The Evolution of Torrents:

**Peer-to-Peer Node Lookup Using Chord Distributed Hash Tables**

Corinne Gaines

Computer Science Department

Bowie State University

11/30/20

COSC 495: Senior Seminar – Fall 2020

Kenneth Mbale

# Abstract

Node lookup and efficiency is an important aspect in peer-to-peer (P2P) applications. The following document is a study of the use of Chord distributed hash tables (DHT) in peer-to-peer (P2P) systems. Chord provides support for one specific operation within P2P systems, given a key, it maps that specific key onto a node. In Chord, data location is easily implemented due to the singular task that Chord performs, because of this associating a key with each data item and storing the key at a node which the key will map to allows Chord to adapt efficiently as nodes join and leave the Chord Ring, which is the representation of Chord’s geometry. Chord is adaptable when nodes join and leave the Chord Ring and can answer queries as the system is continuously changing. The Chord algorithm and its geometry are demonstrated through a series of experiments and simulations to demonstrate its efficiency and usefulness as a node lookup service.

# Table of Contents

[Abstract 2](#_Toc57651177)

[Table of Contents 3](#_Toc57651178)

[List of Figures 5](#_Toc57651179)

[List of Tables 6](#_Toc57651180)

[1 Introduction 7](#_Toc57651181)

[2 Background 8](#_Toc57651182)

[2.1 Literature Review 8](#_Toc57651183)

[2.2 Key Debates and Controversies 13](#_Toc57651184)

[2.3 Research Design 14](#_Toc57651185)

[2.4 Practical Considerations 14](#_Toc57651186)

[2.5 Relevance and Importance of Research 14](#_Toc57651187)

[2.6 Problem Statement 15](#_Toc57651188)

[2.7 Hypothesis 16](#_Toc57651189)

[3 Methodology 16](#_Toc57651190)

[3.1 Primary and Secondary Experiment Metrics 16](#_Toc57651191)

[3.2 Traffic Allocation and Distribution 17](#_Toc57651192)

[3.3 Experiment Variation 1 – Tracking Finger Table Stabilization Time 17](#_Toc57651193)

[3.3.1 Experiment Elements 18](#_Toc57651194)

[3.4 Experiment Variation 2 – Chord Ring Simulation 1 18](#_Toc57651195)

[3.4.1 Experiment Elements 19](#_Toc57651196)

[3.5 Experiment Variation 3 – Chord Ring Simulation 2 19](#_Toc57651197)

[3.5.1 Experiment Elements 19](#_Toc57651198)

[4 Results and Discussion 20](#_Toc57651199)

[4.1 Finger Table Stabilization Time Results 20](#_Toc57651200)

[4.2 Chord Simulation 1 Result 21](#_Toc57651201)

[4.3 Chord Simulation 2 Result 22](#_Toc57651202)

[5 Conclusions and recommendations 22](#_Toc57651203)

[6 References 23](#_Toc57651204)

[Appendix A: Software 24](#_Toc57651205)

[Appendix B: Data Set 64](#_Toc57651206)

# List of Figures

[Finger Table Stabilization Time 1 18](#_Toc57572143)

[Finger Table Stabilization Time 2 20](#_Toc57572144)

[Chord Simulation 1 Result 22](#_Toc57572145)

[Chord Simulation 2 Result 23](#_Toc57572146)

# List of Tables

[Data Set 26](#_Toc57573200)

# 1 Introduction

Torrenting and software piracy has been a mainstay in internet culture for over twenty years. Users have always searched for ways to get free access to software, music, movies, and other forms of media. The act of torrenting is considered illegal and can be viewed as software piracy. Software piracy is the illegal copying, distribution, or use of software that is legally protected. Torrenting was brought into the public eye with the use of the peer-to-peer file sharing platform Napster in 1999. Napster was a music file sharing service in which mostly college students would use to share .mp3 files illegally. In December of 1999 Napster was sued by the Recording Industry Association of America (RIAA) and was subsequently shutdown; however, in 2001 Napster returned as a subscription-based music service in which users could pay a monthly fee to enjoy the music of their liking. Peer-to-Peer networks are not all illegal, as some peer-to-peer networks can be used in office in conditions for the sharing of resources for collaborative work to be completed. Although torrenting can be considered unethical, the technology that makes torrenting possible is quite fascinating.

Torrenting is made possible through the use of Peer-To-Peer (P2P) networks and Distributed Hash Tables (DHT). P2P networks are defined as a distributed architecture that partitions tasks or workloads between nodes or peers [1]. Each peer is deemed to be equally privileged when participating in the application of the P2P network. Within the network all peers are interconnected to share resources amongst each other without the use of a centralized server. Distributed hash tables work in tandem with structured P2P networks to achieve the goal of transmitting data from one peer to another. A distributed hash table is a decentralized storage system that provides lookup and storage schemes similar to that of a typical hash table. A hash table is a data structure that implements an associative array of an abstract data type that maps keys to values. Every node within a DHT is responsible for the keys and mapped values in which nodes can efficiently retrieve. As with regular hash tables, distributed hash tables use values mapped against keys and can be any abstract data type. Within a DHT nodes collectively form the system without any central authority denoting that it is decentralized and autonomous, meaning the system is reliable with many nodes joining, leaving, and failing thus denoting high fault tolerance, system functionality, and scalability.

This brings us specifically to the Chord peer-to-peer DHT protocol and algorithm. Chord was first introduced in 2001 by Ion Stoica, Robert Morris, David Krager, Frans, Kaashoek, and Hari Balakrishnan, and developed at MIT [1]. Chord supports one operation, when given a key it will determine the node responsible for storing the key’s value [1]. The Chord protocol specifies how to find the location of keys, how new nodes join the system, and how to recover from the failure or planned departure of existing nodes. Chord DHT is a valuable component for peer-to-peer applications and this will be shown through research and experimentation.

# 2 Background

## 2.1 Literature Review

The following resources provide information about the algorithm that Chord is based on, its geometry when used in peer-to-peer DHT, and how it can be implemented in real-world scenarios. These resources provide essential information on how the Chord protocol works and it can be implemented in P2P networks and systems. *“Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications”* was written by the developers of the Chord Protocol and provides the basis and definition of how Chord works, what Chord is, and the problems in which Chord is meant to resolve. The problem which Chord is meant to resolve is keeping track of the efficient location of the node that stores the desired data item. Chord addresses this problem by supporting the singular operation of being provided a key and mapping that key onto a node [1]. During this singular operation Chord uses consistent hashing to balance load between nodes, as all nodes receive roughly the same amount of keys. The article discusses how Chord’s lookup protocol works in comparison to other lookup operations such as Gnutella, which uses search based on user-supplied keywords, while Chord looks up data using unique identifiers [1]. The article goes into detail of the structure of Chord and how its system model simplifies the design of P2P systems and applications by addressing several difficult problems such as load balance, decentralization, scalability, availability, flexibility, cooperative mirroring, time-shared storage, distributed indexes, and large-scale combinatorial search. The article also discusses the geometry of the Chord protocol and its Big O notation. When using Chord within an N-node network each node maintains information about only O(log N) other nodes, and a lookup requires O(log N) messages. The geometry of Chord is typically in the form of ring, known as the Chord ring, which denotes consistent hashing where identifiers are ordered on the Chord ring at modulo . Within the Chord ring 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 of node of key k, denoted by successor(k). If identifiers are represented as a circle of numbers from 0 to , then successor(k) is the first node clockwise from k [1][2]. This article provides all basis information about the Chord protocol and will be an extremely useful in my research, because this article provides the algorithm for Chord as well as explanation of how Chord can be utilized in P2P systems.

When determining what routing algorithm is best for your P2P system an important factor is flexibility and resilience. These factors are discussed in *“The Impact of DHT Routing Geometry on Resilience and Proximity”*. This journal article focuses on the different types of routing geometries each DHT routing algorithm uses and how each geometry distinguishes itself based on the flexibility it provides. The article provides in-depth information on the design of various DHTs. The design of a DHT begins with its geometry which can be divided into two categories: its routing-level and system-level. Routing-level is confined to two well-defined issues, neighbor selection and route selection [2]. The first category of routing-level determines the routing behavior of the DHT. The second category is the system-level which are higher level decisions that apply across all routing-level choices, an example would be caching and replication and whether the actual delivery of messages is done iteratively or recursively. Another distinction between DHTs is the difference between the algorithm and geometry of the DHT [2]. The algorithm refers to the precise details of how neighbors and next hops are chosen, and how geometry is not exactly a precise term. Geometry refers more-so to the way in which neighbors and routes are chosen and usually has a geometric representation such as the ring in Chord. All DHT geometries are capable of providing O(log n) path lengths with O(log n) neighbors, so the article is not directly focused on how the geometry effects space and efficiency, but how the geometries place different constraints on route and neighbor selection [2]. These constraints represent the flexibility that each geometry provide, which demonstrates the impact of resilience and proximity properties of the system. The specific DHT geometries discussed are tree, hypercube, ring, butterfly, xor, and hybrid. The article compares the properties of each geometry, such as how neighbor selection is completed, route selection for both optimal and non-optimal paths, and if natural support for sequential neighbors is supported. A study is conducted in which hop-counts are compared for different DHTs for over a 65,536-node network with no failures [2]. The results for the study show that the ring routing geometry is more superior to the other options, demonstrating that Chord will provide the most optimal results in the category of least node failures. The article also provides solutions to several more studies which favor the Chord ring DHT in comparison to the other geometries previously mentioned.

With the discussion of the geometry of the Chord DHT comes the use and discussion of Chord’s implementation of finger tables. The following article provides insight on how finger tables are used in Chord, their limitations, and how the use of bidirectional finger tables can circumvent those limitations. *“Implementation of Chord P2P Protocol Using Bidirectional Finger Tables”* takes an in depth look at how finger tables are used with the Chord DHT. When using the Chord protocol, a finger table is maintained to search for keys in a more efficient manner. A routing table is maintained by each node which contains a maximum of entries, where the th entry contains a mapping to the nodes first successor, f, which succeeds the node, n by at least within Chord ring, this is known as the finger table [3]. Finger table entries include both the Chord identifier, IP address and port number of the relevant node. This article will be relevant to my research as it provides information and test results on the implementation of Chord DHT using bidirectional finger tables based on Open Chord, which is an open source platform for a Chord algorithm simulation. Open Chord can be used to demonstrate how Chord can be applied in different situations. *“A Peer-to-Peer Dictionary Using Chord DHT”* provides a demonstration on the implementation of Open Chord in a real-world project. Within the paper the Chord DHT algorithm is used with the SHA-1 algorithm for its consistent hashing algorithms [4]. The project demonstrates how Open Chord can be used to easily implement a P2P DHT using Java in a Java Runtime Environment and how to complete a basic search within a Chord P2P network [4]. This paper will be helpful in my research as it will help provide an example in how to setup a DHT with the use of Open Chord. When used in real world scenarios Chord can also be implemented as a location manager. Location management is an important factor in internet protocol (IP) services. *“DHT Chord as Location Manager”* discusses the implementation of Chord as a location manager to maintain reachability for new connections [5]. Conventional problems of location management include unavoidable delay and central point of failure. Chord provides a decentralized approach to resolve these conventional problems. To complete this study the following network components are required: chord nodes, bootstrap nodes, node positioning, an addressing scheme, node entry, and location search [5]. The chord nodes are essentially the building blocks of the network and are connected to one another in the overlay network which is constructed by the application layer [5]. The bootstrap nodes are the same as the chord nodes but are less mobile and resided in the network for longer periods. The node position is the node on the identifier circle (chord ring) which primarily depends on its Node ID, obtained by hashing its IP address. The addressing scheme is every nodes unique identifier. Node entry refers to a nodes temporary location value when it initially enter the network. Upon entry to the network, the node needs to locate a bootstrap node whose information can be obtained by querying the DHCP server. Location search refers to when a corresponding node wants to begin a session it requires a nodes IP address. To obtain this IP address the corresponding node hashes the nodes unique identifier and submits a query to its successor node for its temporary locator value. The success of Chord as a location manager is dependent on how efficiently it can retrieve the required key-value pair, and how often it replies to queries requested by the corresponding node [5]. With the implementation of Chord in real world scenarios comes the discussion of the durability of Chord and how secure it is when faced with possible Denial of Service attacks. *“Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service* “provides insight on how Chord’s highly distributed nature helps in the resistance in many different types of denial of service of attacks. Chord is resistant to attacks that take out some network links since nodes nearby in identifier space are unlikely to have any network locality, however, additional steps must be taken to preclude other attacks [6]. Within a Chord system, nodes that could pick their own identifiers could effectively delete a piece a data from the system by positioning themselves as the data’s successor and then failing to store it when asked to. This type of attack can be prevented by requiring that node identifiers correspond to a hash of a node’s IP address, which can be verified by other nodes in the system [6]. Malicious nodes could fail to execute the Chord protocol properly by resulting in arbitrarily incorrect behavior. A single misbehaving node can be detected by verifying its responses with those of other, presumably cooperative, nodes [6]. A group of misbehaving nodes could cooperate and appear to be a self-consistent Chord network while excluding legitimate nodes. There currently is no decentralized solution to this problem and relies instead on the legitimacy of the initial bootstrap node to avoid this attack [6]. This shows that Chord also has some forms of built in security simply based on its highly distributed nature, albeit it does not have complete defense against the previously mentioned attack type. This demonstrates that Chord has real-world scalability and use.

## 2.2 Key Debates and Controversies

There are several different types of DHT geometries that can be implemented such as tree, hypercube, ring, butterfly, xor, and hybrid [2]. The information provided from the resources show that Chord is best suited for implementation in P2P networks. The ring geometry, which is representative of Chord appears to be the best choice for DHT because of its flexibility. Chord’s flexibility will be demonstrated through the use of multi-threaded programming and stabilization. Multi-threading and stabilization are major components of the Chord simulation and experimentation.

## 2.3 Research Design

The design of the research will be of a correlational nature as the program developed to implement the Chord protocol will assess the relationship between predecessor nodes and successor nodes within a Chord ring with consistent SHA-1 hashing to provide an identifier to a string and socket address. The research will provide qualitative results, as any data will be observed through visual results of programming code (program output). The data collected will be done with primary resources provided through a program developed by me which will demonstrate an implementation of the Chord protocol using the Java programming language.

## 2.4 Practical Considerations

Potential obstacles include being able to successfully assess the relationships between the predecessor nodes and successor nodes within the Chord DHT. To circumvent this issue a valid IP address will be required as well as an open port to be queried. The query should be able to check and see if your request can be completed, if not the program should exit.

## 2.5 Relevance and Importance of Research

Using Chord DHT will benefit P2P networks as Chord has the following properties associated with it; the even spreading of keys over nodes which demonstrates load balance, nodes are equally important in Chord which model decentralization, the cost of node lookup will grow with the log of the number of nodes which models scalability, and there are no constraints on the structure of keys in Chord thus demonstrating flexibility. All properties of Chord DHT can be demonstrated when it is correctly implemented into a P2P network.

## 2.6 Problem Statement

The fundamental problem is how to locate the node that stores a particular data item within the P2P DHT. This is accomplished by finding the correct algorithm or protocol for your DHT in your P2P network. Several types of DHT’s already exist but there are several characteristics which would make a specific DHT be considered “good”. These characteristics include but are not limited to flexibility, neighbor selection, route selection, and static resilience. Flexibility describes the algorithmic freedom that is left after the routing of the DHT has been chosen. Neighbor selection is the freedom to choose node neighbors after setting initial node identifiers. Route selection is when the route has been laid out but may not be working there are other options for data to make its next hop to a node. Static resilience refers to how the DHT can route when a routing state is downed due to a node failure. This leads us to how others have previously tried to solve this problem in the past. Others have tried to solve the problem of fetching data by using Domain Name Systems (DNS) protocol and direct mapping, however DNS relies on special servers which would not be available when dealing with a P2P network. Freenet has also been another option that has been used when trying to fetch data as it serves to search for cached copies of files but does not guarantee the retrieval of the searched files that already exist.

## 2.7 Hypothesis

*How can implementing Chord DHT locate the required data that is requested by each node within a P2P network?*

Implementing Chord DHT can locate the required data that is requested by each node within a P2P network by using O(log(N)) and maintaining a 32-entry finger table per node. Each node supports concurrent joining and leaving on the Chord ring, which is achieved by implementing stabilization [1][2]. Every node within the Chord ring will consistently communicate with its successor node and constantly correct its finger table with multi-threaded programming.

# 3 Methodology

To successfully simulate the Chord Ring an implementation of the Chord algorithm was created using the Java programming language, Java Runtime Edition (JRE) 11, Java Development Kit (JDK) 11 and Ubuntu 20.04 LTS distribution. The program uses several class files with multi-threading and stabilization to achieve a 32-entry finger table per available node on the Chord Ring.

### 3.1 Primary and Secondary Experiment Metrics

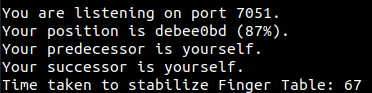
The simulation will measure the accuracy of the Chord algorithm by displaying each node’s finger table per user request. These actions will be completed with the use of multi-threaded programming and stabilization. A sample of 100 of finger tables will be taken to determine how long it takes for each finger table to stabilize. The Chord algorithm will be further tested via simulation by allowing the user to interact with a Chord Ring by joining and leaving with a selected node and querying search keys.

### 3.2 Traffic Allocation and Distribution

A selected node will be moved onto the Chord Ring locally after being chosen by the user. Only open ports will be available for use, occupied ports will not be available for this experiment. During all experiment variations traffic will be local to the user’s computer.

### 3.3 Experiment Variation 1 – Tracking Finger Table Stabilization Time

During this experiment, a sample of 100 randomly generated port numbers between 7000-8999 will be used to determine how long it takes for each individual 32-entry table finger table to stabilize in milliseconds based on Chord Ring position. This tests if the nodes location on the Chord Ring has any impact on how quickly a nodes finger table is stabilized. The ports 7000-8999 were chosen because they have a higher probability of being open and available, rather than choosing ports between 4000-6000. To accomplish tracking the time it takes for each node’s finger table to stabilize an experimental version of the Stabilizer thread class was created named StabilizerE.java which implemented System.currentTimeMillis(). System.currentTimeMillis() uses the Coordinated Universal Time (UTC) to do benchmarking and performance monitoring by calculating time differences between starting points and end points [7]. System.currentTimeMillis() provides precision over accuracy to the milliseconds based on the underlying machine [7]. To accurately track the time of the stabilization time of each finger table System.currentTimeMillis() was ran at the start of the action of finger table stabilization and ended after the action was complete. Upon completion of finger table stabilization, the length of time it took for the finger table to stabilize will print out in milliseconds to the console. The following figure demonstrates the output of the code



Finger Table Stabilization Time 1

### 3.3.1 Experiment Elements

The following elements are present in this variation of the experiment:

Independent Variable: Port Number

Dependent Variables: Chord Ring, Chord Ring position, System IP Address

### 3.4 Experiment Variation 2 – Chord Ring Simulation 1

This simulation demonstrates an implementation of Chord by using multi-threaded programming and stabilization. This variation of my experiment is a simulation which demonstrates how the Chord algorithm works by allowing nodes to concurrently join and leave the Chord Ring based on user input. Each finger table will be consistently corrected and checked for errors. Upon successfully joining the Chord Ring the user will be able to view the nodes position on the Chord Ring in percentage form, the nodes predecessor and successor node, and its 32-entry finger table. The user should input “java Chord xxxx” as a chosen port number. Upon successful join of the Chord Ring the user will receive information about the port they are currently listening on, the host name, the system IP address, Chord Ring position, successor node, and predecessor node. The user will then be prompted to interact with the menu to view the nodes 32-entry finger table or to quit the program and leave the Chord Ring.

### 3.4.1 Experiment Elements

The following elements are present in this variation of the experiment:

Independent Variable: Port Number

Dependent Variables: Chord Ring, System IP Address

### 3.5 Experiment Variation 3 – Chord Ring Simulation 2

This simulation demonstrates how keys can be queried after nodes have been joined to the Chord Ring. The user joins the Chord Ring as demonstrated in experiment variation 2, upon successful join the user must run Query.java in an additional console window. The user should enter “java Query xxx.x.x.x. xxxx” with an IP address and port number to begin the query. Upon successful connection to the Chord Ring the user will be provided with the option to enter a search key or to exit the program. If the user chooses to search a key, they will be provided with the keys hash value in 8-digit hexadecimal code along with the original nodes Chord Ring position.

### 3.5.1 Experiment Elements

The following elements are present in this variation of the experiment:

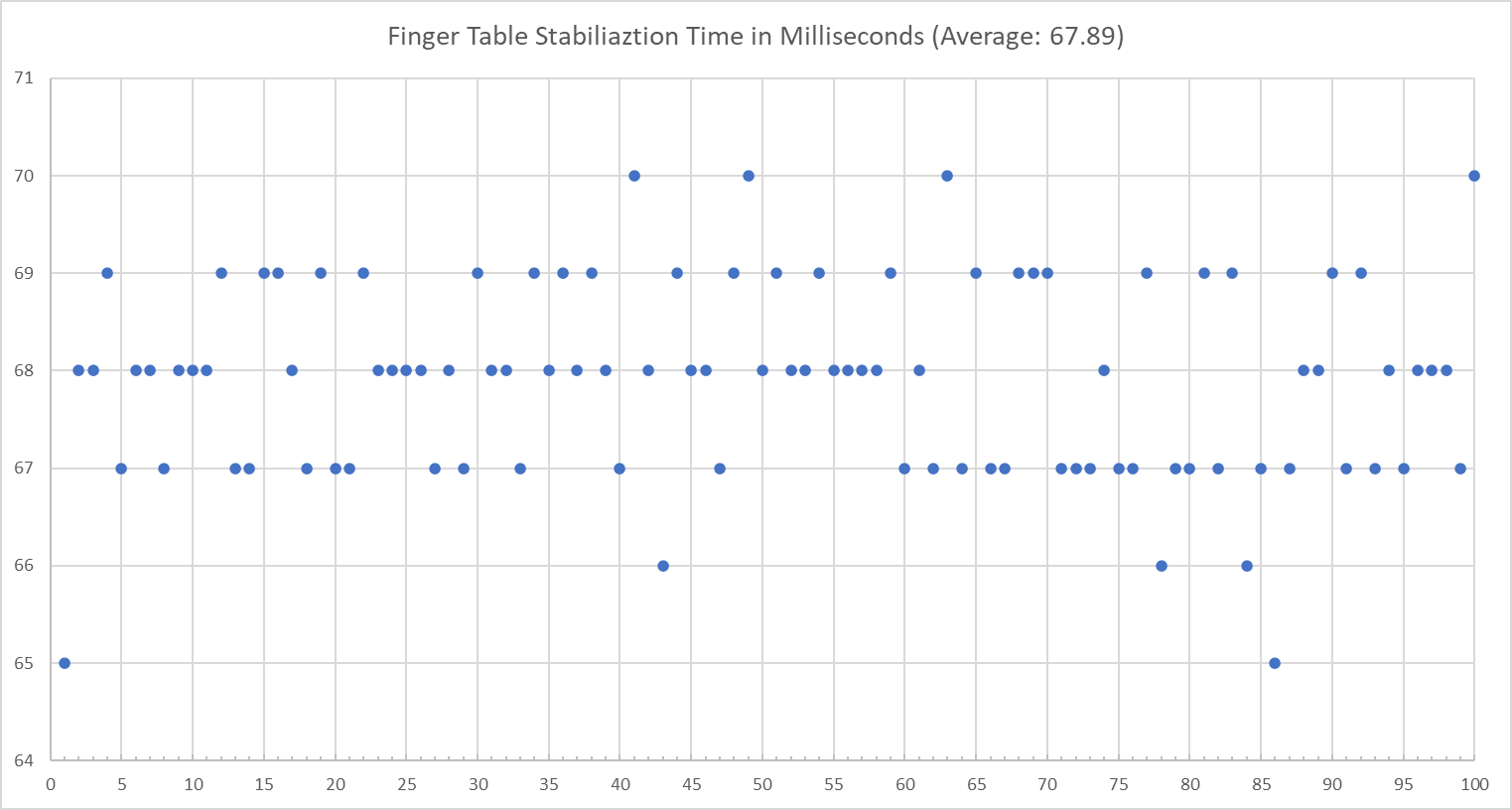
Independent Variables: System IP Address, Port Number

Dependent Variables: Experiment Variation 2, Chord Ring, Chord Ring Position

# 4 Results and Discussion

### 4.1 Finger Table Stabilization Time Results

After selecting 100 ports with varying numbers between 7000-8999 enough data was collected to determine the average finger stabilization time of all samples was 67.89 milliseconds. The lowest finger table stabilization times were 65 milliseconds for two nodes, one being at location two on the Chord Ring and the other being at location 87 on the Chord Ring. The highest stabilization time was 70 milliseconds for, four nodes, one being at location 40, two being at location 49, three being at location 62, and four being at location 99 on the Chord Ring.

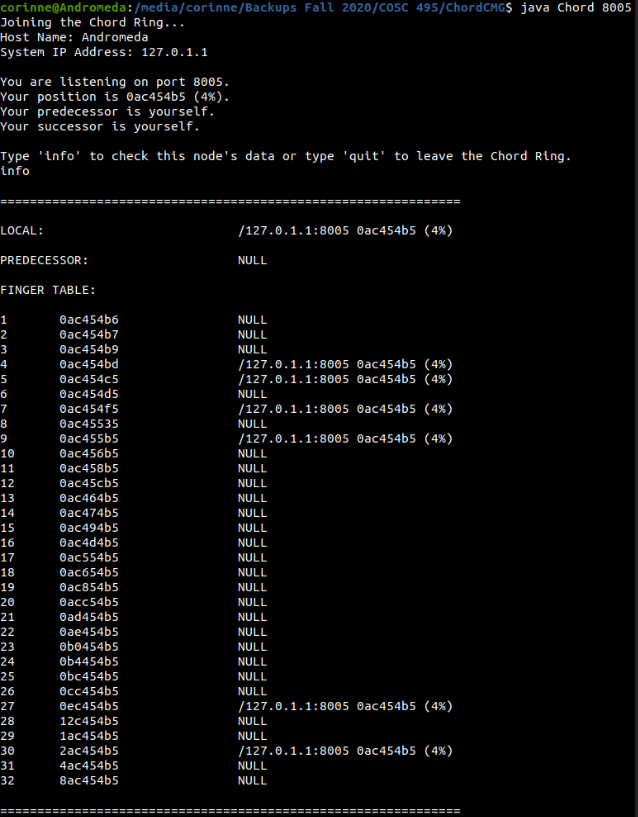


Finger Table Stabilization Time 2

Based on these results the nodes location on the Chord Ring is not correlational to how long it will take for the nodes finger table to stabilize. However, the amount of time it takes for the 32-entry finger tables to stabilize is extremely reasonable and fast and demonstrates Chord’s usability as a DHT in peer-to-peer systems.

### 4.2 Chord Simulation 1 Result

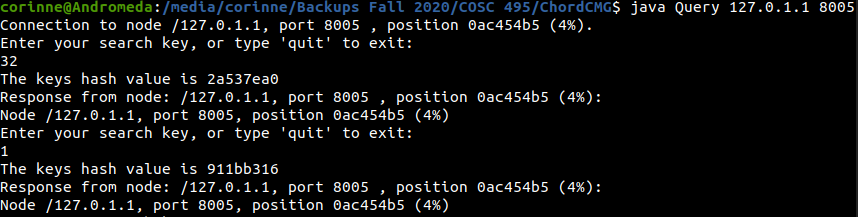
The Chord Ring simulation implements the Chord algorithm via multi-threaded programming and user interaction. The following figure shows the Chord Ring simulation running to completion with full view of a selected nodes finger table.



Chord Simulation 1 Result

### 4.3 Chord Simulation 2 Result

The second Chord Ring simulation is a continuation of the first Chord Ring simulation. After successfully running the first simulation the second simulation can be run to query hash key values in 8-digit hex code along with the original location of the node on the Chord Ring.



Chord Simulation 2 Result

# 5 Conclusions and recommendations

The Chord protocol supports one powerful primitive; given a key it determines the node responsible for storing the key’s value, and performs this primitive efficiently based on how each node maintains routing information in an N-node network for O(log N) other nodes and resolves node lookups via O(log N) messages sent to other nodes. Chord is simplistic by nature and demonstrates provable performance when faced with con-current node departures and node arrivals. In the face of departures and arrivals Chord continues to work correctly through degrading performance, even when a nodes information is partially correct. My analysis of the simulation results confirm that Chord can recover correctly from node departures and joins, can answer most node lookup events correctly during recoveries, and that Chord scales well based on the number of nodes within its Chord Ring. Chord will be a valuable tool in real-world peer-to-peer applications such as cloud file storage systems and cooperative file sharing.

# 6 References

[1] Morris R, Kaashoek M, Karger D, Balakrishnan H, Stoica I, Liben-Nowell D, Dabek F. Chord: A scalable peer-to-peer look-up protocol for internet applications. IEEE/ACM Transactions On Networking. 2003;11(1):17–32.

[2] Gummadi K, Gummadi R, Gribble S, Ratnasamy S, Shenker S, Stoica I. The Impact of DHT Routing Geometry on Resilience and Proximity. Computer Communication Review. 2003;33(4):381–394. doi:10.1145/863997.863998

[3] Kalia D, Ramachandran P, Li Y, Chaudhary S. Implementation of chord P2P protocol using bidirectional finger tables. <http://www.cse.msu.edu/~chaudh34/>

[4] Sarmady S. A Peer-to-Peer Dictionary Using Chord DHT. School of Computer Science, Universiti Sains Malaysia. 2007. <http://www.sarmady.com/siamak/papers/dht-soft-300807.pdf>

[5] Imtiaz WA, Ahmed H. DHT CHORD as Location Manager.

[6] Dabek F, Brunskill E, Kaashoek MF, Karger D, Morris R, Stoica I, Balakrishnan H. Building peer-to-peer systems with chord, a distributed lookup service. Proceedings of the Workshop on Hot Topics in Operating Systems - HOTOS. 2001;(Section 2):81–86. doi:10.1109/hotos.2001.990065

[7] Current Millis. <https://currentmillis.com/tutorials/system-currentTimeMillis.html>

# Appendix A: Software

The code was built using the Java programming language, using Java Runtime Edition (JRE) 11, Java Development Kit (JDK) 11 and Ubuntu 20.04 LTS distribution. The program uses several class files with multi-threading to demonstrate how the Chord DHT algorithm is implemented.

* The Chord class is the main class of the program, which offers UI to create a Chord node and join an existing Chord ring.
* Query is a secondary main class which must be run concurrently with Chord.java upon joining the Chord ring. This class provides an interface for users to search for keys by querying a valid chord node via IP address and port number.
* The Node class implements the Node data structure and the functionalities of the Chord node.
* The Hasher class provides a myriad of services including hashing, computation, and network & address services
* The Stabilizer class is a thread class which periodically asks the successor node for its predecessor and determines if the current node should be updated or deleted.
* The StabilizerE thread class is the same as the Stabilizer class, however this version allows for the time it takes for finger table stabilization to be tracked in milliseconds.
* The CheckPredecessor class is a thread class which continuously checks the predecessor thread and periodically asks for the predecessor nodes “keep-alive” signal and will delete the predecessor node if it is dead.
* The FixFT class is a thread class which accesses entries in the finger table and corrects them.
* The PortListener class is a thread class that continuously listens to a port and asks the “talking” thread to process when a request us accepted.
* The TalkToSocket class is a runnable thread class that processes requests accepted by listener nodes and writes their responses to the appropriate socket.

import java.util.\*;

import java.io.\*;

import java.net.\*;

import static java.lang.System.out;

/\*

Main Class - Offers UI to create a Chord Node and join an existing ring

\*/

public class Chord

{

private static Node node;

private static InetSocketAddress contact;

private static Hasher hasher;

public static void main(String[] args) throws SocketException

{

hasher = new Hasher();

String host\_name = null;

String sys\_ip = null;

try

{

InetAddress ilocalhost = InetAddress.getLocalHost();

host\_name = ilocalhost.getHostName();

sys\_ip = ilocalhost.getHostAddress();

}

catch(Exception e)

{

host\_name = "Cannot get Host Name";

sys\_ip = "Cannot get System IP Address";

}

node = new Node(Hasher.createSocketAddress(sys\_ip+":"+args[0])); //create new node

if(args.length == 1)

{

contact = node.getAddress();

}

else if(args.length == 3)

{

contact = Hasher.createSocketAddress(args[1]+":"+args[2]);

if(contact == null)

{

System.out.println("Cannot find socket address. Exiting Program...");

return;

}

}

else

{

System.out.println("Incorrect Input. Exiting program...");

System.exit(0);

}

boolean successful\_join = node.join(contact);

if(!successful\_join)

{

System.out.println("Cannot connect to the node you are trying to contact. Exiting Program...");

System.exit(0);

}

System.out.println("Joining the Chord Ring...");

System.out.println("Host Name: "+host\_name);

System.out.println("System IP Address: "+sys\_ip);

node.printNeighbors();

Scanner input = new Scanner(System.in);

while (true)

{

System.out.println("\nType 'info' to check this node's data or type 'quit' to leave the Chord Ring.");

String command = null;

command = input.next();

if(command.startsWith("quit"))

{

node.stopAllThreads();

System.out.println("Leaving the Chord Ring...");

System.exit(0);

}

else if(command.startsWith("info"))

{

node.printDataStructure();

}

}

}

} //END CHORD

import java.util.\*;

import java.io.\*;

import java.net.\*;

/\*

Query class that provides an interface for users to do searches by querying a valid chord node

\*/

public class Query

{

private static InetSocketAddress localAddress;

private static Hasher hasher;

public static void main(String[] args)

{

hasher = new Hasher();

if(args.length == 2)

{

localAddress = Hasher.createSocketAddress(args[0]+":"+args[1]); //parses the socket address from the argument

if(localAddress == null) //on parsing fail

{

System.out.println("The address you are trying to contact cannot be found. Exiting Program...");

System.exit(0);

}

String response = Hasher.sendRequest(localAddress, "KEEP");

if(response == null || !response.equals("ALIVE"))

{

System.out.println("1The node you are trying to contact cannot be found. Exiting Program...");

System.exit(0);

}

System.out.println("Connection to node "+localAddress.getAddress().toString()+", port "+localAddress.getPort()+" , position "+Hasher.hexIdAndPosition(localAddress)+".");

//System stability check

boolean pred = false;

boolean succ = false;

InetSocketAddress pred\_addr = Hasher.requestAddress(localAddress, "YOUREPRE");

InetSocketAddress succ\_addr = Hasher.requestAddress(localAddress, "YOURSUCC");

/\*if(pred\_addr == null || succ\_addr == null)

{

System.out.println("2The node you are trying to contact has been disconnected. Exiting program...");

System.exit(0);

}

if(pred\_addr.equals(localAddress))

pred = true;

if(succ\_addr.equals(localAddress))

succ = true;

\*/

//System is stable if the node has a valid predecessor and successor or neither

while(pred^succ)

{

System.out.println("System Stabilizaing...");

pred\_addr = Hasher.requestAddress(localAddress, "YOURPRE");

succ\_addr = Hasher.requestAddress(localAddress, "YOURSUCC");

if(pred\_addr == null || succ\_addr == null)

{

System.out.println("3The node you are trying to contact has been disconnected. Exiting program...");

System.exit(0);

}

if(pred\_addr.equals(localAddress))

pred = true;

else

pred = false;

if(succ\_addr.equals(localAddress))

succ = true;

else

succ = false;

try

{

Thread.sleep(500);

}

catch(InterruptedException e)

{}

}

//Get user input

Scanner userinput = new Scanner(System.in);

while (true)

{

System.out.println("Enter your search key, or type 'quit' to exit: ");

String command = null;

command = userinput.nextLine();

if(command.startsWith("quit"))

{

System.exit(0);

}

else if(command.length() > 0)

{ //search

long hash = Hasher.hashString(command);

System.out.println("The keys hash value is "+Long.toHexString(hash));

InetSocketAddress result = Hasher.requestAddress(localAddress, "FINDSUCC\_"+hash);

if(result == null)

{ //if send request fails the local node is disconnected

System.out.println("4The node you are trying to contact has been disconnected. Exiting program...");

System.exit(0);

}

System.out.println("Response from node: "+localAddress.getAddress().toString()+", port "+localAddress.getPort()+" , position "+Hasher.hexIdAndPosition(localAddress)+":");

System.out.println("Node "+result.getAddress().toString()+", port "+result.getPort()+", position "+Hasher.hexIdAndPosition(result));

}

}

}

else

{

System.out.println("Invalid input. Exiting Program...");

}

}

} //END QUERY

import java.util.\*;

import java.io.\*;

import java.net.\*;

/\*

Implements the Node data structure and functionalities of the Chord Node

\*/

public class Node

{

private long localId;

private InetSocketAddress localAddress;

private InetSocketAddress predecessor;

private HashMap<Integer, InetSocketAddress>finger;

private FixFT fix\_ft;

private PortListener listener;

private CheckPredecessor check\_predecessor;

private Stabilizer stabilizer;

public Node (InetSocketAddress inet)

{ //Node Constructor

localAddress = inet;

localId = Hasher.hashSocketAddress(localAddress);

finger = new HashMap<Integer, InetSocketAddress>(); //initializes empty FT

for(int i = 1; i <= 32; i++)

{

updateIthFinger(i, null);

}

predecessor = null; //initializes predecessor node

listener = new PortListener(this); //initialize threads

stabilizer = new Stabilizer(this);

fix\_ft = new FixFT(this);

check\_predecessor = new CheckPredecessor(this);

}

/\*GETTERS\*/

public long getId()

{

return localId;

}

public InetSocketAddress getAddress()

{

return localAddress;

}

public InetSocketAddress getPredecessor()

{

return predecessor;

}

public InetSocketAddress getSuccessor()

{

if (finger != null && finger.size() > 0)

{

return finger.get(1);

}

return null;

}

/\*NODE/CHORD RING EVENT FUNCTIONS\*/

public boolean join(InetSocketAddress contact)

{ //Creates a chord ring or joins node to already created chord ring

if(contact != null && !contact.equals(localAddress)) //if the contact is a different node, contact that node

{

InetSocketAddress successor = Hasher.requestAddress(contact, "FINDSUCC\_" +localId);

if(successor == null)

{

System.out.println("Cannot find the node you are trying to contact. Exiting Program...");

return false;

}

updateIthFinger(1, successor);

}

listener.start(); //kickoff all threads after node joins the chord ring

stabilizer.start();

fix\_ft.start();

check\_predecessor.start();

return true;

}

/\*SUCCESSOR/PREDECESSOR NODE FUNCTIONS\*/

private synchronized void setPredecessor(InetSocketAddress pre)

{ //Sets predecessor node using a new value

predecessor = pre;

}

private void fillSuccessor()

{ //Fills successor node with candidiates from finger table, or uses the predecessor node

InetSocketAddress successor = this.getSuccessor();

if(successor == null || successor.equals(localAddress))

{

for(int i = 2; i <= 32; i++)

{

InetSocketAddress ithfinger = finger.get(i);

if(ithfinger!= null && !ithfinger.equals(localAddress))

{

for(int j = i-1; j >= 1; j--)

{

updateIthFinger(j, ithfinger);

}

break;

}

}

}

successor = getSuccessor();

if((successor == null || successor.equals(localAddress)) && predecessor!=null && !predecessor.equals(localAddress))

{

updateIthFinger(1, predecessor);

}

}

private void deleteSuccessor()

{ //Deletes successor node and all following nodes equal to the previous successor node

InetSocketAddress successor = getSuccessor();

if(successor == null) //if there is nothing to delete, simply return

return;

int i = 32; //finds the last existence of successor in the finger table

for(i = 32; i > 0; i--)

{

InetSocketAddress ithF = finger.get(i);

if(ithF != null && ithF.equals(successor))

break;

}

for(int j = i; j >= 1; j--) //deletes last successor

updateIthFinger(j, null);

if(predecessor != null && predecessor.equals(successor)) //if predecessor node is successor, delete it

setPredecessor(null);

fillSuccessor(); //tries fills successor node

successor = getSuccessor();

if ((successor == null || successor.equals(successor)) && predecessor != null && !predecessor.equals(localAddress))

{ //If the succesor node is still null or the local node and the predecessor node is another continue querying the

//predecessor node until it finds the local nodes new successor

InetSocketAddress p = predecessor;

InetSocketAddress p\_pre = null;

while (true)

{

p\_pre = Hasher.requestAddress(p, "YOURPRE");

if (p\_pre == null)

break;

if (p\_pre.equals(p) || p\_pre.equals(localAddress)|| p\_pre.equals(successor))

{ //If p's predecessor is the chosen node, it is deleted, or if nothing is found in p or the local address

//p is the current nodes new sucessor

break;

}

else //otherwise, keep querying

{

p = p\_pre;

}

}

// update successor

updateIthFinger(1, p);

}

}

public InetSocketAddress find\_successor(long id)

{ //Asks the current node to find the ID of the successor node

InetSocketAddress ret = this.getSuccessor(); //initialize the return value as the nodes successor

InetSocketAddress pre = find\_predecessor(id); //get predecessor node

if (!pre.equals(localAddress)) //if it is another node, get its successor

ret = Hasher.requestAddress(pre, "YOURSUCC");

if (ret == null) //if the return is null, set it as the local node

ret = localAddress;

return ret;

}

private InetSocketAddress find\_predecessor(long findid)

{

InetSocketAddress n = this.localAddress;

InetSocketAddress n\_successor = this.getSuccessor();

InetSocketAddress most\_recently\_alive = this.localAddress;

long n\_successor\_relative\_id = 0;

if(n\_successor != null)

n\_successor\_relative\_id = Hasher.computeRelativeId(Hasher.hashSocketAddress(n\_successor), Hasher.hashSocketAddress(n));

long findid\_relative\_id = Hasher.computeRelativeId(findid, Hasher.hashSocketAddress(n));

while(!(findid\_relative\_id > 0 && findid\_relative\_id <= n\_successor\_relative\_id))

{

InetSocketAddress pre\_n = n; //temp to save the current node

if(n.equals(this.localAddress)) //if the current node is the local node, find the closest

{

n = this.closest\_preceding\_finger(findid);

}

else //if the current node is remote, send a request to get the closest

{

InetSocketAddress result = Hasher.requestAddress(n, "CLOSEST\_" + findid);

if(result == null) //if no response, set n to the most recent node

{

n = most\_recently\_alive;

n\_successor = Hasher.requestAddress(n, "YOURSUCC");

if(n\_successor==null)

{

System.out.println("It's not possible.");

return localAddress;

}

continue;

}

else if (result.equals(n)) //if n is the closest to itself, return itself

return result;

// else n's closest is other node "result"

else

{ //else if n's closest return the result

most\_recently\_alive = n; //set n as most receently alive

n\_successor = Hasher.requestAddress(result, "YOURSUCC"); //get result from successor node

if (n\_successor!=null) //if there is a response, it is the next node

{

n = result;

}

else

{ //else n is successor

n\_successor = Hasher.requestAddress(n, "YOURSUCC");

}

}

//compute the relative ID's for judgement in while loop

n\_successor\_relative\_id = Hasher.computeRelativeId(Hasher.hashSocketAddress(n\_successor), Hasher.hashSocketAddress(n));

findid\_relative\_id = Hasher.computeRelativeId(findid, Hasher.hashSocketAddress(n));

}

if (pre\_n.equals(n))

break;

}

return n;

}

public void clearPredecessor()

{ //Clears the predecessor node

setPredecessor(null);

}

public String notify(InetSocketAddress successor)

{ //Notifies successor node that this node should be its predecessor

if(successor!=null && !successor.equals(localAddress))

return Hasher.sendRequest(successor, "IAMPRE\_"+localAddress.getAddress().toString()+":"+localAddress.getPort());

else

return null;

}

public void notified (InetSocketAddress newpre)

{ //Notifies another node and sets it as predecessor if it is

if(predecessor == null || predecessor.equals(localAddress))

{

this.setPredecessor(newpre);

}

else

{

long oldpre\_id = Hasher.hashSocketAddress(predecessor);

long local\_relative\_id = Hasher.computeRelativeId(localId, oldpre\_id);

long newpre\_relative\_id = Hasher.computeRelativeId(Hasher.hashSocketAddress(newpre), oldpre\_id);

if(newpre\_relative\_id > 0 && newpre\_relative\_id < local\_relative\_id)

this.setPredecessor(newpre);

}

}

/\*FINGER TABLE FUNCTIONS\*/

public synchronized void updateFingerTable(int i, InetSocketAddress inet)

{ //Synchrnize all threads trying to modify the finger table. Updates the finger table based on params

if(i > 0 && i <= 32) //index in [1-32], updates the ith Finger

updateIthFinger(i, inet);

else if(i == -1) //calling node wants to delete successor

deleteSuccessor();

else if(i == -2) //calling node wants to delete a finger in the finger table

deleteCertainFinger(inet);

else if(i == -3) //calling node wants to fill successor

fillSuccessor();

}

private void updateIthFinger(int i, InetSocketAddress inet)

{ //Updates ith finger in the finger table using a new value

finger.put(i, inet);

if(i == 1 && inet != null && !inet.equals(localAddress))

notify(inet); //if the updated node is the successor, notify the new successor

}

private void deleteCertainFinger(InetSocketAddress f)

{ //Deletes a node from the finger table (erases complete existence)

for(int i = 32; i > 0; i--)

{

InetSocketAddress ithfinger = finger.get(i);

if(ithfinger != null && ithfinger.equals(f))

finger.put(i, null);

}

}

public InetSocketAddress closest\_preceding\_finger(long findid)

{ //Returms the closest finger to the preceeding node

long findid\_relative = Hasher.computeRelativeId(findid, localId);

for(int i = 32; i > 0; i--) //checks the last item in the finger table

{

InetSocketAddress ith\_finger = finger.get(i);

if (ith\_finger == null)

{

continue;

}

long ith\_finger\_id = Hasher.hashSocketAddress(ith\_finger);

long ith\_finger\_relative\_id = Hasher.computeRelativeId(ith\_finger\_id, localId);

if(ith\_finger\_relative\_id > 0 && ith\_finger\_relative\_id < findid\_relative) //if the relative ID is the clocest check if alive

{

String response = Hasher.sendRequest(ith\_finger, "KEEP");

if(response!=null && response.equals("ALIVE")) //if alive, return it

{

return ith\_finger;

}

else

{ //remove its existence from the finger table

updateFingerTable(-2, ith\_finger);

}

}

}

return localAddress;

}

/\*PRINTABLES\*/

public void printNeighbors()

{ //Prints node neighbors

System.out.println("\nYou are listening on port "+localAddress.getPort()+"."+ "\nYour position is "+Hasher.hexIdAndPosition(localAddress)+".");

InetSocketAddress successor = finger.get(1);

//if predecessor and successor are not found

if((predecessor == null || predecessor.equals(localAddress)) && (successor == null || successor.equals(localAddress)))

{

System.out.println("Your predecessor is yourself.");

System.out.println("Your successor is yourself.");

}

else

{ //if either successor or predecessor are found

if(predecessor != null)

{

System.out.println("Your predecessor is node "+predecessor.getAddress().toString()+", "

+ "port "+predecessor.getPort()+ ", position "+Hasher.hexIdAndPosition(predecessor)+".");

}

else

{

System.out.println("Your predecessor node is updating...");

}

if(successor != null)

{

System.out.println("Your successor is node "+successor.getAddress().toString()+", "

+ "port "+successor.getPort()+ ", position "+Hasher.hexIdAndPosition(successor)+".");

}

else

{

System.out.println("Your successor node is updating...");

}

}

}

public void printDataStructure()

{ //Prints finger table

System.out.println("\n==============================================================");

System.out.println("\nLOCAL:\t\t\t\t"+localAddress.toString()+"\t"+Hasher.hexIdAndPosition(localAddress));

if(predecessor != null)

System.out.println("\nPREDECESSOR:\t\t\t"+predecessor.toString()+"\t"+Hasher.hexIdAndPosition(predecessor));

else

System.out.println("\nPREDECESSOR:\t\t\tNULL");

System.out.println("\nFINGER TABLE:\n");

for(int i = 1; i <= 32; i++)

{

long ithstart = Hasher.ithStart(Hasher.hashSocketAddress(localAddress),i);

InetSocketAddress f = finger.get(i);

StringBuilder sb = new StringBuilder();

sb.append(i+"\t"+ Hasher.longTo8DigitHex(ithstart)+"\t\t");

if(f!= null)

sb.append(f.toString()+"\t"+Hasher.hexIdAndPosition(f));

else

sb.append("NULL");

System.out.println(sb.toString());

}

System.out.println("\n==============================================================\n");

}

/\*EXECUTABLES\*/

public void stopAllThreads()

{ //Stops all threads

if (listener != null)

listener.toDie();

if (fix\_ft != null)

fix\_ft.toDie();

if (stabilizer != null)

stabilizer.toDie();

if (check\_predecessor != null)

check\_predecessor.toDie();

}

} //END NODE

import java.util.\*;

import java.io.\*;

import java.net.\*;

import java.security.\*;

/\*

Hasher Class - provides, hashing, computation, and network & address services

Hashes for string, socket address, and integers

Computation is done for the relative ID of a node (how far one node is behine another node), an address' hex string and its %

position on the Chord ring, the address' 8-digit hex string, the ith start of the node's finger table, and the power of 2 when needed

Network & Address services meaning requests are sent to a node to get the required socket address/response, services also create a socket address object using a sting, and read that string from an input stream

\*/

public class Hasher

{

private static HashMap<Integer, Long> powOfTwo = null;

public Hasher()

{ //Hasher Constructor

powOfTwo = new HashMap<Integer, Long>(); //initializes a power of two hash table

long base = 1;

for(int i = 0; i <= 32; i++)

{

powOfTwo.put(i, base);

base \*= 2;

}

}

/\*HASHING FUNCTIONS\*/

private static long hashHashCode (int i)

{ //Computes a 32-bit integers identifier

byte[] hashbytes = new byte[4]; //32-bit regular hash code to byte [4]

hashbytes[0] = (byte) (i >> 24);

hashbytes[1] = (byte) (i >> 16);

hashbytes[2] = (byte) (i >> 8);

hashbytes[3] = (byte) (i); // >> 0

MessageDigest md = null; //Tries to create SHA-1 digest

try

{

md = MessageDigest.getInstance("SHA-1");

}

catch (NoSuchAlgorithmException e)

{

e.printStackTrace();

}

if (md != null) //if SHA-1 digest is successfully created convert byte [4]

{

md.reset();

md.update(hashbytes);

byte[] result = md.digest();

byte[] compressed = new byte[4]; //compress the result for byte [4]

for (int j = 0; j < 4; j++)

{

byte temp = result[j];

for (int k = 1; k < 5; k++)

{

temp = (byte) (temp ^ result[j+k]);

}

compressed[j] = temp;

}

long ret = (compressed[0] & 0xFF) << 24 | (compressed[1] & 0xFF) << 16 | (compressed[2] & 0xFF) << 8 | (compressed[3] & 0xFF);

ret = ret&(long)0xFFFFFFFFl; //compressed result in type long

return ret; //return 32-bit identifier

}

return 0;

}

public static long hashSocketAddress(InetSocketAddress inet)

{ //Get a sockets 32-bit address identifier

int i = inet.hashCode();

return hashHashCode(i);

}

public static long hashString(String str)

{ //Get a strings 32-bit identifier

int i = str.hashCode();

return hashHashCode(i);

}

/\*COMPUTE FUNCTIONS\*/

public static long ithStart (long nodeid, int i)

{ //Universal construct to return a nodes start in the finger table - finger[i]

return (nodeid + powOfTwo.get(i-1)) % powOfTwo.get(32);

}

public static long getPowofTwo(int k)

{ //Get pow of 2

return powOfTwo.get(k);

}

public static long computeRelativeId(long universal, long local)

{ //Compute universal ID's value in realtion to its local ID [local node is regarded as 0]

long ret = universal - local;

if(ret < 0)

{

ret += powOfTwo.get(32);

}

return ret;

}

public static String longTo8DigitHex(long l)

{ //Generates a numbers 8 digit hex string in type long

String hex = Long.toHexString(l);

int lack = 8-hex.length();

StringBuilder sb = new StringBuilder();

for (int i = lack; i > 0; i--)

{

sb.append("0");

}

sb.append(hex);

return sb.toString();

}

public static String hexIdAndPosition(InetSocketAddress addr)

{ //Computes a socket address in SHA-1 hash in hex and its approximate porision in type string

long hash = hashSocketAddress(addr);

return (longTo8DigitHex(hash)+" ("+hash\*100/Hasher.getPowofTwo(32)+"%)");

}

/\*REQUST FUNCTIONS\*/

public static InetSocketAddress createSocketAddress (String str)

{ //Creates socket address using IP address and port number

if(str == null)

return null;

String[] split = str.split(":"); //splits string

if(split.length >= 2)

{

String ip = split[0]; //gets and preprocesses IP address string

if(ip.startsWith("/"))

{

ip = ip.substring(1);

}

InetAddress inet = null; //parses IP address and returns null on fail

try

{

inet = InetAddress.getByName(ip);

}

catch(UnknownHostException e)

{

System.out.println("IP Address cannot be created: "+ip);

return null;

}

String port = split[1]; //parses port number

int i\_port = Integer.parseInt(port);

return new InetSocketAddress(inet, i\_port); //combines IP address and port number into socket address

}

else //cannot split string

return null;

}

public static InetSocketAddress requestAddress (InetSocketAddress server, String req)

{ //Generates requested address by sending a request to the server

if(server == null || req == null) //invalid input, return null

{

return null;

}

String response = sendRequest(server, req); //sends the request to the server

if(response == null) //if no response, return null

{

return null;

}

else if(response.startsWith("NOTHING")) //if the server cannot find anything, the server returns itself

return server;

else //server found something, use response to create, if it fails return null

{

InetSocketAddress ret = Hasher.createSocketAddress(response.split("\_")[1]);

return ret;

}

}

public static String sendRequest(InetSocketAddress server, String req)

{ //Sends request to the server and reads its response

if(server == null || req == null) //invalid input handler

return null;

Socket talkSocket = null;

try //try to open a talkSocket and output this request to this socket

{

talkSocket = new Socket(server.getAddress(),server.getPort());

PrintStream output = new PrintStream(talkSocket.getOutputStream());

output.println(req);

}

catch(IOException e)

{

//System.out.println("\nCannot send request to "+server.toString()+"\nRequest is: "+req+"\n");

return null;

}

try //sleep, while waiting for server response

{

Thread.sleep(60);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

InputStream input = null; //get users input

try

{

input = talkSocket.getInputStream();

}

catch(IOException e)

{

System.out.println("Cannot get input stream from "+server.toString()+"\nRequest is: "+req+"\n");

}

String response = Hasher.inputStreamToString(input);

try //close socket

{

talkSocket.close();

}

catch(IOException e)

{

throw new RuntimeException("Cannot close socket", e);

}

return response;

}

/\*PRINTABLES/INPUTS/EXCEPTIONS\*/

public static String inputStreamToString (InputStream in)

{ //Reads a line from the input stream

if (in == null)

{

return null;

}

BufferedReader reader = new BufferedReader(new InputStreamReader(in));

String line = null;

try //read line from the input stream

{

line = reader.readLine();

}

catch (IOException e) //catch if nothing can be read from the input stream

{

System.out.println("Cannot read line from input stream.");

return null;

}

return line;

}

} //END HASHER

import java.util.\*;

import java.io.\*;

import java.net.\*;

/\*Stabilizer thread that periodically asks the successor node for its predecessor and determines if the current node should be updated or deleted\*/

public class Stabilizer extends Thread

{

private Node local;

private boolean alive;

public Stabilizer(Node \_local)

{

local = \_local;

alive = true;

}

@Override

public void run()

{

while (alive)

{

InetSocketAddress successor = local.getSuccessor();

if(successor == null || successor.equals(local.getAddress()))

{

local.updateFingerTable(-3, null);

}

successor = local.getSuccessor();

if(successor != null && !successor.equals(local.getAddress()))

{

InetSocketAddress x = Hasher.requestAddress(successor, "YOURPRE"); //get successor nodes predecessor

if(x == null) //if theres a bad connection with the successor delete it

{

local.updateFingerTable(-1, null);

}

else if(!x.equals(successor)) //if the successors predecessor is not itself

{

long local\_id = Hasher.hashSocketAddress(local.getAddress());

long successor\_relative\_id = Hasher.computeRelativeId(Hasher.hashSocketAddress(successor), local\_id);

long x\_relative\_id = Hasher.computeRelativeId(Hasher.hashSocketAddress(x),local\_id);

if (x\_relative\_id>0 && x\_relative\_id < successor\_relative\_id)

{

local.updateFingerTable(1,x);

}

}

else //if the successor nodes predecessor is itself then notify the successor node

{

local.notify(successor);

}

}

try

{

Thread.sleep(60);

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

public void toDie()

{

alive = false;

}

}//END STABILIZER

import java.util.\*;

import java.io.\*;

import java.net.\*;

/\*Stabilizer thread that periodically asks the successor node for its predecessor and determines if the current node should be updated or deleted\*/

//EXPERIMENT VERSION

public class StabilizerE extends Thread

{

private Node local;

private boolean alive;

public long finalTime;

public StabilizerE(Node \_local)

{

local = \_local;

alive = true;

}

@Override

public void run()

{

long startTime = System.currentTimeMillis(); //starts thread safe timer in miliseconds

while (alive)

{

InetSocketAddress successor = local.getSuccessor();

if(successor == null || successor.equals(local.getAddress()))

{

local.updateFingerTable(-3, null);

}

successor = local.getSuccessor();

if(successor != null && !successor.equals(local.getAddress()))

{

InetSocketAddress x = Hasher.requestAddress(successor, "YOURPRE"); //get successor nodes predecessor

if(x == null) //if theres a bad connection with the successor delete it

{

local.updateFingerTable(-1, null);

}

else if(!x.equals(successor)) //if the successors predecessor is not itself

{

long local\_id = Hasher.hashSocketAddress(local.getAddress());

long successor\_relative\_id = Hasher.computeRelativeId(Hasher.hashSocketAddress(successor), local\_id);

long x\_relative\_id = Hasher.computeRelativeId(Hasher.hashSocketAddress(x),local\_id);

if (x\_relative\_id>0 && x\_relative\_id < successor\_relative\_id)

{

local.updateFingerTable(1,x);

}

}

else //if the successor nodes predecessor is itself then notify the successor node

{

local.notify(successor);

}

}

long endTime = System.currentTimeMillis(); //ends thread safe timer in miliseconds

long timer = endTime - startTime; //gets final time in miliseconds

try

{

System.out.println("Time taken to stabilize Finger Table: "+finalTime);

Thread.sleep(60);

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

public void toDie()

{

alive = false;

}

}//END STABILIZERe

import java.util.\*;

import java.io.\*;

import java.net.\*;

/\*Continuoiusly checks predecessor thread and periodically asks for the predecessor nodes keep-alive signal

and will delete the predecessor if its dead\*/

public class CheckPredecessor extends Thread

{

private Node local;

private boolean alive;

public CheckPredecessor(Node \_local) {

local = \_local;

alive = true;

}

@Override

public void run() {

while (alive) {

InetSocketAddress predecessor = local.getPredecessor();

if (predecessor != null) {

String response = Hasher.sendRequest(predecessor, "KEEP");

if (response == null || !response.equals("ALIVE")) {

local.clearPredecessor();

}

}

try {

Thread.sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public void toDie() {

alive = false;

}

} //END CHECKPREDECESSOR

import java.util.\*;

import java.io.\*;

import java.net.\*;

/\*Thread that accesses random entry in the finger table and corrects it\*/

public class FixFT extends Thread

{

private Node local;

Random random;

boolean alive;

public FixFT (Node node)

{

local = node;

alive = true;

random = new Random();

}

@Override

public void run()

{

while(alive)

{

int i = random.nextInt(31) + 2;

InetSocketAddress ithFinger = local.find\_successor(Hasher.ithStart(local.getId(), i));

local.updateFingerTable(i, ithFinger);

try

{

Thread.sleep(500);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

}

}

public void toDie()

{

alive = false;

}

}//END FIXFT

import java.util.\*;

import java.io.\*;

import java.net.\*;

/\*Port listener thread that contiuously listens to a port and asks the "talking" thread to process when a request is accepted\*/

public class PortListener extends Thread

{

private Node local;

private ServerSocket serverSocket;

private boolean alive;

public PortListener(Node n)

{

local = n;

alive = true;

InetSocketAddress localAddress = local.getAddress();

int port = localAddress.getPort();

try //opens new socket

{

serverSocket = new ServerSocket(port);

}

catch(IOException e)

{

throw new RuntimeException("Error opening port: "+port+". Exiting Program...", e);

}

}

@Override

public void run()

{

while(alive)

{

Socket talkSocket = null;

try

{

talkSocket = serverSocket.accept();

}

catch(IOException e)

{

throw new RuntimeException("Cannot accept connection:", e);

}

new Thread(new TalkToSocket(talkSocket, local)).start(); //create new talker

}

}

public void toDie()

{

alive = false;

}

}//END PORTLISTENER

import java.util.\*;

import java.io.\*;

import java.net.\*;

/\*Thread that processes requests accepted by listener node and writes the responses to the socket\*/

public class TalkToSocket implements Runnable

{

Socket talkSocket;

private Node local;

public TalkToSocket(Socket \_talkSocket, Node \_local)

{

talkSocket = \_talkSocket;

local = \_local;

}

public void run()

{

InputStream input = null;

OutputStream output = null;

try

{

input = talkSocket.getInputStream();

String request = Hasher.inputStreamToString(input);

String response = processRequest(request);

if(response != null)

{

output = talkSocket.getOutputStream();

output.write(response.getBytes());

}

input.close();

}

catch (IOException e)

{

throw new RuntimeException("Cannot talk.\nServer port: "+local.getAddress().getPort()+"; Talker port: "+talkSocket.getPort(), e);

}

}

private String processRequest(String request)

{

InetSocketAddress result = null;

String ret = null;

if(request == null)

{

return null;

}

if(request.startsWith("CLOSEST"))

{

long id = Long.parseLong(request.split("\_")[1]);

result = local.closest\_preceding\_finger(id);

String ip = result.getAddress().toString();

int port = result.getPort();

ret = "MYCLOSEST\_"+ip+":"+port;

}

else if(request.startsWith("YOURSUCC"))

{

result =local.getSuccessor();

if(result != null)

{

String ip = result.getAddress().toString();

int port = result.getPort();

ret = "MYSUCC\_"+ip+":"+port;

}

else

{

ret = "NOTHING";

}

}

else if(request.startsWith("YOURPRE"))

{

result =local.getPredecessor();

if(result != null)

{

String ip = result.getAddress().toString();

int port = result.getPort();

ret = "MYPRE\_"+ip+":"+port;

}

else

{

ret = "NOTHING";

}

}

else if(request.startsWith("FINDSUCC"))

{

long id = Long.parseLong(request.split("\_")[1]);

result = local.find\_successor(id);

String ip = result.getAddress().toString();

int port = result.getPort();

ret = "FOUNDSUCC\_"+ip+":"+port;

}

else if(request.startsWith("IAMPRE"))

{

InetSocketAddress new\_pre = Hasher.createSocketAddress(request.split("\_")[1]);

local.notified(new\_pre);

ret = "NOTIFIED";

}

else if (request.startsWith("KEEP"))

{

ret = "ALIVE";

}

return ret;

}

}//END TALKTOSOCKET

# Appendix B: Data Set

Excel Spreadsheet Object can be edited or opened to view entire data set

Data Set

