DEVELOPMENT OF ROUTING MANAGER API FOR PEER TO PEER NETWORKS

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Aman Sharma

22104140

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

May 2024

CERTIFICATE

It is certified that the work contained in the thesis entitled "*Development of Routing Manager API for Peer to Peer Networks*" by *Aman Sharma* has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

May 2024

Dr.Yatindra Nath Singh
Professor,
Department of Electrical Engineering,
Indian Institute of Technology Kanpur,
Kanpur, India, 208016.

Declaration

This is to certify that the Thesis report titled "*Development of Routing Manager API for Peer to Peer Networks*" has been authored by me. It presents the research conducted by me under the supervision of Dr. Yatindra Nath Singh. To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgments, in line with established norms and practices.

Aman Sharma
M.Tech–22104140,
Signal Processing, Communications and Networks,
Department of Electrical Engineering,
Indian Institute of Technology Kanpur
Kanpur, India, 208016.

Abstract

Name of the student: **Aman Sharma** Roll No: **22104140**

Degree for which submitted: **Master of Technology** Department: **Electrical**

Thesis title: **Development of Routing Manager API for Peer to Peer Networks**

Thesis Supervisor: **Dr.Yatindra Nath Singh**

Month and year of thesis submission: **May 2024**

Peer-to-Peer(P2P) rapid and promising evolution is based on their numerous attractive features,providing the efficient means for resource exchange in the internet.Peer-to-peer networking has long been utilized by researchers, companies, and universities, employing various routing schemes like Chord, Pastry, Tapestry, and Kademlia. These schemes aim to enhance routing efficiency by reducing the network's search space at each hop or routing decision, thereby minimizing search time and required hops for resource discovery. One innovative approach is the "Chord-Tapestry Hybrid," combining the strengths of Chord and Tapestry algorithms to streamline routing table size and computational complexity. This thesis seeks to achieve two objectives: firstly, to validate the Chord-Tapestry algorithm through Dart implementation, encompassing the rules for populating and updating node routing tables, and developing pseudo-code for the hybrid algorithm. Secondly, to create the Routing Manager API for the Brihaspati-4 Project, enabling efficient data routing between nodes. This involves Dart-based development of libraries, classes, and methods to facilitate seamless data and query transmission within the Brihaspati-4 project.In addition to it for maintining even more seamless connectivity among nodes neighbour tables based on routb trip time and lat-long tables based on location are also maintained as part of Routing Manager API.

Contents

List of Figures

Acknowledgments

I would like to thanks my supervisor Dr. Yatindra Nath Singh, for providing me a wonderful opportunity to work in a novel area and motivated me to think freely with an open mind and pursue the research area of my interest, always helped me with patience whenever needed and provided me important suggestions and advice. The opportunity to work with him has been an incredible experience for me as I have learnt so much from him. This research experience has been very invaluable to me as I learned to face challenges with more patience and determination.

Chapter 1

Introduction

1.1 Peer-to-Peer Networks

In a server-based system, a central server provides resources, data, or services to other connected devices, known as clients. This setup allows for efficient centralized management and scalability. Server-based systems are most commonly employed architecture in various domains such as web hosting, database management, file storage, and application hosting. However, they suffer from several drawbacks like a single point of failure, scalability, resource bottlenecks, and costly infrastructure.

These issues can be alleviated to a significant extent through the utilization of the peer-to-peer (P2P) model. The peer-to-peer concept, a well-established and extensively researched technology, is already employed in popular applications such as Gnutella and BitTorrent. In a peer-to-peer setup, all peers are regarded as equals and can collaborate without relying on a central authority to manage various tasks. They independently choose their level of involvement and corresponding access rights to network resources. Several key features contribute to the unique capabilities and benefits such as decentralization, scalability, fault tolerance, redundancy, privacy, and security. Data replication across multiple locations ensures long-term preservation and high reliability. Scalability is attained by adding resources to boost performance or availability, while resource sharing among peers reduces ownership costs, making the network economical. In a peer-to-peer network, efficiency and stability hinge on trust among nodes, established through the identification and authentication process, which

includes verifying peer credentials such as name, email address, and department.Peer-to-peer network implementation is not without its difficulties; for example, it is more complicated than the client-server model. Peer-to-peer (P2P) networks sometimes require users to exchange their IP addresses with other peers, which can violate users' privacy. This is one of the systems' downsides. Furthermore, there's a chance that private information could be exposed to other peers because peer-to-peer networks depend on direct connections between users. Peer-to-peer networks are also perceived negatively, frequently due to worries about piracy and illegal activity.

Three main parts make up our peer-to-peer network architecture: the communication manager, routing manager, and authentication manager. As the name implies, Authentication Manager (AM) verifies the node's identity. It is in charge of creating keys, certificates, Random Node IDs, and confirming the signatures on them, among other things. The component known as the Routing Manager (RM) is where a node stores pointers to other nodes in a peer network in the form of routing tables. The routing manager keeps a variety of tables that are useful for linking nodes. Several techniques are used to maintain these tables, ensuring that peer connections and routing are effective. The Communicate Manager (CM) is in charge of creating a connection so that peers can speak with one another. Peers may be found on any private network, in the public domain, or behind NAT. CM makes sure that peers connect with each other regardless of where they are in the network. It guarantees that every node in the peer network can connect with every other node.

1.2 Comparison of Server based and P2P networks

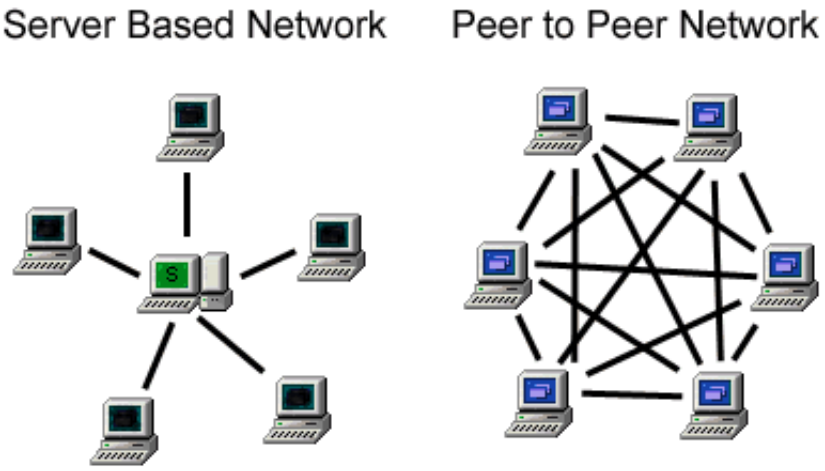

Figure 1.1: Server

Sr No.	Feature	Server-based Networks	P2P Networks
1	Server	It's architecture requires a mandatory server.	It does not require any server.
2	Robustness	It has a single point of failure. If the server fails then the entire network is rendered inoperative.	It has a highly robust architecture; all nodes are equal so even if some nodes fail, it does not affect the network.
3	Scalable	Scalability is an issue which is limited by capacity of the server.	It is highly scalable as nodes can join and leave the network without effecting performance of network.

Table 1.1: Comparison of Server-based Networks and P2P Networks

Sr No.	Feature	Server-based Networks	P2P Networks
1	Vulnerable	If servers experience degradation, the network as a whole may shut down.	It is not affected by the single point of failure problem.
2	Robustness	It has a single point of failure. If the server fails then the entire network is rendered inoperative.	It is a highly robust architecture; all nodes are equal so even if some nodes fail, it does not affect the network.
3	Cost	Server are high end machines and are very costly to procure and maintain.	P2P networks do not require such high end costly machines, hence their cost is relatively less.
4	Network Overload	Due to the server's limited capacity, a significant increase in users may result in network overload.	In P2P networks as more nodes join network becomes more stable.
5	Privacy	Server has every detail of each member of network. It is privacy concern.	P2P networks best maintain the privacy of participating nodes.
6	Storage	If server storage gets corrupted then entire data of all members are lost forever.	If one or more node fails even then the data is not lost, redundancy depends on replication factor .
7	Legal Issues	Server-based systems are easy to obtain approval due to their ease of monitoring.	P2P networks are highly decentralized and are often viewed negatively by agencies.

Table 1.2: Comparison of Server-based Networks and P2P Networks

1.3 Basics Of Routing

The process of choosing routes for data transmission from one network node to another is known as routing. It is similar to figuring out how to get to a location on a map, but for data packets traveling over networked devices. Every device, or node, in a network is given an address, and it is these addresses that decide the path a packet should travel to get to its intended destination. Several routing methods have been developed to effectively manage this task.Routing algorithms are essential for enabling resource sharing and communication between nodes in peer-to-peer (P2P) networks since they eliminate the need for centralized servers. Because these networks are decentralized, every node contributes to the operation of the network by serving as both a client and a server.

P2P network routing algorithms differ according to the size of the network, the distribution of resources, and the rate at which nodes join and exit the network. Every algorithm has trade-offs in terms of effectiveness, scalability, and resilience to network dynamics; the selection process frequently hinges on the particular needs and features of the P2P application. In P2P networks, various routing techniques are utilized.Gnutella-style flooding, Distributed Hash Tables (DHTs), Random Walks, Gradient-based Routing, Replication-based Routing, and Probabilistic Routing are a few of these methods. In our Brihaspati-4 project we have used Distributed Hash Tables(Chord-Tapestry hybrid).

1.4 Distributed Hash Tables(DHTs)

Distributed Hash Tables (DHTs) are a key component of peer-to-peer (P2P) networks, providing a scalable and efficient mechanism for distributed storage and retrieval of data. They operate based on the principles of hash functions and decentralized routing algorithms.Here's a brief overview of routing algorithms are commonly used in DHTs:

(a) **Chord**: Among the first and best-known DHT algorithms is Chord. It arranges the nodes in a ring topology, giving each one a distinct identity determined by a hash

algorithm. The routing method of Chord makes sure that queries are sent to the node whose identifier is nearest to the target key (as measured by distance on the ring). Usually, this process takes O(log N) time [3], where N is the total number of network nodes.

(b) **Kademlia**: Kademlia is another popular DHT algorithm known for its simplicity and efficiency. It organizes nodes in a binary tree-like structure called a K-bucket. Each node maintains a list of contacts for other nodes in its K-buckets. Kademlia's routing algorithm uses XOR distance metrics to determine the next hop for routing queries.[4]

(c) **Pastry**: The DHT algorithm Pastry makes use of a prefix-based routing system. It uses the node identifiers of the nodes to arrange them in a hierarchical framework. Queries are sent to nodes whose identifiers have the longest common prefix with the target key according to routing table.

(d) **Tapestry**: It employs a more adaptable routing table structure. Tapestry is comparable to Pastry as it uses a prefix-based routing method but lets nodes keep several routing entries for load balancing and fault tolerance. Tapestry uses O(log N) hops to accomplish efficient routing[2].

1.5 Hashing

Hashing is a fundamental concept in computer science that involves mapping data of arbitrary size to fixed-size values. This process is achieved using a hash function, which takes an input and produces a fixed-size string of bytes. These strings are typically of a fixed length, regardless of the size of the input.Hash functions are designed to be deterministic, meaning the same input will always produce the same output. Additionally, they should ideally be computationally efficient, ensuring that the process of hashing is quick and doesn't introduce significant overhead.The output of a hash function is commonly referred to as a hash value or hash code. These hash values are often used for various purposes in computer science.Some of these areas are as follows:

(a) **Data Integrity** - Hash functions are used to verify the integrity of data. By storing

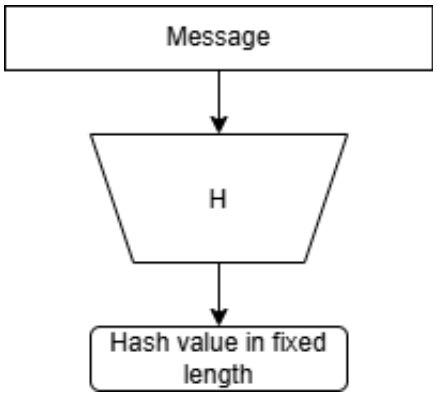

Figure 1.2: Hash Function

the hash value of a piece of data, one can later recompute the hash and compare it to the original value. If they match, it's highly likely that the data has not been altered.

(b) **Data Retrieval** - Hash functions are utilized in data structures like hash tables to quickly locate and retrieve data based on its hash value. This allows for fast access to data without needing to search through the entire dataset.

(c) **Cryptography** - Hash functions are a crucial component of cryptographic algorithms. They are used in cryptographic hash functions, which have properties like pre-image resistance, second pre-image resistance, and collision resistance. These properties ensure that it's computationally infeasible to reverse-engineer the original input from its hash value, find another input that produces the same hash, or find two different inputs that produce the same hash.

1.6 Important Definitions

(a) **Node ID** - It is the hash of public key generated for the peer-to-peer device in the network during the initial boot-up of the system.For Brihaspati-4 Node ID is of 40 nibbles(160 bits) and is generated by authentication manager API.

(b) **Layer ID** - When more services are introduced to Brihaspati-4 they will be associated with layer IDs i.e one layer ID for a specific service. The routing manager will maintain

routing tables separately for each layer ID depending upon whether user has subscribed to the layer(service).

(c) **Bootstrap Node** - A predetermined node in the network called the bootstrap node is in charge of helping any new nodes that join it. In order to obtain its routing table, every newly connected node will first come in touch with the bootstrap node.

(d) **Root Node** - Root nodes are those nodes in the network responsible for holding the key-value pair for any keys that lie between itself and its predecessor.

(e) **Index** - Collection of references to the keys in a Node on the Overlay for which this Node is the Root Node is called index.

(f) **Resource Query** - When a node searches for another node/resource in the Overlay, it generates a query in the form of an (XML) message, called Resources Query.

(g) **Routing Table Exchange** - Routing table exchange happens between different nodes in the network over a while to attain an optimized routing state.

(h) **Routing Table** - Routing table is the table present inside all network nodes, which will route the data or resources from one node to another until it reaches the final destination.

(i) **Neighbour Table** - The collection of the first sixteen closest nodes are called the neighbour table for any particular node. These nodes are decided based on the RTT (Round Trip Time) value.

Chapter 2

Chord Algorithm

2.1 Introduction

In the area of distributed systems, where multiple computers collaborate to achieve a common goal, efficient and scalable methods for locating data are essential. The Chord algorithm stands as a beacon in this landscape, offering a decentralized approach to efficiently locate data in a peer-to-peer (P2P) network.Conceived by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan in 2001, the Chord algorithm provides a distributed hash table (DHT) protocol[3]. It was designed to offer a balance between simplicity, scalability, and fault tolerance, making it an attractive choice for building P2P systems ranging from file-sharing networks to distributed databases.

At its core, the Chord algorithm provides a way to map keys onto nodes in a network, allowing for efficient lookup of data given its key. Its design enables logarithmic lookup time with respect to the number of nodes in the network, irrespective of its size, making it highly scalable. Additionally, Chord maintains its efficiency and robustness even in the face of node failures and joins, making it suitable for dynamic and large-scale environments. In the following sections, we delve deeper into its workings, exploring how it achieves its objectives and examining its strengths and limitations in practical implementations. The Chord protocol describes how to locate keys, how new nodes join the network, and how to recover in the event that an existing node fails (or leaves on schedule). Fundamentally, Chord allows for the quick distributed computation of a hash function that assigns keys to the

nodes in charge of them. Consistent hashing is used, which offers a number of advantageous characteristics [1]. The hash function balances load (every node receives approximately the same number of keys) with a high probability. Only a small portion of the keys are relocated to a new location when a node joins or exits the network, which is also very likely to happen.

2.2 Consistent Hashing

Consistent hashing is a technique used in distributed computing and distributed storage systems to efficiently distribute data across multiple nodes while minimizing the amount of data that needs to be moved when nodes are added or removed from the system. It is particularly useful in scenarios like load balancing, caching, and distributed databases.

In a distributed system, data is typically partitioned across a set of nodes using some hash function. This hash function maps each piece of data to a node in the system based on its key. However, in a dynamic environment where nodes can join or leave the system frequently, traditional hashing techniques can lead to significant data redistribution whenever the set of nodes changes. This redistribution can be inefficient, resource-intensive, and can cause performance bottlenecks.Consistent hashing addresses this issue by providing a way to map both data and nodes onto a common hash space, typically represented as a ring. It works as follows:

(a) **Hash Space Representation** - Imagine a circle (or ring) representing the entire range of possible hash values. Each node in the system and each piece of data are mapped onto this circle using a hash function, typically a cryptographic hash function like MD5 or SHA-1 [1].

(b) **Node Placement** - Each node in the system is represented by one or more points on the hash ring. These points are determined by applying the hash function to the node's identifier (e.g., its IP address or a unique identifier). The number of points assigned to each node is usually proportional to its capacity or weight in the system.

(c) **Data Placement** - Similarly, each piece of data is hashed using the same hash function, and its resulting hash value is mapped onto the hash ring. The data is then assigned to

the next node clockwise from its hash value on the ring until it reaches the first node encountered.

(d) **Handling Node Additions and Removals** - When a new node joins the system or an existing node leaves the system, only a fraction of the data needs to be moved. This is because each node is responsible for a range of hash values, and the departure or arrival of a node only affects the ranges adjacent to it on the hash ring. Data that was previously assigned to the departing node is remapped to its successor(s) on the ring.

(e) **Load Balancing** - Consistent hashing also provides inherent load balancing. Since each node is responsible for a range of hash values, the data is distributed evenly across the nodes, regardless of their capacity or performance.

(f) **Replication** - To provide fault tolerance and data redundancy, consistent hashing can be combined with replication strategies. Each piece of data can be replicated on multiple nodes by placing it on multiple consecutive nodes on the ring. This ensures that even if a node fails, the data can still be retrieved from its replicas.

2.3 Chord

Chord determines which key-value pairs are saved at which nodes by using consistent hashing techniques, like SHA-1[1]. A node ID with the same number of bits as the hash value is another feature that uniquely identifies each node. Every node is arranged in a virtual ring, with the node with the highest node ID being smaller than the current node and the node with the smallest node ID being higher than the current node as the predecessor [2]. In order to identify the responsible node for a given key-value combination, a node first determines if the key's hash is between it and its predecessor.If so, the node in question is the root node, also known as the responsible node; if not, it can forward the search query to its successor. If N nodes are present, this kind of search will require an average of N/2 hops. This will result in an O(N) search complexity. In order to speed up the search for the node ID that is closest to the hash value, Chord uses what are known as finger tables. Finger tables keep track of extra routing data. Typically, it includes node entries that are exponentially more away from

the present node. Predecessor, successor, and middle entry (entry N/2 distance away from the current node) are the minimum number of entries that finger tables can include.

In any DHT algorithm we have to ensure following conditions:

(a) When a peer-to-peer network node dies, it shouldn't happen that other nodes in the network become disconnected as a result of this node.Network Partitioning should never happen in a DHT network.In DHTs, a method known as logarithmic partitioning is employed to prevent network partitioning.

(b) Routing algorithm implemented enables search for a node closest to hash(key). If this is not ensures then the DHT network will never be able to converge.

(c) Everytime query forwarded to node closest to hash(key).The search terminates at a node which is closest to hash(key). It is known as Query forwarding mechanism.

2.4 Routing in Chord

In Chord the nodes are arranged in a circularly fashion and has asymmetric distance between nodes. It means distance from node A to node B is not equal to distance from node B to node A. When we decide a root node we have specifically mention that we are looking for root of the hash ID, distance from hash ID to node id to be minimized or distance from node ID to hash id to be minimized.Depending on this condition routing tables will be framed. In routing mechanism node will look at the hash id which has to be routed to the root of it and will search in routing table or neighbourhood table to find the node which is closest to root node of routing table and send the query to it. This process will continue until the root node is reached. When a node finds out that it is closest to hash id and no other entry exists closer to hash id in routing table then it confirms that it is the root node for the queried hash id. Then it communicates to the query originating node via DHT route or directly through IP address.

Once the node finds out that it is the root node then it looks up into it,s own database and find out that key(phone number/email) is present in database or not. If it is there then it will check the corresponding end point address and send this end point address to node ID

which has generated the query message through same DHT network. If the Query message has sender's IP address then node can directly communicate to sender about the end point address.Hash ID or node id can be of any size, in our Brihaspati 4 project we have hash id of size 160 bits that is there are 2^{160} possibilities or hash id space.Chord means circular DHT and it is a logical organisation which means that continuous connection need not to be maintained.Each node knows how to contact the neighbour nodes by knowing their end point address for the communication.

In the example of chord we have assumed 16 bit node id that is we have 2^{16} hash ID space. The node directly in front of you is called the successor node, while the one directly behind you is known as the predecessor node when going clockwise in a circular arrangement.

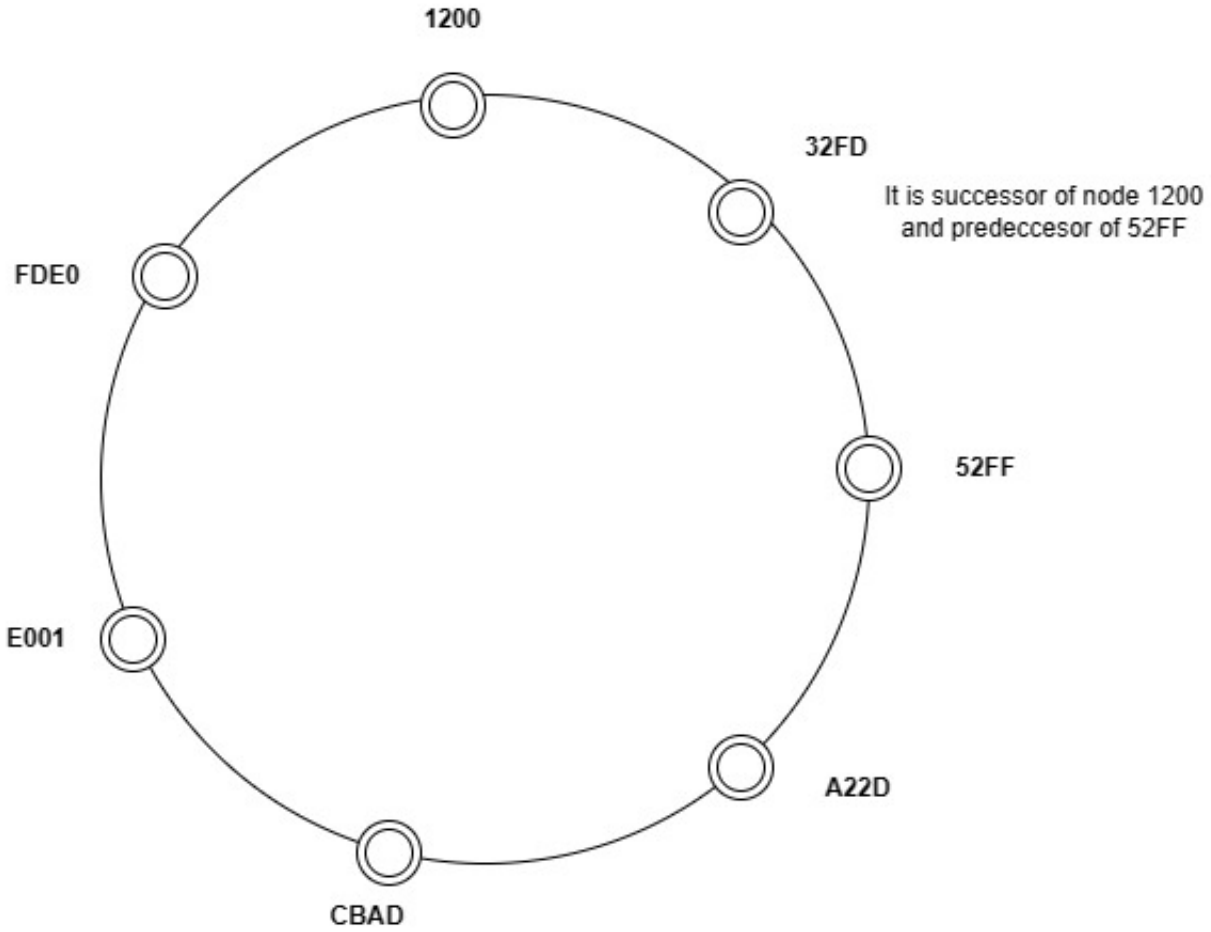

Figure 2.1: Chord Example

In above example each node id is represented by 4 digits and each digit is hexadecimal, hence a node id made up of total 16 bits(4*4).A node id is responsible for storing all <key,value> pairs for hash(key) greater than predecessor and less than or equal to node id. Node ID 1200 is responsible for storing key value pairs for all node id greater than 1200 and $\leq$ 32FD. In these key value pairs, key can be a phone number or e mail address whereas the value is end point address.

2.4.1 Joining of a node in chord

If any new node wants to join a chord network then first pre-requisite is to generate a random node id for itself. It will further connect to Bootstrap node for obtaining network information. It will find if node id is duplicate then it will generate a new node id and re-attempt the connection. A query is passed in the network to find Root(Node ID). A node checks if hash id between predecessor is less than or equal to current node then search terminates and current node is the root node. Otherwise, query is passed on the next node.Let us consider an example on how new node joins a chord DHT network.

Node 10 is the new node which wants to join the DHT network. Initially it passes a query to Node 1 which is Bootstrap node.Search query was passed by node 1 to find root of node 10. Node 13 informs node 1 that Root(10)=13. Hence, node ID 10 is not a duplicate node and can be used. Node 1 informs the new node about it,s root node and node 10 further informs node id 13 about insertion. Node 13 informs node 10 about it's predecessor and informs node 9 about node 10.

Node 9 will change it's successor pointer to new node 10 and node 10 will make node 9 as predecessor and mode 13 as successor. Resulting chord network after insertion of new node is shown below. On similar lines node can also be removed from DHT network.

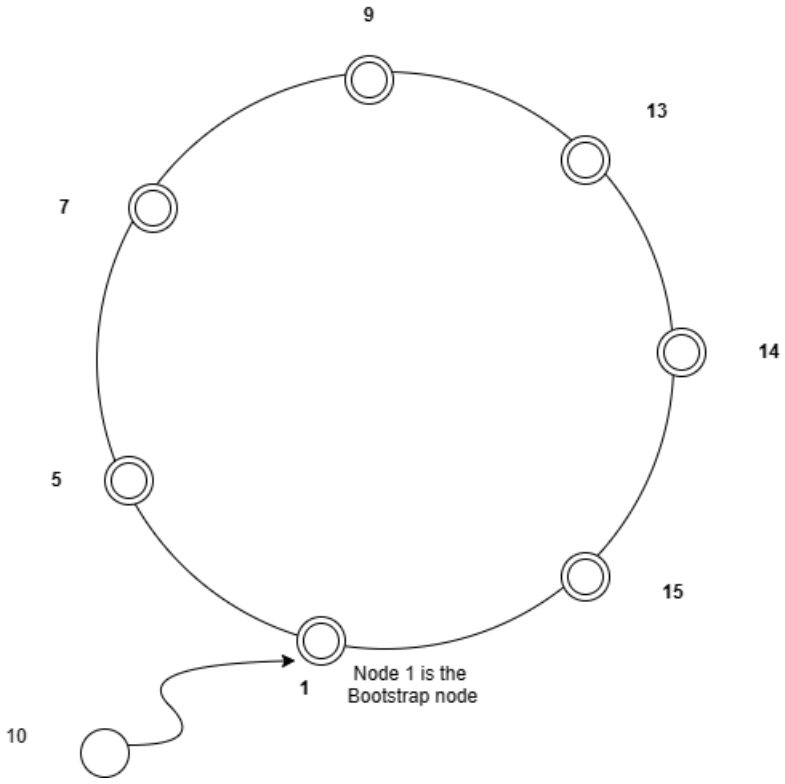

Figure 2.2: Chord Example

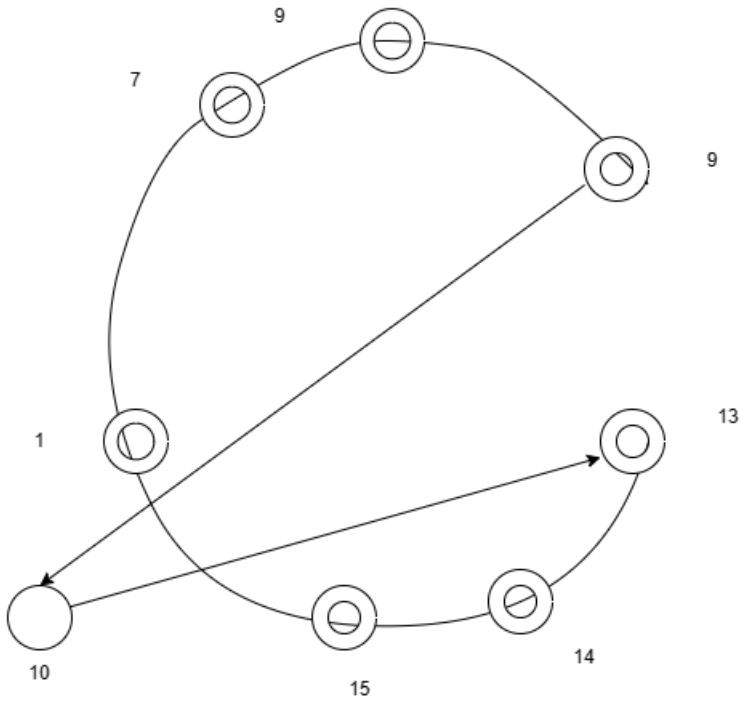

Figure 2.3: Nodes changing Pointers on insertion

2.4.2 Finger Tables

The message will arrive at the intended node after an average of N/2 hops if there are many nodes (N). The network will operate very slowly and inefficiently if N is really large. Finger Tables are kept in chord to mitigate this and speed up message routing. In finger tables each node keep information about other nodes.

A node x can maintain the finger table as Root[(x+2^{n-1})$mod 2^n$], Root[$(x + 2^{n-2})$ mod 2^n], Root[$(x + 2^{n-2})$ mod 2^n].....Root[$(x + 2^{n-2})$ mod 2^n].

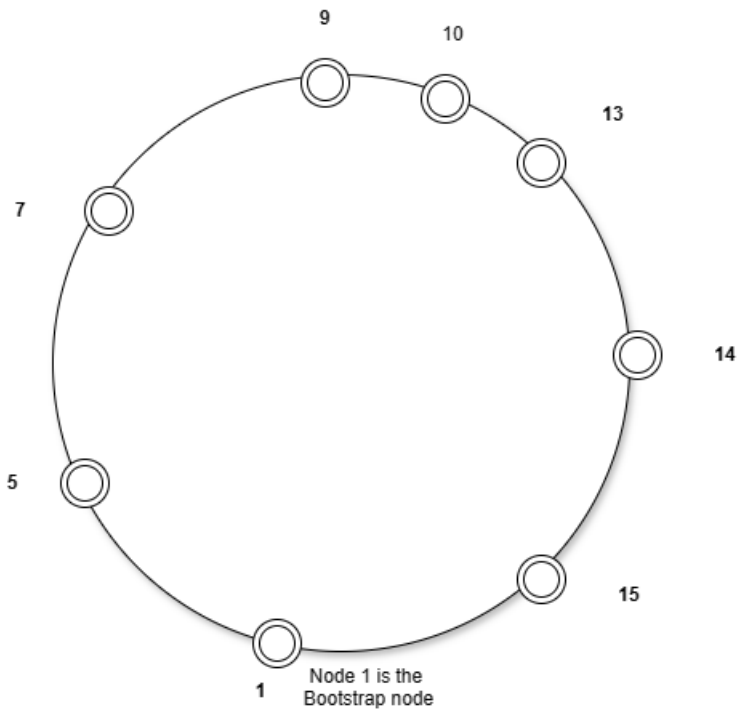

Figure 2.4: Chord Network after insertion of new node

i	Root[9+2^i] mod 2^4
0	10
1	10
2	12
3	5

Figure 2.5: Finger Table for Node 9

i	Root[5+2^i] mod 2^4
0	7
1	7
2	9
3	12

Figure 2.6: Finger Table for Node 5

Every node will have a finger table in this way, and the query will be sent straight to the root node. It significantly lowers the number of hops and increases the network's communication efficiency. Maintaining information about every other node in finger tables is not necessary; the number of nodes can be chosen based on the network's performance criteria. The more nodes in the Finger tables, the more memory they will use, but the routing will be faster because fewer hops will be needed.

Chapter 3

Tapestry Algorithm

3.1 Introduction

The stated objectives of Tapestry are fault-resilience, self-management, and adaptability. Even with high network traffic, this routing architecture can route requests rather effectively. Randomness not only achieves routing locality but also gracefully distributes the load. Due to a certain format that is used and adhered to by all nodes, information about data that a node possesses is also known to other nodes and is interpretable by each node. With this format, data may be found quickly and easily; the querying node just has to know the node id. When Chord builds its routing overlay, it either cannot or does not account for network distances; as a result, if a query is answered for the subsequent hop, it can really be lengthy enough to cross the whole range of the concerned P2P network. Chord uses the general formulation that is populated during runtime and the shortest overlay hops available to route queries. However, Tapestry has the ability to create routing tables that are optimal and relevant locally. It then updates these tables on a regular basis and effectively applies the revised table to reduce routing distance.

In order to distribute objects and resources in a distributed network efficiently, Plaxton and Rajamaran devised a mechanism in 1997. This paper was released prior to the introduction of the peer-to-peer system concept. The fundamental concept used prefix-oriented routing with a predetermined number of potential nodes.Every node involved in the Overlay in Tapestry is given a unique Node ID. The Node ID has been randomly selected from a

vast identifier space. "Tapestry is a peer-to-peer (P2P) overlay network that provides high-performance, scalable, and location-independent routing of messages to close-by endpoints, using only localized resources. The focus on routing brings with it a desire for efficiency: minimizing message latency and maximizing message throughput."[Tapestry.]

3.2 Routing in Tapestry

While some systems provide a DHT interface that fixes the number and placement of item copies, Tapestry gives applications the freedom to arrange objects as needed. In order to enable effective routing to objects with low network stretch, Tapestry "publishes" location pointers throughout the network. Tapestry cab be considered as an extension if Kademelia algorithm. IN Kademlia we have binary bits so hash id space is divided into 2 parts each time.Since Tapestry was conceptualized for hexadecimal bits, the hash ID space was always divided into 16 pieces.However, this concept of tapestry can be extended to any base of digits. Now, let's look at the hexadecimal hash ID $b_{15}.b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_9b_7b_6b_5b_4b_3b_2b_1b_0$.The hash id space will be divided into 16 partitions in the first level, numbered b_0 through b_{15} for each partition. . First partition will start from $b_0$xxxxxxxxxxxxxxxx second will start from $b_1$xxxxxxxxxxxxxxx upto b_{15}xxxxxxxxxxxxxxxx. Now these each of 16 partition will be further mapped to rings of 16 partition each. Secon level partition will begin from $b_0b_0$xxxxxxxxxxxxxxx upto b_0b_{15}xxx..xxx. Until we arrive to partition mapping to a single hash ID, this will go on for up to 16 layers in case of hexadecimal system.

Since Tapestry's efficiency increases with network size, sharing a single, sizable Tapestry overlay network is beneficial for several applications.Below is the example which illustrates how the partitioning of hash id space is done at various levels while using hexadecimal digits.

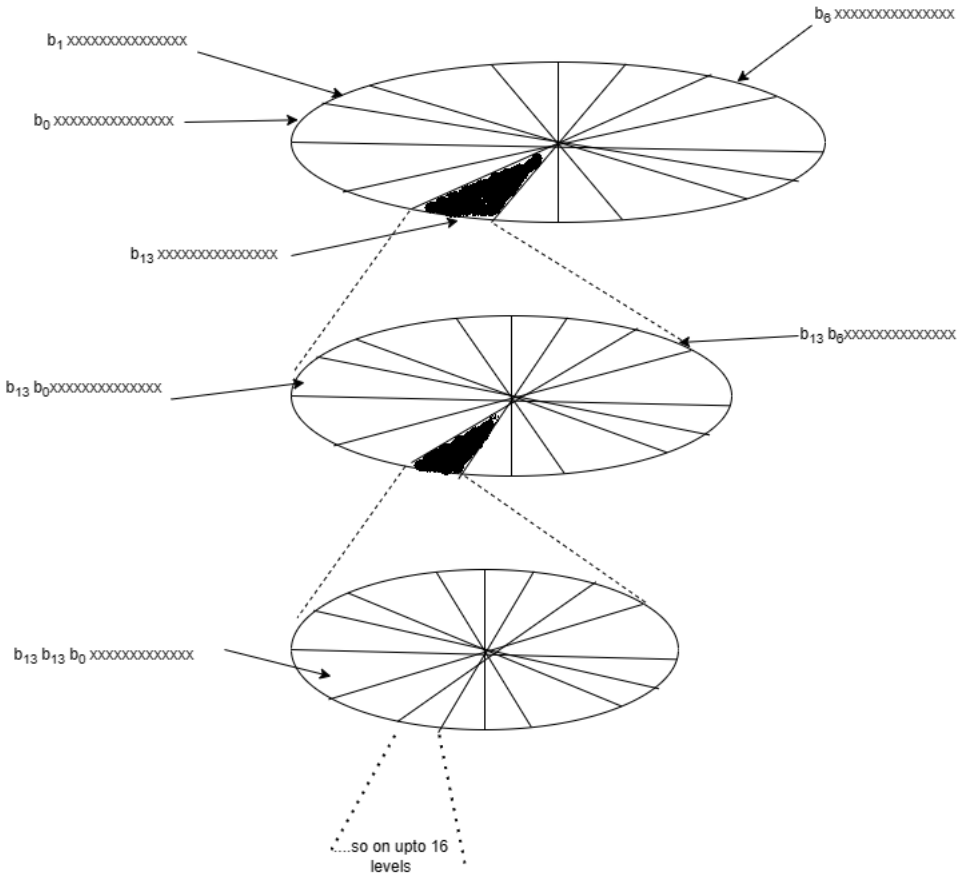

Figure 3.1: Partitioning of hash id space in Tapestry

3.3 Routing Table Structure in Tapestry

The Tapestry method routes messages and queries to their target by utilizing local routing tables at each node. The neighbor map and related routing table that node "5128" uses to determine the next closest hop are displayed in Figure 3.2. Additionally, the graphic illustrates the various degrees of connectivity that node "5128" uses with the other nodes (L1, L2, L3, etc.).

(a) **Resource Publication** In Tapestry each hash id from hash id space will be mapped to a unique node called Root node.Root node is responsible for storing all the information regarding the range of hash id it is responsible for. Fig 3.3 Illustrates the search path

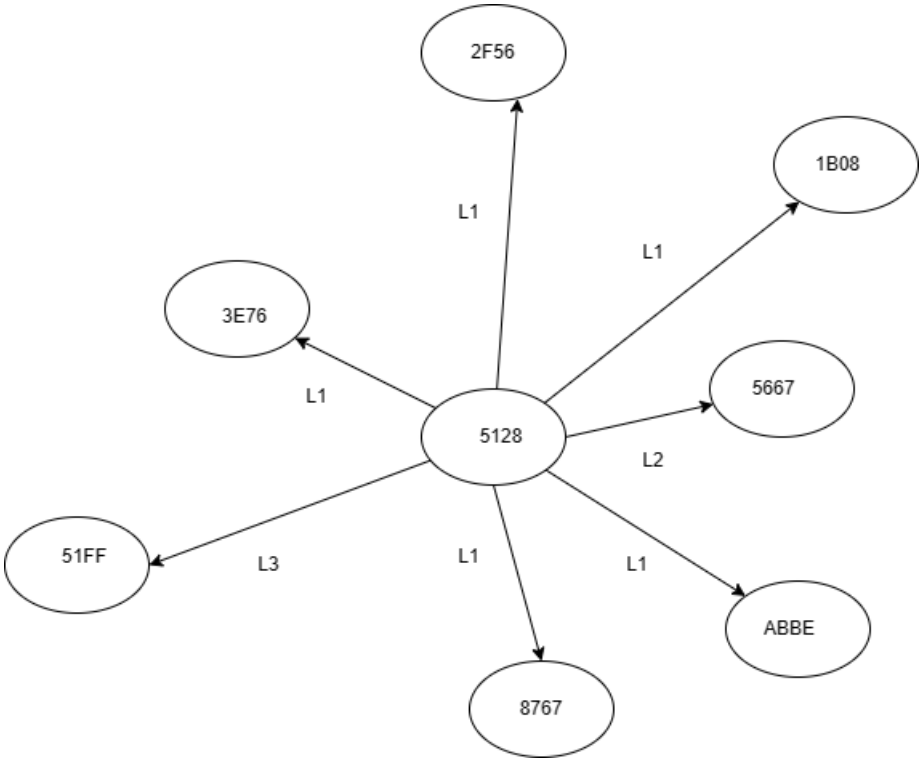

Figure 3.2: Neighbours in Tapestry

Level	1	2	3	4	5	6	7	8
1	1543	2BCD				6444	7BCD	
2			5376					58B6
3	51ED			511F			517B	
4	5121							5128

Table 3.1: Routing Table in Tapestry

by two different nodes to find the same information.During Resource query resolution, the location pointer to the node where this Resource 4378 can be found is stored in each intermediate node in these two pathways.

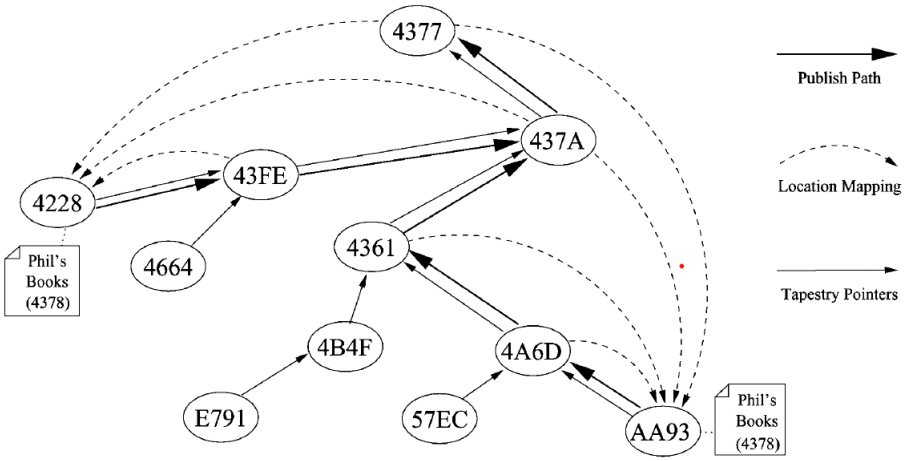

Figure 3.3: Object Publication

(b) **Discovery of Resource** - The resource query for Resource 4378 will be directed toward the root node of 4378 (i.e., IR = 4377) if several nodes search for it from various locations within the network. As seen in Figure 3.4, when they reach the publishing path, they grab onto and follow the pointer to the closest copy of the Hash Id.

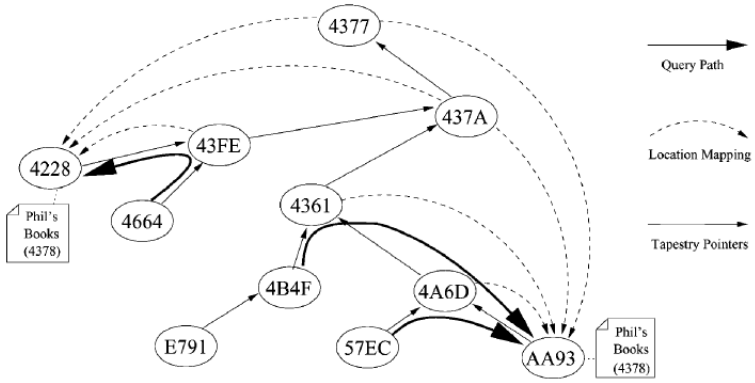

Figure 3.4: Resource Discovery

(c) **Routing** - The message must be forwarded via neighbor connections to nodes whose

node IDs are becoming closer to the root node in order for it to be routed to the root node IR. By matching the larger prefixes, it is achieved. By matching nibbles or characters one at a time from the most significant nibble (MSN) to the least significant nibble (LSN), Tapestry routes overlay messages to the destination ID using a routing table [5].

The primary distinction between Chord and Tapestry lies in the fact that Chord builds a routing overlay without taking network lengths into account. The next hop can therefore cover the circle's diameter, or go straight across it. Nevertheless, the Chord uses runtime heuristics to help and routes the data on the shortest hop that is available. Conversely, Tapestry creates and maintains locally optimal routing tables based on its implementation.

Chapter 4

Chord-Tapestry Hybrid Algorithm

4.1 Introduction

In previous two chapters we have explained about the two famous algorithms DHT namely Chord and Tapestry.These two algorithms can be enhanced to boost routing effectiveness and resilience in a DHT network, although they do have certain drawbacks. Therefore, a way to employ the current methods with modifications to create a stronger and more reliable routing algorithm needs to be devised. In an attempt to overcome the constraints faced by Chord and Tapestry, this chapter suggests a novel algorithm that will do away with all of the restrictions and boost the DHT network's efficiency and resilience. This chapter provides an analytical proposal of the Chord-Tapestry Hybrid Algorithm and provides a brief summary of both Chord and Tapestry, emphasizing their respective limitations.

4.2 Chord-Tapestry

Each node in the chord Tapestry Hybrid (CTH) algorithm will retain an incremental suffix-based routing table. There would only be three entries for that character in each column: Predecessor (P), Successor (S), and Middle (M).Every entry in the routing table will be an object of the B4 Node class, including the following details about each node: its public key, its transport address, its IP address, its port address, its hash ID, and its node ID.

4.2.1 Routing Table Population Rule

The routing table of the current node will be combined with the Node IDs that were collected from the routing tables of the other nodes. The Routing Table's entry addition procedure is as follows:

(a) Every column in the Routing Table denotes a ring of 16 (2^4)position(i.e. (0 to F) as shown in Table 4.1.

Level	**0**	**1**	**2**	**3**
Predecessor				
Successor				
Middle				

Table 4.1: Routing Table in Tapestry

(b) The self Node ID(local node id) and the incoming Node ID(which comes from another routing table) is used to merge with the local node, are compared nibble by nibble to determine if they match or don't match from Most Significant to Least Significant Nibble. The location where a first non-match is discovered in the column is where the incoming Node ID is added.

(c) Only in that specific column can an incoming Node ID replace an already-existing Node at the Predecessor, Successor, or Middle positions. It may occur if its nibble at that location more closely matches a predetermined set of logic requirements than the nibble at that location of the already existing node in the predecessor, successor, or middle nodes. Otherwise, the current entry is kept. In the routing table, the first entries are all null.When a node updates in one specific location within a column, its ID is replicated in the other two locations within the same column. A column may include a node id in each of the three locations, or it may be null in all three. You cannot have a partially empty column.

(d) The incoming Node ID and self Node ID are compared nibble by nibble for a match/mismatch. The first column, where a non-match found, is considered for adding incoming Node

ID. Suppose some other nodes occupy all the position in that particular column. In that case, the incoming node is compared with the nibble of already existing nodes(Predecessor, Successor and Middle) to check whether it lies close to the self node. If it is true, then a particular predecessor or successor node is replaced with the incoming Node. For the Middle column entry, the node closest to the opposite end of the nibble position of the self node is considered. So at every column, we will have nodes closer to the self node corresponding to that particular nibble position.

(e) In case where incoming node id during nibble match happens to be at a column which is already populated at all three locations (Predecessor, Successor and Middle), in this case closeness of the incoming node id and that of already existing node ids in routing table to local node is is calculated. The node id which is closest to local node id is kept in the table.

4.2.2 Merging of Routing Table Example

Initially when a node wants to join the network it sends a message to Bootstrap node. Bootstrap node is well advertised nodes for every peer network and all nodes know about it. After receiving the message Bootstrap node sends copy of it's routing table to new node and also update it's own routing table. Let us assume our Bootsrap node id is "EFA2" and routing table of bootstrap is shown in table below.

Level	**0**	**1**	**2**	**3**
Predecessor	**B4FF**	**E612**	**-**	**-**
Successor	**3A88**	**E612**	**-**	**-**
Middle	**8B4A**	**E612**	**-**	**-**

Table 4.2: Routing Table of Bootstrap

(a) Let us assume the node id of joining node is "62D6". When the node sends request to Bootstrap node, the bootstrap node updates it's own routing table with new node.During updation there is mismatch at first nibble thereby updation will be done in first column.Now it will match firstly with predecessor node id B4FF, but it is already

a better match as B4FF is more closer to EFA2 that 62D6 for condition of predecessor. Then it will match with successor node id 3A88, but it is also a better match than incoming node id.Now further it will match with mid node id in routing table 8B4A. In this case the ideal mid node id shall start with number 6 because ((E+8) mod 16) is equal to 6. Bootstrap routing table will be updated and new mid node id will be 62D6.

Level	**0**	**1**	**2**	**3**
Predecessor	**B4FF**	**E612**	**-**	**-**
Successor	**3A88**	**E612**	**-**	**-**
Middle	**62D6**	**E612**	**-**	**-**

Table 4.3: Updated Routing Table of Bootstrap

(c) When node 62D6 send request to Bootstrap for joining the network. In the very first step bootstrap send a copy of it's own routing table to 62D6 . Initially the routing table of 62D6 contains all null entries.When copy of Bootstrap routing table is received then 62D6 node update it's own routing table.

Level	**0**	**1**	**2**	**3**
Predecessor	**-**	**-**	**-**	**-**
Successor	**-**	**-**	**-**	**-**
Middle	**-**	**-**	**-**	**-**

Table 4.4: Initial Routing Table of node 62D6

(c) The routing table of node 62D6 will receive following nodes B4FF,3A88,8B4A,E612.All node ids have mis-match at first nibble therefore all nodes will compete for place in first column. Due to distance criteria predecessor node will be 3A88, successor will be 8B4A and mid node will be EFA2.The routing table of node id 62D6 after first exchange will be as follows:

(d) Let us assume node 62D6 learned about other nodes 62FF and 62F1.All node IDs have

Level	**0**	**1**	**2**	**3**
Predecessor	**3A88**	**-**	**-**	**-**
Successor	**8B4A**	**-**	**-**	**-**
Middle	**EFA2**	**-**	**-**	**-**

Table 4.5: Routing Table of node 62D6 after first exchange

first mis match at third nibble. Hence,updation in third column of routing table will be done. By Distance criteria 62F1 will become successor node, 62F1 will be mid node and 62FF will be predecessor node.

Level	**0**	**1**	**2**	**3**
Predecessor	**3A88**	**-**	**62FF**	**-**
Successor	**8B4A**	**-**	**62F1**	**-**
Middle	**EFA2**	**-**	**62F1**	**-**

Table 4.6: Routing Table of node 62D6 after second exchange

Chapter 5

Implementation of Routing Manager for Brishaspati-4

5.1 Introduction

In the previous chapter we learned about the chord tapestry hybrid algorithm.The various steps involved in the algorithm, like creating the routing table, initialisation, population and merging, were clearly explained.Now we will implement this chord tapestry hybrid as Routing Manager API. We will use Flutter Dart language to implement routing manager.

Dart is a modern, object-oriented programming language developed by Google. Designed for building both mobile and web applications, it is known for its speed, efficiency, and ease of use. Dart's syntax is similar to other C-style languages, making it accessible for developers familiar with Java, JavaScript, or C++.

5.2 Salient Features of Dart

Dart is a versatile language that strikes a balance between development speed and execution performance. Its close integration with Flutter has made it a popular choice for developers looking to create high-quality, cross-platform applications. With continuous improvements and a strong community, Dart remains a robust option for modern app development.

(a) Dart, combined with Flutter, is widely used for developing cross-platform mobile ap-

plications with a single codebase.

(b) Asynchronous programming is supported by default in Dart. This implies that programmers can create code that runs in parallel without interrupting the operation of other programs. Applications that use asynchronous programming are more responsive and efficient because it can handle tasks like file operations, network requests, and user interactions.

(c) Dart provides a range of deployment options. It is applicable to the development of desktop, online, and mobile apps. Because of this versatility, developers can create apps that run on a variety of devices and appeal to a wider user base.

(d) A vast array of pre-built libraries and packages in Dart offer readily usable functionality for frequent tasks. File input/output, network connectivity, data manipulation, user interface creation, and other topics are covered by these libraries. Developers can write strong apps more quickly and with less effort by utilizing these frameworks.

5.3 Workflow of Routing Manager

In figure 5.1 workflow of routing manager has been illustrated.Initially object of routing manager is created and it checks if any routing table file is already existing.If file is exists then it loads the file and check for aliveness of nodes and send copies of own routing table to alive nodes. If no such file exists then it create routing table for all layers and initialise them with null values.Further it send message to Bootstrap node and receives it's routing table, node updates it's own routing table from bootstrap routing table.Concurrently it keeps checking for messages in Communication Manager Buffer. If any message for Routing Manager exists then it take out the message and update it's own routing table. Periodically node sends copies of it's own routing table to nodes existing in routing table.

5.4 Implemented Classes

In previous section it is mentioned that Routing Manager API has been coded in Dart language.Dart is an object-oriented language and utilizes classes to define the blueprint for

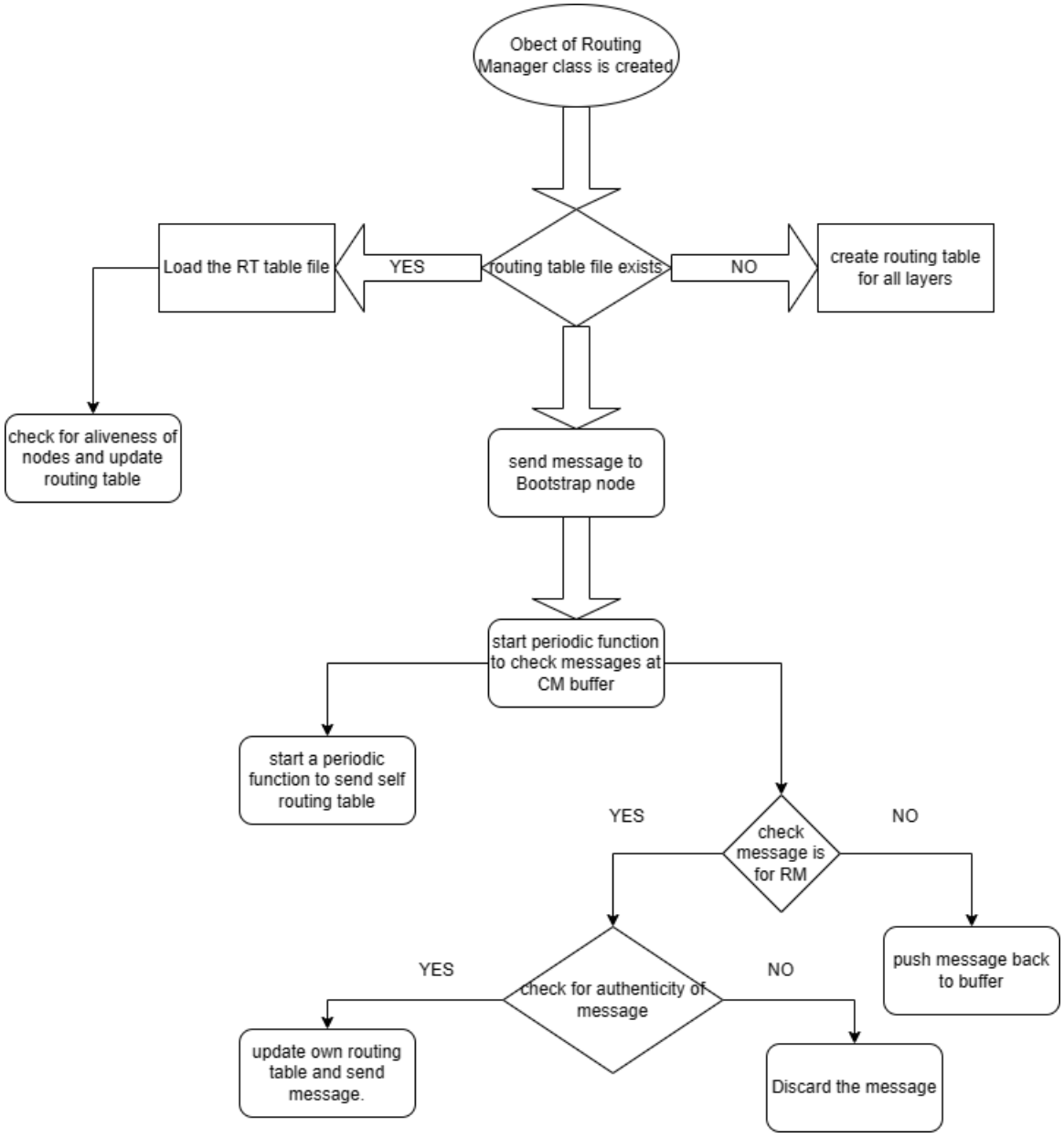

Figure 5.1: Workflow of Routing Manager

objects. This object-oriented nature allows Dart to encapsulate data and behavior within objects, facilitating modular, reusable, and maintainable code.Various classes has been implemented to achieve the functionality of routing manager. These classes ang it's methods are explained in further sections.

5.5 B4RTTable class

The object of this class holds various types of routing tables such as main routing table, neighbour tables, latitude-longitude table. Method to find next hop for the incoming query is also implemented in this class.This class has one constructor which accepts one argument.Various functions of class are explained in detail in subsequent section.

5.5.1 updateRtTable()

This function receives two arguments, first local routing table and second incoming routing table. Next, iterating through the entire node routing table, it modifies the routing table for every entry. It compares each nibble by nibble throughout each iteration to determine the index at where the local node ID and the inbound node ID initially differ. To enable nibble by nibble comparison the node is first split in the form of list of string and then each of this entry of list is converted to decimal integer from hexadecimal integer. This index provides the column index where the update will take place.Three entries—a predecessor node, a mid node, and a successor node—make up a column.The arriving node's distance is calculated using each of the three column entries, and it is compared to the local node's distance and the three entries in the rotating table, in that order.

The specific table record is changed with the new node ID if a better match is discovered. The routing table stays the same in all other cases.The algorithm for the same has been written as Algorithm 1 and Algorithm 2.

Algorithm 1: Update Routing Table Algorithm

Input: localRTtable (3x40 list), rtTable (3x40 list)

Output: Updated localRTtable

Algorithm `UpdateRouting(`*localRTtable, rtTable*`)`

```
RT ← localRTtable;
for each row from 0 to 2 do
    for each col from 0 to 39 do
        if rtTable[row][col] ≠ null and rtTable[row][col].hashID ≠
         localIdb.nodeid.hashID then
            nodeIdC ← split(rtTable[row][col].hashID);
            localNodeIdC ← split(localIdb.nodeid.hashID);
            m ← −1;
            for each i from 0 to 39 do
                if nodeIdC[i] ≠ localNodeIdC[i] then
                    m ← i ; break;
                end
            end
            if RT[0][m] = null and RT[1][m] = null and RT[2][m] = null then
                RT[0][m] ← rtTable[row][col];RT[1][m] ← rtTable[row][col];
                RT[2][m] ← rtTable[row][col];
            end
            else
                preNodeId ← RT[2][m].hashID;
                midNodeId ← RT[1][m].hashID; sucNodeId ← RT[0][m].hashID;
                UpdateLocation(preNodeId, midNodeId, sucNodeId, nodeIdC,m);
            end
        end
    end
end
```

Algorithm 2: Update node id in routing table

Procedure `UpdateLocation(`*preNodeId, midNodeId, sucNodeId, nodeIdC, m*`)`

preNodeIdC ← split(preNodeId);
midNodeIdC ← split(midNodeId);
sucNodeIdC ← split(sucNodeId);
preNodeIdint ← int.parse(preNodeIdC[m], radix: 16);
midNodeIdint ← int.parse(midNodeIdC[m], radix: 16);
sucNodeIdint ← int.parse(sucNodeIdC[m], radix: 16);
localnodeIdint ← int.parse(localNodeIdC[m], radix: 16);
nodeIdint ← int.parse(nodeIdC[m], radix: 16);
idealMidNodeIdint ← (localnodeIdint + 8) % 16;
if $((localnodeIdint - preNodeIdint + 16)\%16) > ((nodeIdint - preNodeIdint + 16)\%16)$ **then**
  if *mRtt contains* `RT[2][m]` *and* (`RT[2][m]` $\neq$ `RT[1][m]` *or* `RT[2][m]` $\neq$ `RT[0][m]`) **then**
    `mRtt.remove`(RT[2][m]);
  end
  RT[2][m] ← rtTable[row][col];
end
else if $((sucNodeIdint - localnodeIdint + 16)\%16) > ((nodeIdint - localnodeIdint + 16)\%16)$ **then**
  if *mRtt contains* `RT[0][m]` *and* (`RT[0][m]` $\neq$ `RT[1][m]` *or* `RT[0][m]` $\neq$ `RT[2][m]`) **then**
    `mRtt.remove`(RT[0][m]);
  end
  RT[0][m] ← rtTable[row][col];
end
else if $\min((idealMidNodeIdint - midNodeIdint + 16)\%16, (midNodeIdint - idealMidNodeIdint + 16)\%16) > \min((idealMidNodeIdint - nodeIdint + 16)\%16, (nodeIdint - idealMidNodeIdint + 16)\%16)$ **then**
  RT[1][m] ← rtTable[row][col];
end

5.5.2 nexthop()

Node ID and local routing table are the two arguments required by the next hop function.In DHT routing, it gives back the node id of the subsequent hop.This routing table function is crucial since it provides the hop node id, which is the basis for query passing in DHT. The local node is the root node if hash id and local node id are equivalent.If not, the first mismatch index between the hash ID and the local node ID will be discovered.The index of the column containing the next hop node ID will be provided by the mismatch index.Due to the fact that we no longer have to examine and match each node in the table, this strategy has increased efficiency and decreased code run time.The next hip node id might be either the successor node, the mid node, or the predecessor node once the mismatch index has been determined. The hash ID distance to each of these nodes is computed, and the lowest of the three distances is recorded.The hash ID and local ID distances are now computed and compared to the minimal distance that was saved. The specific table entry is returned as the next hop if the minimum distance is smaller than the distance between the hash ID and the local node id; else, the local node is the destination.

Algorithm 3: Find Next Hop in DHT

Input: localNodeId, hashID, localRTtable

Output: Next Hop ID

Algorithm `UpdateRouting`(*localIdb, hashID, localRTtable*)

```
if localnodeID == hashID then
    return localnodeID ;                          // current node is the root node
end
hashIdC ← split(hashID);
localNodeIdC ← split(localNodeId);
distanceHashId ← [0, 0, 0];
distanceLocalID;
misMatch ← −1;
l ← −1, i ← −1;
for i ← 0 to 39 do
    if hashIdC[i] ≠ localNodeIdC[i] then
        misMatch ← i;
        if localRTtable[0][i] == null then
            for l ← i to 39 do
                if localRTtable[0][l] == null or hashIdC[l] == localNodeIdC[l] then
                    misMatch++;
                    if misMatch == 40 then
                        return localNodeId;
                    end
                end
                else
                    l ← 40;
                end
            end
        end
        i ← 40;
        findHop (localRtTable,misMatch);
    end
end
```

Algorithm 4: Find Hop FUnction Definiton

Input: localRTtable , misMatch

Output: nextHopID

$preNodeId \leftarrow localRTtable[2][misMatch].hashID$

$midNodeId \leftarrow localRTtable[1][misMatch].hashID$

$sucNodeId \leftarrow localRTtable[0][misMatch].hashID$

$preNodeIdC \leftarrow \text{split}(preNodeId)$

$midNodeIdC \leftarrow \text{split}(midNodeId)$

$sucNodeIdC \leftarrow \text{split}(sucNodeId)$

$distance[0] \leftarrow calculateDistance(preNodeIdC[misMatch], hashIdC[misMatch])$

$distance[1] \leftarrow calculateDistance(midNodeIdC[misMatch], hashIdC[misMatch])$

$distance[2] \leftarrow calculateDistance(sucNodeIdC[misMatch], hashIdC[misMatch])$

$distanceL \leftarrow calculateDistance(localNodeIdC[misMatch], hashIdC[misMatch])$

$minValue \leftarrow \min(distance[0], distance[1], distance[2])$

if $distance[index\ of\ minValue] < distanceLocalID$ **then**

  if $index\ of\ minValue == 0$ **then**

    return $preNodeId$

  end

  if $index\ of\ minValue == 1$ **then**

    return $midNodeId$

  end

  else

    return $sucNodeId$

  end

end

else

  return $localNodeId$

end

5.5.3 putOnHold()

If a node in the routing table doesn't respond to a query, it is transferred to the onHoldNodes map and removed from the routing table. This function accepts the updated local routing table and parameters. This function starts a counter that keeps track of how many times a particular node has ignored a request for information. After a node misses three attempts to respond to a query, it is deleted from the onHoldNodes map. It is initially verified that the node ID is present in the onHoldNodes mappings if a node ID is received for updating in the routing table. It is taken out from the map if it is included in it.

Algorithm 5: Node Handling Algorithm

```
Input: onHoldNodes, node, localIdb, localRTtable
if onHoldNodes ≠ null and onHoldNodes.containsKey(node) then
    if onHoldNodes[node] ≥ 2 then
        onHoldNodes.remove(node) purge the NodeId
    end
    else
        onHoldNodes[node] ← onHoldNodes[node] + 1 increment the attempts
         counter
    end
end
else
    onHoldNodes[node] ← 1 if node is not present in the map nodeIdC ←
     node.hashID.split(") localnodeId ← localIdb.nodeid.hashID localnodeIdC ←
     localnodeId.split(") for i = 0 to 39 do
        if nodeIdC[i] ≠ localnodeIdC[i] then
            for k = 0 to 2 do
                if localRTtable[k][i] == node then
                    if k == 0 then
                        localRTtable[k][i] ← localRTtable[1][i] cyclically copies the
                         previous node, where node id needs to be removed
                    end
                    else if k == 1 then
                        localRTtable[k][i] ← localRTtable[2][i]
                    end
                    else if k == 2 then
                        localRTtable[k][i] ← localRTtable[1][i]
                    end
                end
            end
            i ← 40
        end
    end
end
```

5.6 RoutingManager class

It is the Routing Manager API's core class, which manages all of its features.It is implemented as a singleton class that has the ability to manage various B4rtrtable class instances.The private constructor of this class is one. As a result, multiple copies and varieties of routing tables based on layers can be stored in one RM.This class sends and receives messages via the Communication Manager (CM) API.It obtains the local node ID and verifies the legitimacy of received messages with the aid of the Authentication Manager (AM) API.The sections that follow provide explanations for the functions implemented in this class.

5.6.1 init()

The RoutingManager class constructor calls this function. When the RoutingManager class is initialized, this function is the first one to be called. The routing table file's exit is initially verified in this function. If it does, the file is loaded and the liveliness of each routing table node is verified. Several instances of the B4rttable class are stored in the routing table according to the number of layers if the file is not present.Then it sends message to the Bootsrap node ID and initializes the function to periodically look for messages in CM buffer. This class does not receive any arguments and primary aim of this class is to initialize the Routing Manager.

5.6.2 createMessageRM()

This function receives 11 arguments and creates a message in a specific format which contains local routing table which has to be passed to other nodes. The final message framed is encoded in jSON format so that mesaage can be transmitted over a network.It's implementation logic has been explained in ALgorithm 6.

5.6.3 sendmessageRM()

This function uses Communication Manager API to send message.It first creates a message using function createMessageRM() and then send this message.

Algorithm 6: Create Message RM

Input: RM, Relay, myNodeID, hashID, s, current, R, nodeID, myEndpoint, layerID, reqRT

Output: jsonMessageRM

Function `createMessageRM(`*RM, Relay, myNodeID, hashID, s, current, R, nodeID, myEndpoint, layerID, reqRT*`):`

```
jsonRT ← routingTables[layerID].RoutingTable.map(innerList: innerList.map( nodeID:
if nodeID ≠ null then
    { 'hashID': nodeID.hashID,
    'publicKey': nodeID.pubKey.toString(),
    'sign': { 'r': nodeID.sign.r.toString(), 's': nodeID.sign.s.toString() },
    'publicKeyPem': nodeID.publicKeyPem.toString() }
else
    null
end
).toList()).toList();
jsonMyNode ← { 'pubKey': myNodeID.pubKey.toString(),
'hashID': myNodeID.hashID.toString(),
'sign': { 'r': myNodeID.sign.r.toString(), 's': myNodeID.sign.s.toString() },
'publicKeyPem': myNodeID.publicKeyPem.toString() };
jsonStringMyNode ← jsonEncode(jsonMyNode);
jsonNodesString ← jsonEncode(jsonRT);
messageRM ← { 'RM': "RM",
'Relay': "R",
'myNodeID': jsonStringMyNode,
'hashID': hashID,
's': "s",
'current': current,
'R': "R",
'nodeID': nodeID,
'myEndpoint': myEndpoint,
'reqRT': reqRT,
'layerID': layerID,
'RT': jsonNodesString };
jsonMessageRM ← jsonEncode(messageRM);
return jsonMessageRM;
```

5.6.4 mergeTables()

Route table and layerID are the two parameters passed to this function. Node IDs from the receiving routing table are updated in the local routing table.It makes use of the B4rttable class's updateRttable() function, which implements all of the logic involved in updating a routing table.RM contains numerous B4rtTable class objects. The routing table for a given layer or item is changed based on its layerID.

5.6.5 retrieveFullRT()

This function returns the complete routing table for the given layerID.

5.7 Conclusion

The hybrid Chord-Tapestry approach is included into the Dart coding framework.Each function contained a collection of test cases with corresponding test codes that were run.Each feature was assessed under a variety of conditions that might occur in an application environment found in the real world.Following the successful completion of all test cases, the Routing Manager API and Communication Manager API were integrated and tested together. At first, RM included its own routing table in messages it delivered to the Bootstrap node. Bootstrap gave its routing table back in exchange.Node and Bootstrap both exchanged and updated their own routing tables.

5.8 Future Work

The study of the Chord-Tapestry Hybrid model can go in a number of different areas, some of which could be pursued through projects that are proposed as part of the research project. These projects could include the following:

(a) There is always room for improvement in written code to preserve the economy of computational and storage resources.

(b) The Broadcasting Routing tables in conjunction with Communication Manager API is yet to be implemented for live streaming application.

(c) Research on the mathematical verification of the Chord-Tapestry Hybrid model's effectiveness in comparison to other models already in use could prove to be stimulating and interesting area of research.

Bibliography

[1] M. J. B. Robshaw, "D2, md4, md5, sha and other hash functions," *RSA Labs., vol. 4.0, Tech. Rep. TR-101.*

[2] J. S. Ben Y. Zhao, Ling Huang, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE*, 2004.