [47,49)	51
		55 [55,59)	56			47 [43,47)	48
		59 [59,3)	1			43 [35,43)	48
		3 [3,19)	8			35 [19,35)	38
		19 [19,52)	21			19 [50,19)	21
56	Key	Span	Node	56	Key	Span	Node
		57 [57,58)	1			55 [54,55)	56
		58 [58,60)	1			54 [52,54)	56
		60 [60,0)	1			52 [48,52)	56
		0 [0,8)	1			48 [40,48)	48
		8 [8,24)	8			40 [24,40)	42
		24 [24,57)	32			24 [55,24)	32

Suppose, a search request comes at node 1, it calculates where search data resides and get node 43. Now let's get routing path from node 1 to node 43 using regular chord protocol and enhanced chord protocol using above finger-table and anti-finger table.

For original chord protocol, query request goes from node 1 to node 38, node 38 to node 42 and node 42 to node 48 (where data of node 43 stored). It takes 3 hops to reach from node 1 to node 43.

For enhanced chord protocol, node 1 calculates half distance node which is node 33 in this example. Node 43's distance is greater than node 33 distance, so node 1 use anti-finger table to reach to node 43. Using anti finger table, query request goes from node 1 to node 51, node 51 to node 48. It takes 2 hops to reach from node 1 to node 43. This way enhanced chord protocol reduces hop counts for this example.

However, this doesn't change how the keys are stored. Keys are located at the immediate successor node just like in native chord.

### **Modified lookup algorithm :**

The lookup algorithm of native chord searches for the node which is the immediate key successor. But when we use the anti finger table where it stores the node's predecessors. So while using anti finger table, the key and the value will be stored in key's predecessor rather than key's successor.

Using the anti finger table in some cases can we don't have the need to iterate the whole ring. If the key we want to find is not in the anti finger table then we make use of the finger table. So this way we make use of both the table taking clockwise and anti clockwise direction. So doing this way the lookup of the key is faster than the native by reducing the number of hops in the network which ultimately reduces the network traffic in the system.

### **Algorithm to improved search efficiency by adding caching**

We will add LRU (least recently used) caching to surpass efficiency of current chord protocol in terms of time to search query data. To implement caching we will use hashmap of key-value pairs where key is data string and value is an object which will store reference count of data which will indicate how recently it has been used along with ip address and port number of node that is responsible to hold that data key. This function checks periodically whether data has been used in any fixed time frame  $t_1$ ; if it has been used it increases its reference count of that data key by 1. It also reset count value of all keys stored in cache after some fixed time to eliminate effect of old dominant data key.

Whenever any node gets a data query request from user/other nodes, it first searches its cache to find data in it. If cache contains data, it writes data key in temp storage  $s_1$  and send query request directly to the node whose ip and port is stored in cache. After time interval  $t_1$  (as

mentioned in above passage), it takes all keys stored in  $s_1$ , update count value of those keys by 1 and make storage  $s_1$  empty. If cache doesn't contain data, sender node sends query request to receiver node. Once it gets response query data, it finds key with least count value and remove that key value from hashmap and add newly searched data with count value 1 and receiver's ip, port.

This way, we can make search more efficient and respond most popular data request in a constant time  $O(1)$  by implementing cache. If any data query is frequent, cache contains receiver node's id and port, so sender will directly contact that node using cache and need not to use finger table search mechanism.

## **Dynamic Operations and failures**

This section describes the behavior of chord algorithm whenever a new node joins/leaves the system or system crashes.

### **(1) New node joins and stabilization**

In order to ensure that lookups execute correctly as the state of participating nodes changes continuously, 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. Above figure shows pseudocode for joins and stabilization. As we can see, whenever a new node  $n$  join the system it would call  $n.join(n')$  where  $n'$  is known chord node. This function asks  $n'$  to get the information about the predecessor of new node  $n$  which would be stored by new node  $n$ . Every node in chord would run stabilization procedure in the backend periodically to learn about the newly joined nodes. Each time a node runs  $stabilization()$  procedure and notifies its successor about its existence as a predecessor. If successor node has changed its predecessor information then in return it would send information about the new node. Hence eventually each node would update their successor and predecessor pointers correctly.

### **(2) Fix\_fingers() and check\_predecessor()**

As mentioned in above pseudocode, each participating node would run  $fix\_fingers()$  and  $check\_predecessor()$  procedure in a background periodically.  $fix\_fingers()$  process to make sure that their finger table entries are correct. This way all newly joined nodes would get a chance to initialize their finger tables and all the existing nodes would get a chance to incorporate new nodes into their finger tables.

Each node would also run  $Check\_predecessor()$  periodically, and clear the node's predecessor pointer if node's predecessor has been failed.

### **(3) Existing node leave the system (Voluntarily exit)**

Whenever an existing node  $n$  voluntarily leave the system would transfer all of its keys to its successor before it departs. Node  $n$  may notify its predecessor  $p$  and successor  $s$  before leaving. As a

result, node p will remove n from its successor list, and add the last node in n's successor list to its own list. Similarly, node s will replace its predecessor with n's predecessor.

### **Data Operations to be performed:**

#### **Store Data :-**

In Modified Chord protocol which is presented in this paper, the approach to store a data key would work same as native algorithm. It will use consistent hashing to get the unique key of data and use the same to identify the node responsible for storing the data key. The Node which will get the request to store the data key k first, it will first check its own range and store it in local machine if it is responsible to store the data other wise it will simply pass the request to closest successor node till it reaches to the destination node.

#### **Search Data :-**

The lookup algorithm of this improvised Chord protocol need to be modified compared to native chord protocol as we want to preserve the key and data mapping intact. As mentioned in section (2), the way original Chord works is that it looks for a particular key's successor by using finger table. But here while using anti-finger table, we store the node's predecessors so we cannot use the same algorithm to get the keys. Whenever we use anti-finger table, we won't need to go through the entire lookup routing procedure as traditional approach. We can check the anti-finger table to find 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 we don't find a particular key by using anti-finger table entry (This case could happen under dynamic environment. Whenever a node join/leave the system and we got the search request before stabilization process update the information), we will go through the normal lookup procedure as we mentioned in section 2. By using this approach of anti-finger table we will find the key earlier than traditional algorithm as were searching it in both the directions.

#### **Delete data :-**

In this modifier chord protocol mentioned in this paper, partial operation of delete operation which is to find the data would be same as search operation as mentioned in previous paragraph. After finding the desired data, we will delete the same from the system.

### **4.2.2. Language Used**

The whole project will be developed by using JAVA programming language.

### **4.2.3. Tools Used**

We have used the eclipse IDE to implement our project.



## 4.3. How to generate output

We will run traditional chord algorithm and our modified chord algorithm with anti-finger table on a distributed environment and observe the number of hop count each request is taking to complete. We will also observe the search time of each request. In the end, we will plot a graph of hop counts and search time to show our analysis.

## 4.4. How to test against hypothesis

We will divide our testing phase into three parts which are mentioned below :

### (1) Routing table size

As mentioned in section (3) we are storing two tables (finger table, anti-finger table) on each node in modified chord algorithm. We will fetch the data of space that has been used by each node and compare against traditional chord approach.

### (2) Number of hop count :

As mentioned in previous section that we are adding anti finger table to reduce the hop counts that need to be traversed to serve our data request. We will test this part by keeping a count in a temporary variable and incrementing it whenever a node route query request to other nodes. We will plot a graph of this count to compare both approaches.

### (3) Latency time :

As mentioned in section (3) that we are implementing a caching feature to make search more efficient. To prove our result, we will log the latency period of each request and plot the results in graph to show our analysis.

# 5. Implementation

## 5.1. code (refer programming requirements)

We have made use of Java Programming language for the full project implementation. Eclipse IDE is used to write, compile and test the code in local machines. The code includes implementation for Chord protocol and our modified bidirectional searching Chord protocol. The details of the coding is as below:

### (1) I/O details:

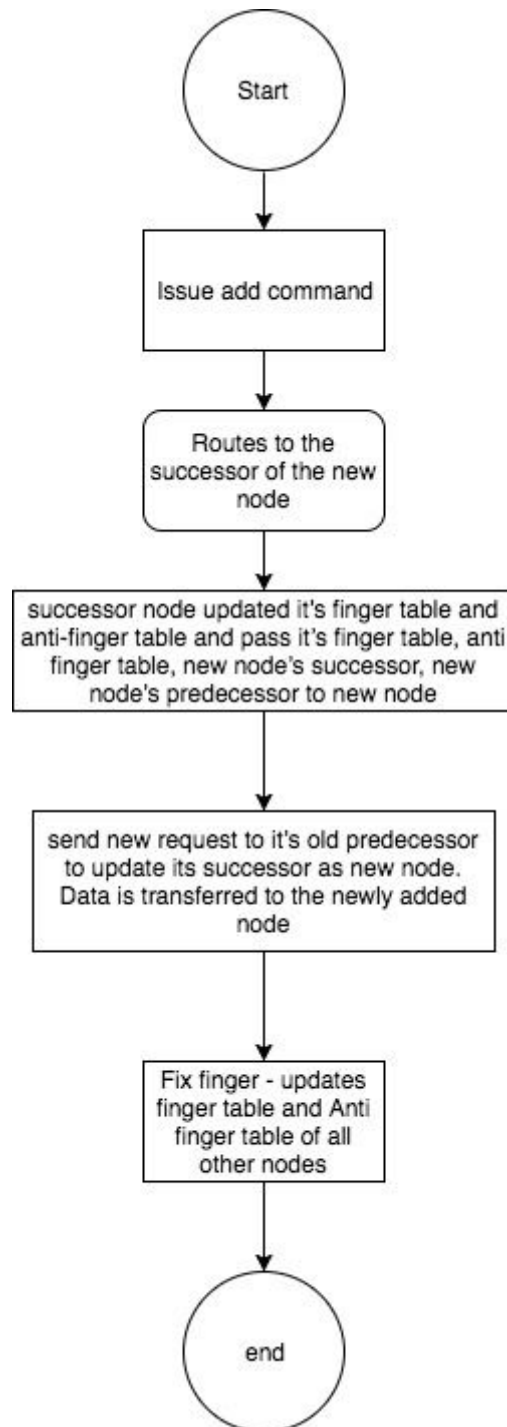
Our code takes in standard input from the keyboard, and output is generated on the machine's monitor. Various "print" instructions have been used to demonstrate the functionality of the project.

### (2) Code Debug:

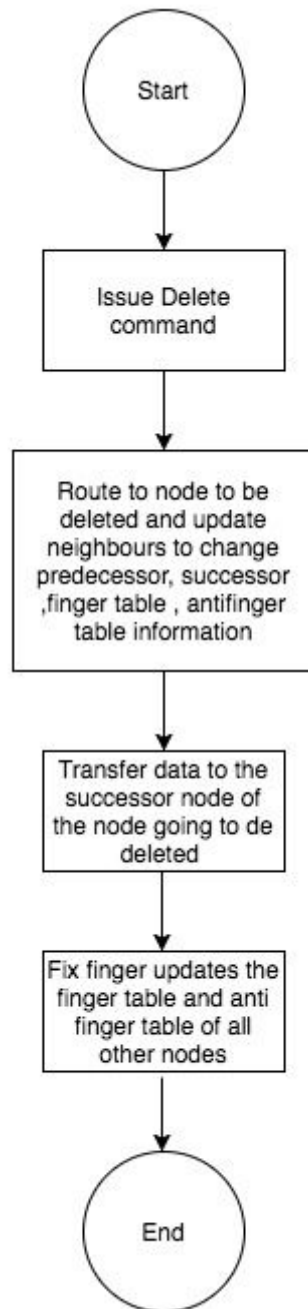
We have tried to Debug our code and test against the hypothesis. We are only printing those functionalities which are required for the results. Test results are provided in the later sections of this proposal.

## 5.2. Flowchart

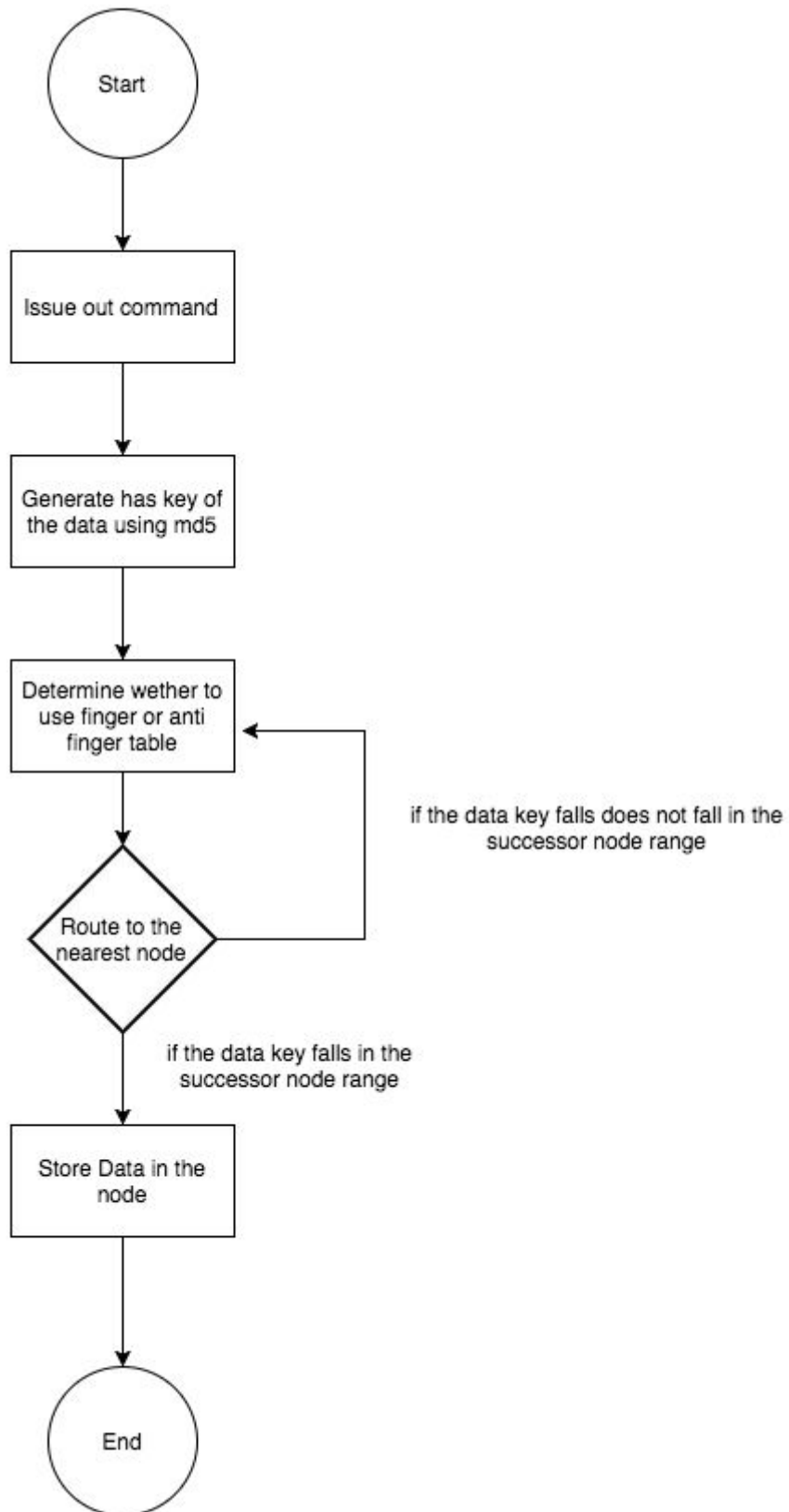
### 1. ADD



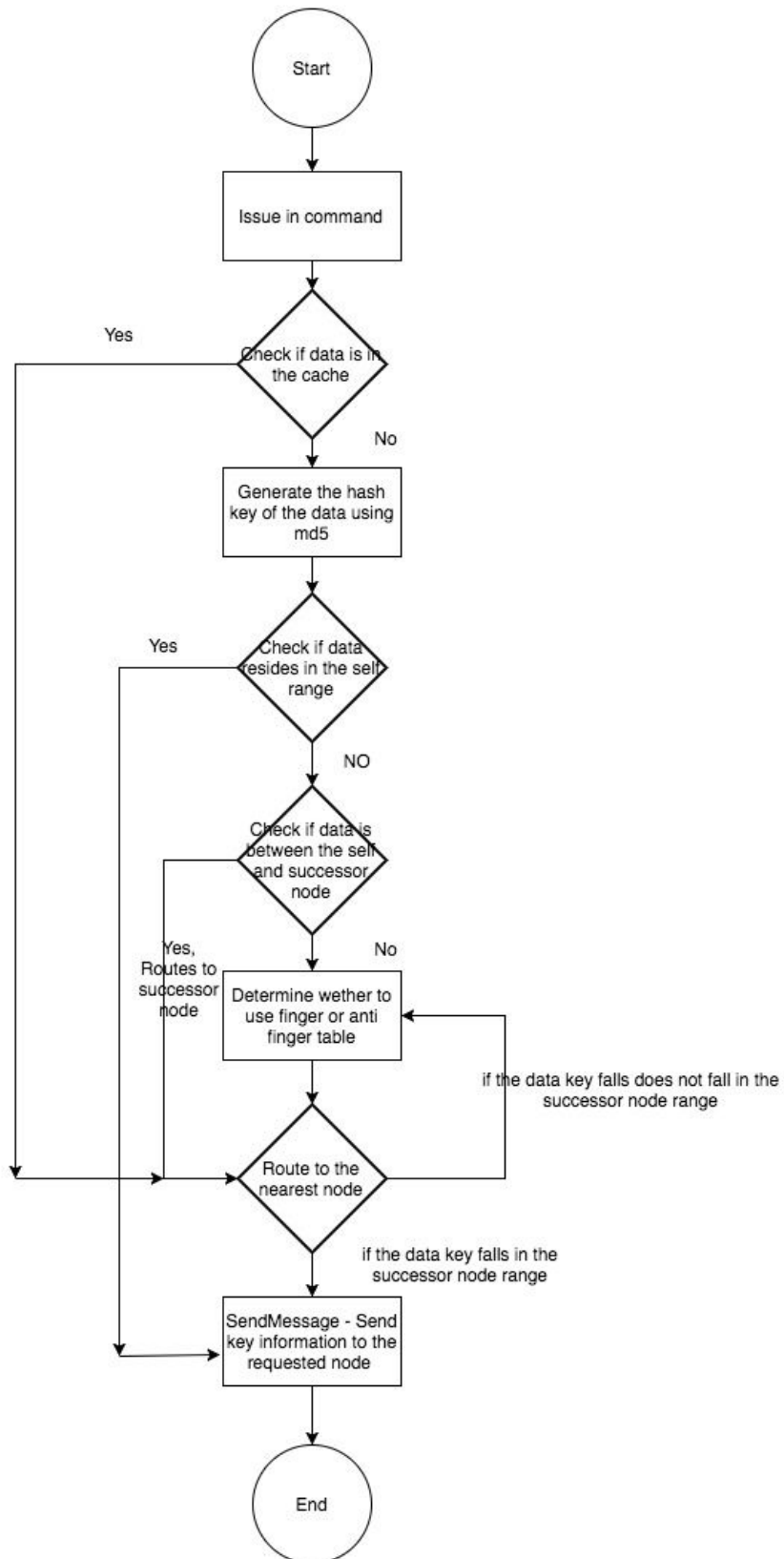
## 2. Delete



### 3. Out command



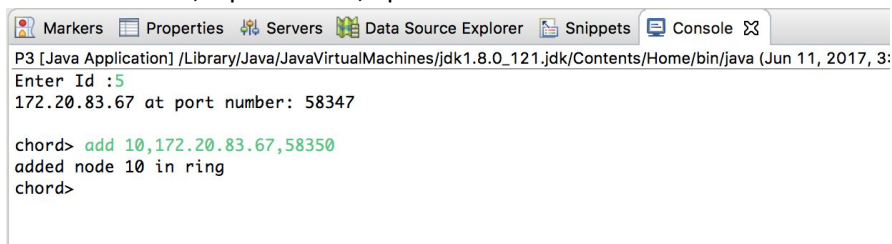
#### 4. In command



## 6. data analysis and discussion

### 6.1. output generation

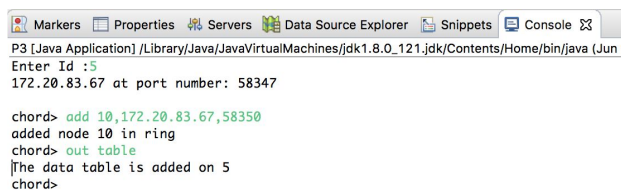
1. Add: This command is used to add a node in the chord ring. The command is used as add <node number>, <ip address>, <port number>.



```
Markers Properties Servers Data Source Explorer Snippets Console
P3 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (Jun 11, 2017, 3:
Enter Id :5
172.20.83.67 at port number: 58347

chord> add 10,172.20.83.67,58350
added node 10 in ring
chord>
```

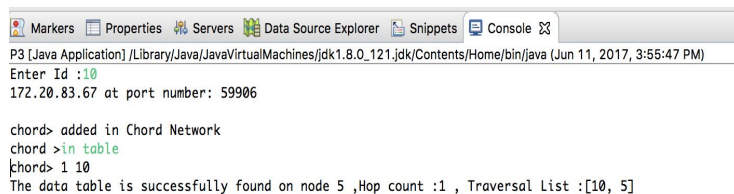
2. Delete: This command is used to delete an existing node from the chord ring. It can be used as, delete <node number>.
3. Out: This command is used to put data out to a specific node. It is used as out cloud, where cloud is the data item to be inserted.



```
Markers Properties Servers Data Source Explorer Snippets Console
P3 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (Jun
Enter Id :5
172.20.83.67 at port number: 58347

chord> add 10,172.20.83.67,58350
added node 10 in ring
chord> out table
The data table is added on 5
chord>
```

4. In: This command is used to get the node number on which the data resides. It is used as in cloud. The output will be the node number which has this data item.



```
Markers Properties Servers Data Source Explorer Snippets Console
P3 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (Jun 11, 2017, 3:55:47 PM)
Enter Id :10
172.20.83.67 at port number: 59906

chord> added in Chord Network
chord > in table
chords 1 10
The data table is successfully found on node 5 ,Hop count :1 , Traversal List :[10, 5]
chord>
```

5. printFinger: this command is used to print the finger table of a node which resides in the chord ring. It will show the successor and predecessor node values for that node.

```

printFinger
Key =11 : range =12 : successor =5 : ip =172.20.83.67 :port =59903
Key =12 : range =14 : successor =5 : ip =172.20.83.67 :port =59903
Key =14 : range =18 : successor =5 : ip =172.20.83.67 :port =59903
Key =18 : range =26 : successor =5 : ip =172.20.83.67 :port =59903
Key =26 : range =42 : successor =5 : ip =172.20.83.67 :port =59903
Key =42 : range =10 : successor =5 : ip =172.20.83.67 :port =59903
chord>

```

6. `printAntiFinger`: This command is used to print out the anti finger table of a node, which resides in the chord ring.

```

chord> printAntiFinger
Key =4 : range =5 : successor =5 : ip =172.20.83.67 :port =59903
Key =3 : range =4 : successor =5 : ip =172.20.83.67 :port =59903
Key =1 : range =3 : successor =5 : ip =172.20.83.67 :port =59903
Key =61 : range =1 : successor =5 : ip =172.20.83.67 :port =59903
Key =53 : range =61 : successor =5 : ip =172.20.83.67 :port =59903
Key =37 : range =53 : successor =5 : ip =172.20.83.67 :port =59903
chord>

```

7. `printData`: This command is used to print out the keys which resides in that node.
8. `nodeDetail`: This command is used to print all the information about node's successor and predecessor node.

```

chord> nodeDetail
Node id: 5, ip: 172.20.83.67, port: 59903
Node Successor:10, ip: 172.20.83.67, port: 59906
Node Predecessor:10, ip: 172.20.83.67, port: 59906
chord>

```

9. `paChord` : This command is used to print out the hops taken for each key searched in the traditional chord.

```

chord> paChord
The data key 0 is successfully found on node 5 ,Hop count :0 , Traversal List :[5]
The data key 1 is successfully found on node 5 ,Hop count :0 , Traversal List :[5]
0 4
0 4
The data key 2 is successfully found on node 5 ,Hop count :0 , Traversal List :[5]
0 6
The data key 3 is successfully found on node 5 ,Hop count :0 , Traversal List :[5]

```

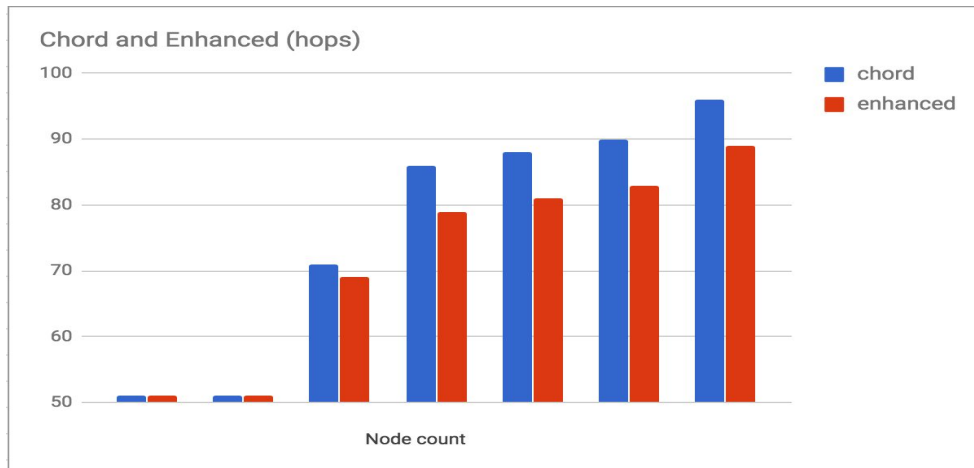
10. `paEnhanced` : This command is used to print out the hops taken for each key searched in our modified enhanced chord.

```

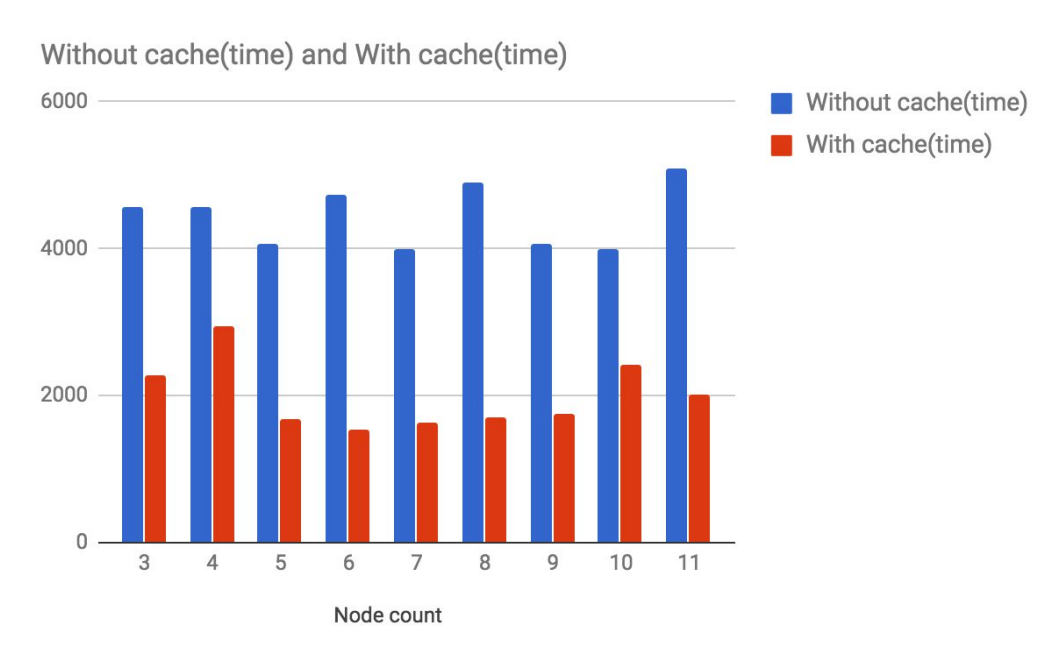
paEnhanced
The data key 0 is successfully found on node 5 ,Hop count :0 , Traversal List :[5]
0 1
The data key 1 is successfully found on node 5 ,Hop count :0 , Traversal List :[5]
0 1
The data key 2 is successfully found on node 5 ,Hop count :0 , Traversal List :[5]
0 3
The data key 3 is successfully found on node 5 ,Hop count :0 , Traversal List :[5]
0 7
The data key 4 is successfully found on node 5 ,Hop count :0 , Traversal List :[5]

```

## 6.2. output analysis

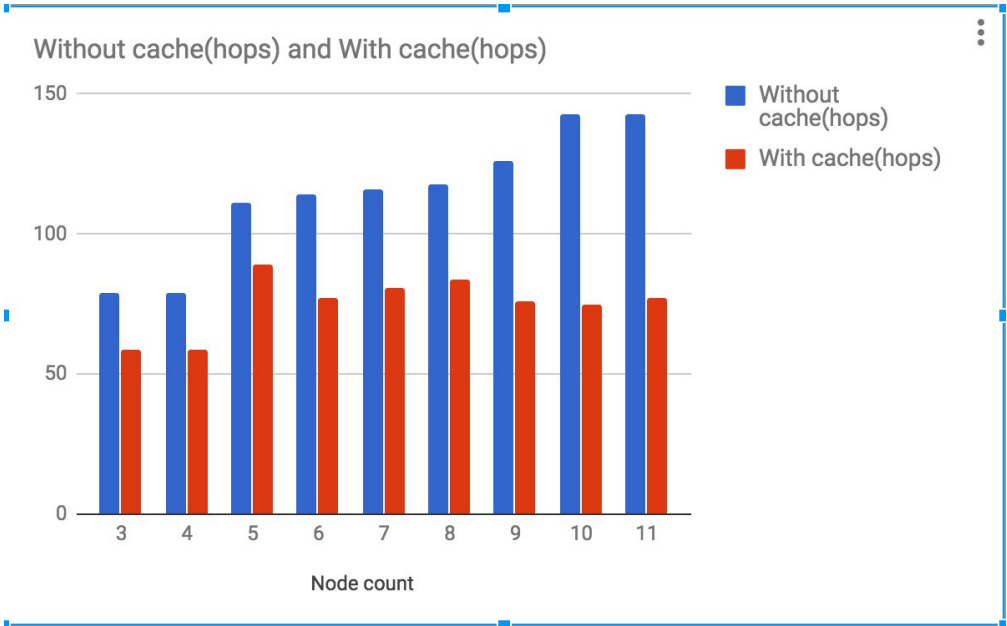


This is a graph showing the number of Hops made in Chord protocol and enhanced bidirectional chord. The x-axis represent the node count. The Y-axis represents the number of Hops made. It is clearly seen in this graph that the number of hops while using Enhanced Chord protocol reduces as opposed to Chord protocol.



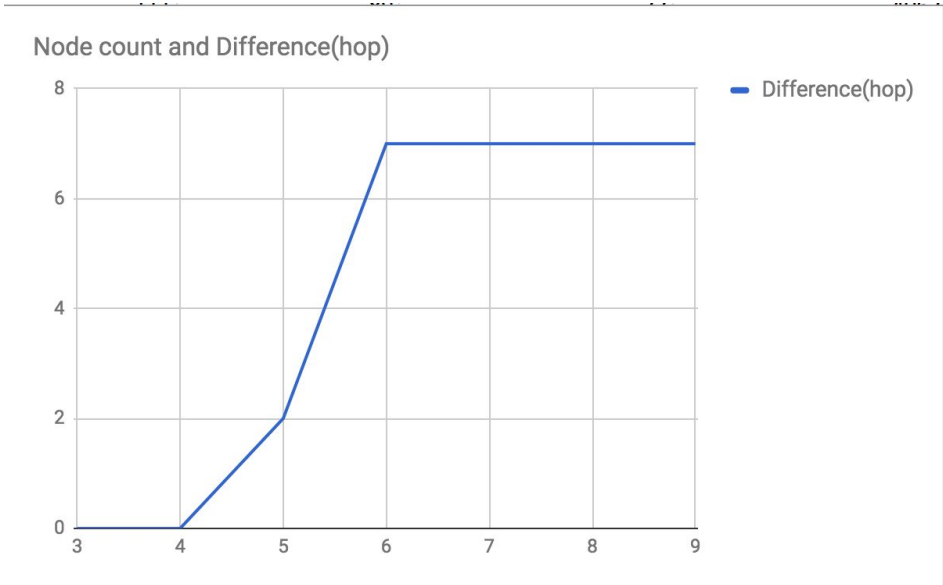
This is a graph showing the time taken to search the node where a data item is stored. It compares the cases when Cache is used and when it is not used. The x-axis represents the node number on which the query is thrown, while the y-axis show time in milliseconds.



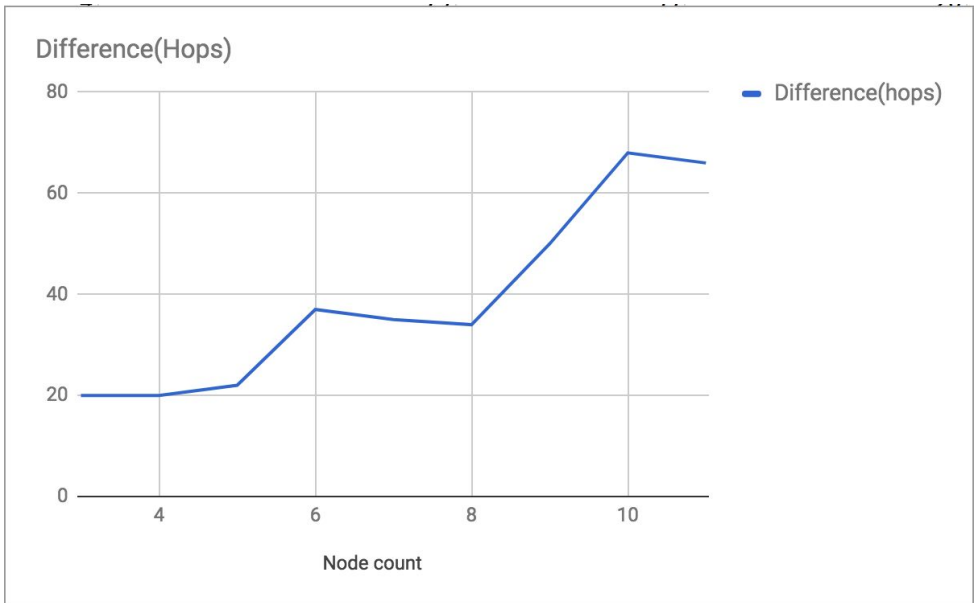


This is a graph showing the number of Hops made to locate a data item while using Cache and comparing it against Cache less Chord ring. The X-axis represent the number of nodes while the Y-axis represent the number of Hops.

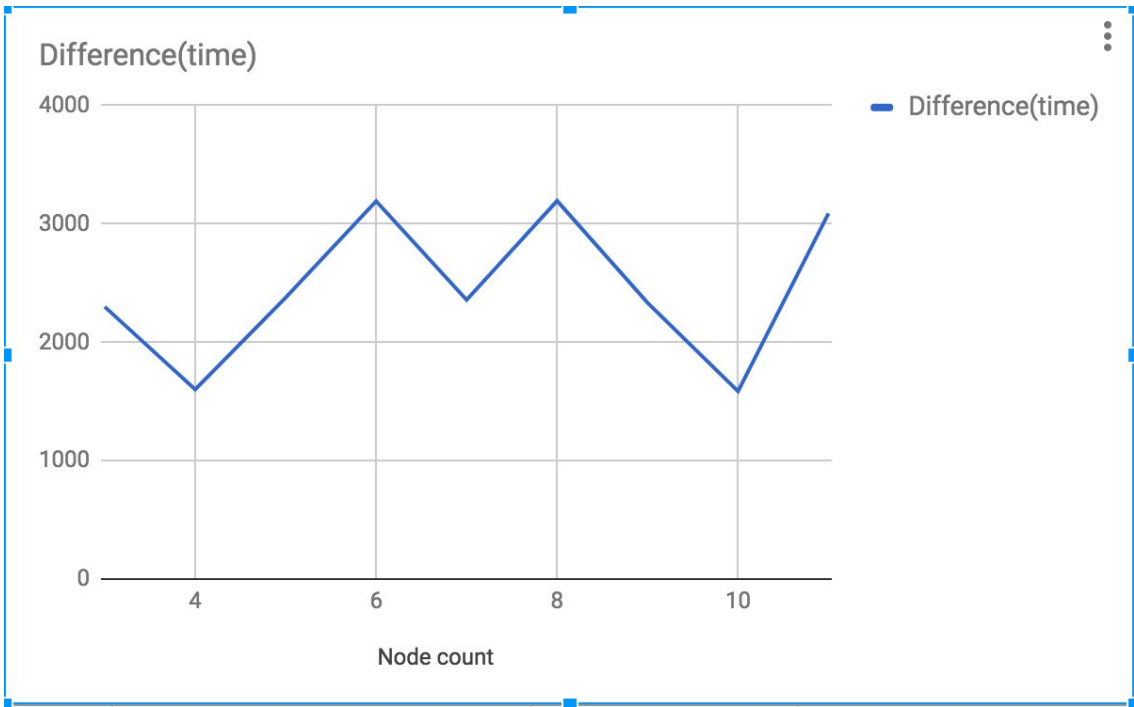
### 6.3. compare output against hypothesis



This graph shows the difference in the Hop counts(y\_axis) between the normal Chord protocol and Enhanced Chord protocol when we change number of nodes(X-axis) in chord ring. The X-axis represent the node count, while the Y-axis represent the difference.



This graph shows the difference in the number of Hops when Cache is used as opposed to when Cache is not used. The X-axis represent size of cache and y-axis represent difference between total of difference between regular chord protocol hops and enhanced chord protocol hops for 64 nodes.



This graph shows the difference in the amount of time taken to locate a node. It shows the difference in time when Cache is used as opposed to when it is not used. The X-axis represent the number of nodes when query is thrown while the Y-axis represents the difference in time measure in milliseconds.

## 6.4. abnormal case explanation (the most important task)

There are few abnormal cases which we have found in our project. Each of these is explained below with greater details :

(1) Look up data key while the finger tables are trying to get the consistency.

As mention above, chord algorithm periodically run a background thread to keep the finger table and anti finger table up-to-date with all correct information. However, there might be some time where lookup data takes more time than usual especially whenever we get a data request during the time interval when fix\_finger thread just started its periodical turn to update the information. During this case a node might disguise due to incorrect information first but eventually our data query route to other nodes and gives the correct information.

(2) Non-voluntary exit of any node (Fail-stop or byzantine failure)

Chord runs on a dynamic distributed environment where nodes join/leave the system continuously. Sometimes nodes also get failed which can not be detected by other nodes. During this time ,there might be a chances where we can get failed to serve the data request. Though we have planned to handle this situation for our future work by using heartbeat mechanism to get the the up-to-date state of any node

(3) Cache contains stale data or dirty information

As to improve the search more efficiently we have provided a cache mechanism. However, there might be a chances that our cache contains a stale/dirty information about node (Specially when a node has been added / deleted from the system and our data key has been transferred to new node but cache still points to the same old node.) We have also planned to handle this issue as our future work by using the same heartbeat mechanism only by updating the cache whenever we detect any failure.

# 7. conclusions and recommendations

## 7.1. summary and conclusions

### 7.1.1 Summary

We implemented the Chord protocol used for P2P to store and retrieve data on distributed nodes. Our project includes the use of Java programming language in Eclipse IDE. Not only did we implement the normal working of Chord protocol but also modified it by making the search bidirectional. Using this modification, we tried to reduce the number of hops during search of data. We also implemented Least Recent Used (LRU) cache for storing the nodes used repetitively to further increase the efficiency by reducing the search time. Whenever a query is passed to the CHord protocol, it contains a key pointing to a particular data item. This key is mapped into a Distributed hash Table (DHT), which contains the node IDs of all active nodes in the chord ring. Our

implementation works in the same manner, while searching the nodes in both clockwise and anti-clockwise direction.

### 7.1.2 Conclusion

We implemented the chord protocol and our proposed modified chord protocol. Our analysis suggest that the number of hops to locate a data stored on a node is reduced as compared to the normal unidirectional working of the protocol. It can also be concluded that by the use of cache to store recently used nodes, the search time for data item stored on such nodes is constant  $O(1)$ .

## 7.2. recommendations for future studies

### (1) Recovery and failure handling

As it is explained in abnormal cases , we are planning to handle the crash failure and recovery of node under the dynamic environment. To handle this case we will run one separate thread on each node in a background which will periodically ping it's successor node. Whenever it detects a failure it will simply replace its successor with the next alive node resides in finger table. This mechanism is call heartbeat mechanism to check the aliveness of all participating nodes in p2p network.

### (2) Maintain the cache consistent under a dynamic environment

As explained in abnormal cases, sometime cache contains a dirty entry (points to the incorrect/dead node) especially when a new node joins/leaves the system. This situation can also be handled by heartbeat mechanism and fix finger process. If a background thread detects the failure it will also replace/delete the data entry available with that node in cache. Hence eventually every node would get the correct information.

### (3) Test the algorithm on scalable environment.

We have tested our modified algorithm by using 6 bit identifier which can accommodate upto 64 nodes. To make our analysis more stronger, we will test the same algorithm for more scalable environment where we can analyse performance, overheads of our modified algorithm and compare the efficiency.

## 8. Bibliography

[http://pages.cs.wisc.edu/~swift/classes/cs739-sp12/blog/2012/03/chord\\_a\\_scalable\\_peertop\\_eer\\_lo.html](http://pages.cs.wisc.edu/~swift/classes/cs739-sp12/blog/2012/03/chord_a_scalable_peertop_eer_lo.html)

I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In SIGCOMM '01: Proceedings of the 2001 on Applications, technologies, architectures, and protocols for computer communications, 2001. ACM.

<http://www.ijraset.com/fileserve.php?FID=56>

INTERNATIONAL JOURNAL FOR RESEARCH IN APPLIED SCIENCE AND ENGINEERING TECHNOLOGY (IJRASET) Page 44 Advanced Chord Algorithm Vikash Jaglan Dr. Sukhvir Singh.  
Research Scholar, NCCE, ISRANA Head and Associate Professor, NCCE, ISRANA.

<https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf>

<https://msu.edu/~liyng5/docs/Chordprotocolreport.pdf>

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/>

<http://ieeexplore.ieee.org.libproxy.scu.edu/document/6835563/>

DDChord -- A Double Deck P2P System Based on Chord.

Author: Xiaoyun Zang

Published in: Computer Sciences and Applications (CSA), 2013 International Conference on

<http://ieeexplore.ieee.org.libproxy.scu.edu/document/5898953/>

**Authors:** Thanassis Bouras ; Anastasios Zafeiropoulos ; Athanassios Liakopoulos

Published in: Telecommunications (ICT), 2011 18th International Conference

<http://ieeexplore.ieee.org/document/5576476/>

[https://link.springer.com/chapter/10.1007/978-3-540-31957-3\\_88](https://link.springer.com/chapter/10.1007/978-3-540-31957-3_88)

## 9. appendices

### 9.1. program source code with documentation

Our project code is available on github. The link is provided below for reference.

[https://github.com/padariya-shraddha/P3\\_Chord](https://github.com/padariya-shraddha/P3_Chord)

### 9.2. input/output listing

Input file : dataFile.txt,data\_1.txt,data\_2.txt,data\_3.txt,data\_4.txt,data\_5.txt(included in source code)

Output file : we are printing output on console and also printing the log of finger table and anti-finger table on a text file.