VIM - Virtual Id Based M-way Lookup Protocol for Peer-to-Peer Network

A DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF

MASTER OF ENGINEERING
(COMPUTER TECHNOLOGY & APPLICATIONS)

Submitted by

TEJ TARUN MAHINDRA

College Roll No: 25/CTA/04

Delhi University Roll No: 4005

Under the guidance of

MR. RAJEEV KUMAR

DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING, NEW DELHI
(UNIVERSITY OF DELHI)

JUNE 2007

CERTIFICATE

This is to certify that the work that is being presented in this dissertation entitled "**VIM - Virtual Id Based M-way Lookup Protocol for Peer to Peer Network**", in partial fulfillment of the requirement for the award of the degree of **Master of Engineering** in Computer Technology and Application submitted by **Mr. Tej Tarun Mahindra** (University Roll no. 4005) to the Department of Computer Engineering, Delhi College of Engineering, is an authentic record of the student's own work carried out under the supervision and guidance of the undersigned.

Mr. Rajeev Kumar
Lecturer
Department of Computer Engineering
Delhi College of Engineering
Delhi

Prof D. Roy Choudhary
Head
Department of Computer Engineering
Delhi College of Engineering
Delhi

ACKNOWLEDGEMENT

I feel pride in placing on record my deep gratitude to my honorable guide Mr. Rajeev Kumar who despite his extremely tight and busy time schedule spared enough quality time to guide me throughout the journey of this dissertation. Whether it was review of relevant literature or clearing my doubts, he helped me in solving all the problems with comfortable ease. He also encouraged and motivated me to sail through in difficult times. It is hard to imagine successful completion of such a dissertation without his guidance, care and inspiration. I am grateful to Prof D. Roy Choudhary , Prof. Goldie Gabrani , Mrs. Rajni Jindal , Mr. Manoj Sethi for their mature direction, who spared their valuable time. Their painstaking and timely guidance evokes in me natural good feelings and gratitude towards them.

The task is major and so is the contribution of friends, family members and teachers. People are my own and with a great sense of modesty, they prefer their names not to be written individually in the expression of gratitude, in this brief space.

My sincere thanks to one and all, associated with this dissertation directly or indirectly at any point of time.

Tej Tarun Mahindra

M.E.(Computer Technology & Applications)

College Roll No. 25/CTA/04

Delhi University Roll No. 4005

ABSTRACT

Peer-to-peer systems and applications are distributed systems. P2P systems form the basis of several applications, such as file sharing systems and event notification services. Distributed Hash Table (DHT) based P2P systems such as CAN, Chord, Pastry and Tapestry, use uniform hash functions to ensure load balance in the participant nodes. But their evenly distributed behavior in the virtual space destroys the locality between participant nodes. The topology-based hierarchical overlay networks like Grapes and Jelly, exploit the physical distance information among the nodes to construct a two-layered hierarchy. This highly improves the locality property, but disturbs the concept of decentralization as the leaders in the top layer get accessed very frequently, becoming a performance bottleneck and resulting in a single point of failure. In this dissertation, a virtual id based m-way search tree (VIM) P2P overlay infrastructure, called VIM is proposed .It is shown through simulation that VIM can achieve both the decentralization and locality properties along with high fault tolerance and a logarithmic data lookup time.

Contents

List of Figures

Chapter 1

Introduction

1.1 Peer-to-Peer Overview

The peer-to-peer working group describes peer-to-peer computing to be the sharing of computer resources and services by direct exchange between systems. These resources and services include the exchange of information, processing cycles, cache storage, and disk storage for files. Thus, by taking advantage of the network infrastructure and the existing computing power, peer-to-peer networks [3][4] can harness the latent resources of a group of computers, thereby allowing for dissemination of knowledge and sharing of CPU cycles.

Before the introduction of the peer the peer technology, client server technology has been used in the computer world, in which few powerful machines (the servers) provided services to the less powerful machines (clients) that were usually desktop computers. It resulted in a waste of resources. With the introduction of powerful desktop machines, high bandwidth networks, and the ease of interconnection, Peer-to-peer computing makes possible the utilization of these resources, resulting in a more optimized computing community. Each computer can act as both a server and a client, invalidating the need for large mainframes and high end machines. Peer-to-peer computing replaces the asymmetric client/server relationship with a symmetric one in which any peer can request service of and provide service to another peer.

Peer to peer computing not fully eliminated the concept of client and servers, certain applications follow a peer-to-peer architecture (e.g. chatting) and application like Search engines, electronic marketplaces, and applications requiring real-time access to data (e.g. stock quotes) are better off with centralized client-server architecture.

However, the computing industry did not always have a client-server mindset. In fact, peer-to-peer is the oldest architecture in the communications world. The goal of ARPANET was to share computing resources around the U.S. The idea was to integrate different kinds of existing networks as well as future technologies with common network architecture that would allow every host to be an equal player. Although applications like Telnet and FTP were client-server, the network architecture allowed

any host to Telnet or FTP to any other host. The emergence of web browsers and modem connections shifted the peer-to-peer architecture to what it is today. Thus, in a way, peer-to-peer technologies will return the Internet to its original version, in which everyone creates as well as consumes.

Thus, peer-to-peer architecture allows for the separation of the concepts of authoring information and publishing that same information. It also allows for decentralized application design. As peer-to-peer applications become more widespread, the network architecture is going to have to change to better handle the new traffic patterns.

Definition

Because Peer-to-Peer systems are relatively young and still evolving, a precise definition is hard to establish.

What is common to most definitions is the idea that such systems have resource sharing at aim, they must have certain degree of autonomy and decentralization, the fact that dynamic IP addresses are usually involved, and last but not least, the client-and-server dual role of participants, e.g.:

According to Oram P2P is a class of applications that takes advantage of resources available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, P2P nodes must operate outside the DNS system and have significant or total autonomy from central servers.

According to Miller P2P is a network architecture in which each computer has equivalent capability and responsibility. P2P has five key characteristics. (i) The network facilitates real-time transmission of data or messages between the peers. (ii) Peers can function as both client and server. (iii) The primary content of the network is provided by the peers. (iv) The network gives control and autonomy to the peers. (v) The network accommodates peers that are not always connected and that might not have permanent Internet Protocol (IP) addresses.

P2P Working Group gave a clear definition of peer to peer system as P2P computing is the sharing of computer resources and services by direct exchange between systems. These resources and services include the exchange of information, processing cycles, cache storage, and disk storage for files. Peer-to-peer computing takes advantage of existing desktop computing power and networking connectivity,

allowing economical clients to leverage their collective power to benefit the entire enterprise.

1.2 Evolution

Peer-to-Peer systems have evolved during the last 6 years or so, since the introduction of Napster. The evolution of the peer to peer to peer system depended mostly on two features: decentralization, guarantee of success and scalability.

The Beginning

Napster offered its users a way to share files with the use of a centralized directory service, while the storage was decentralized. This centralization brought two difficulties. First, it was a problem that most of the material shared in the network had copyright. The directory server was storing and issuing information that ultimately led to what were considered illegal downloads. And second, the directory server is a single point of failure; moreover, the system is also difficult to scale, given that the load in the directory server increases with linear cost relative to the number of participants in the network.

Gnutella [1] came up with ideas involving flooding systems in networks where one participant only needed to know about another one peer to start proceedings and gain knowledge of other participants in the network. Similarly, a participant performs a flooding algorithm by asking all of his neighbors about a given query. His neighbors act similarly and the process is stopped by a query embedded Time-To-Live value that prevents further forwarding of queries, and thus, an ultimate collapse of the network due to increasing traffic. With this idea the centralization problem was overcome, but it still remained the issue of scalability.

Structure

With the issue of scalability in mind a new idea crawled into the researchers minds: to impose a logical structure to the network topology laying within. And thus the structured Peer-to-Peer systems were born. Their major representatives were Chord[5], CAN[6] , Pastry[7] and Tapestry[8].

The technology on top of which these projects are based is known as a Distributed

Hash Tables (DHT) [18]. A node (Peer) in such systems acquires an identifier based on a cryptographic hash [16] of some unique attribute such as its IP address. A key for a data item is also obtained through hashing. The hash table actually stores data items as values indexed by their corresponding keys. That is, node identifiers and key-value pairs are both hashed to one identifier space. The nodes are then connected to each other in a certain predefined topology, e.g. a circular space in Chord, a d-dimensional Cartesian space in CAN [6] and a mesh in Tapestry [8] and key- value pairs are stored at nodes according to the given structure. With the structured topology, data lookup becomes a routing process with low (typically logarithmic) routing table size and maximum path length. DHTs provide high data location guarantees because no restriction on the scope of search [9] [10] is imposed. .

Given the desirable properties of scalability and high guarantees while meeting the requirements of full decentralization, DHTs are currently considered as the most reasonable approach to routing and location in P2P systems. While having a common principle, each system has some relative advantages. e.g., The Chord system has the property of simple design. Tapestry and Pastry address the issue of proximity routing. The most attractive property in all current DHT systems is self-organization. Due to the focus on the absence of central authority, DHTs provide mechanisms by which the structural properties of the network are maintained while the peers are continuously joining and leaving it. Periodic stabilization is the system used by Chord, CAN and Pastry. It involves a number of routines being executed in a periodic fashion to correct the routing information that each node maintains.

Correction-on-change complements correction-on-use by proposing that each time a node joins leaves or drops from the network some new routing information has to be injected into a number of nodes that will propagate the information according to needs.

The combination of correction-on-change and correction-on-use does not have the high cost of bandwidth that periodic stabilization shows. If there are no changes in the network, no extra traffic is added. Furthermore, the use of this combination adds an extra robustness to the systems that use it that comes from the fact that when a node joins or fails other nodes are pro-actively notified.

1.3 Taxonomy

Basically considering two variables (decentralization and topology) the following taxonomy in Fig 1.1 is considered as suitable for Peer-to-Peer systems. This taxonomy captures major differences between P2P systems. The network structure characteristic aims at looking at systems from the topological perspective. Two levels of structuring are identified: unstructured and structured. In an unstructured topology, an overlay network is realized with a random connectivity graph. In a structured topology, the overlay network has a certain predetermined structure such as a ring or a mesh.

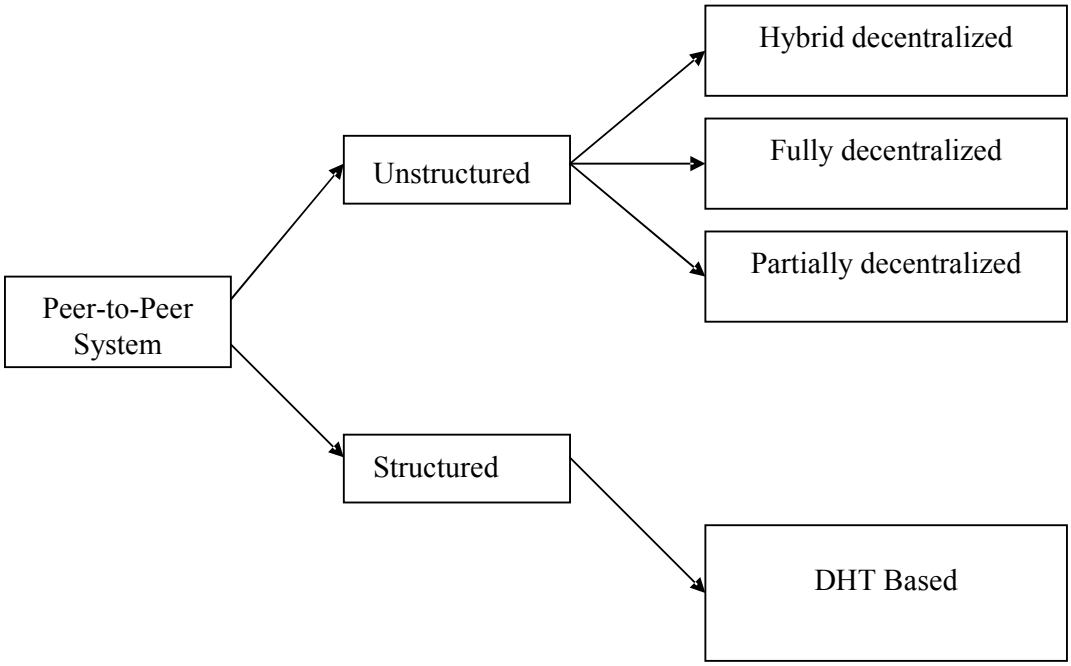


Fig 1.1 Taxonomy of Peer-to-Peer systems

The degree of centralization means to what extent the set of peers depends on one or more servers to facilitate the interaction between them. Three degrees are identified:

- *Fully decentralized*
- *Partially decentralized*
- *Hybrid decentralized*.

In the fully decentralized case, all peers are of equal functionality and none of them is important to the network more than any other peer.

In the partially decentralized case, a subset of nodes can play more important roles

than others, e.g. by maintaining more information about their neighbor peers and thus acting as bigger directories that can improve the performance of a search [9] process. This set of relatively more important peers can drastically vary in size while the system remains to be functioning.

In the Hybrid Decentralization case, the whole system depends on one or very few irreplaceable nodes which provide a special functionality in one aspect such as a directory service. However, all other nodes in the system, while depending on one special node, are of equal functionality and they autonomously offer services to one another in a different aspect such as storage. Thus, a system of that class is a hybrid system that is centralized in one aspect and decentralized in another aspect.

1.4 Scope of work

Peer-to-peer systems and applications are distributed systems. The core operation in most peer-to-peer systems is efficient location of data items. The current well-known peer-to- peer systems like Napster and Gnutella have scalability problem in location of data items. To solve the scalability problem, some scalable peer-to-peer lookup services show up, such as CAN, Chord, Pastry, and Tapestry. But their evenly distributed behavior in the virtual space destroys the locality between participant nodes. So, a self-organizing hierarchical virtual network infrastructure, called Grapes, for peer-to-peer lookup services is introduced. Hierarchical approach of Grapes brings two benefits. First, a node can find data in its sub-network with the high probability due to the data replication in its sub-network. Second, the hierarchical structure makes lookup hops shorter than those of the flat one. Although hierarchical overlay network like Grapes can highly improve the locality property of DHTs, but it disturbs the decentralization property. The leader has to route all the queries of its sub-network and has to manage super-network routing too, thus becoming a performance bottleneck. Also there is a possibility of a high degree of non-uniformity at lower levels; while some of the nodes may have to handle a great amount of network traffic others may not be used at all.

Here scheme is proposed - VIM that solves the problem of decentralization by distributing the network traffic between multiple hosts and also its increases the fault tolerance of the system as well as reduces the lookup path length with the use of virtual Id (for hosts and nodes), personal address book (PAB) and common address book (CAB).

1.5 Organization of Dissertation

The organization of this dissertation is as follows:-

Chapter 1 deals with the concept of peer-to-peer network, taxonomy for peer-to-peer system and scope of the dissertation.

Chapter 2 explains about the unstructured and structured peer-to-peer networks. This is chapter elaborates the Gnutella, Napster, CHORD and CAN.

Chapter 3 discusses the concepts of hierarchical peer to peer network like Grapes.

Chapter 4 explains the complete design of VIM Scheme.

Chapter 5 presents the performance analysis of the proposed Scheme VIM.

Chapter 6 presents the conclusion over the various aspects of the proposed design and the extension of the dissertation for future.

References

Appendix – Source Code

Chapter 2

Unstructured and Structured P2P Networks

2.1 Unstructured Peer to Peer Networks

2.1.1 Centralized Network – Napster

Napster's infrastructure was based around centralized index servers that maintained a database of all the content on the network and clients currently logged on at any time.

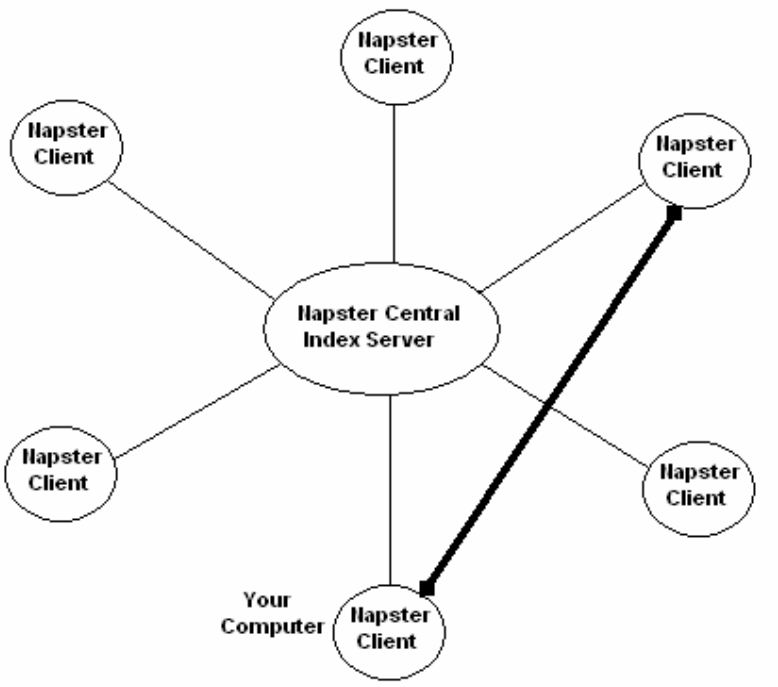


Fig 2.1 Napster & its Clients

In shown figure (Fig 2.1) few computers/peers are attached to a central computer and central computer maintains a database of the content that each computers are having.

Napster [2] is a **Hybrid P2P:**

- ✓ Has a central server that keeps information on peers and responds to requests for that information.

- ✓ Peers are responsible for hosting the information as the central server doesn't store files, for letting the central server know what files they want to share and for downloading its shareable resources to peers that request it.
- ✓ Route terminals are used addresses, which are referenced by a set of indices to obtain an absolute address

Requirements for using Napster

Each user must have Napster software in order to take part in file transfers. The user runs the Napster program. Once executed, this program checks for an Internet connection. If an Internet connection is detected, another connection between the user's computer and one of Napster's Central Servers will be established. This connection is made possible by the Napster file-sharing software. The Napster Central Server keeps a directory of all client computers connected to it and stores information on them as described above. If a user wants a certain file, they place a request to the Napster Centralized Server that it's connected to. The Napster Server looks up its directory to see if it has any matches for the user's request. The Server then sends the user a list of all that matches (if any) it as found including the corresponding, IP address, user name, file size, ping number, bit rate etc. The user chooses the file it wishes to download from the list of matches and tries to establish a direct connection with the computer upon which the desired file resides. It tries to make this connection by sending a message to the client computer indicating their own IP address and the file name they want to download from the client. If a connection is made, the client computer where the desired file resides is now considered the host. The host now transfers the file to the user. The host computer breaks the connection with the user computer when downloading is complete.

Disadvantages

1. This type of network cannot be scaled.
2. This type of network suffers from single point of failure problem or bottleneck problem

2.1.2 Scalable Network – Gnutella

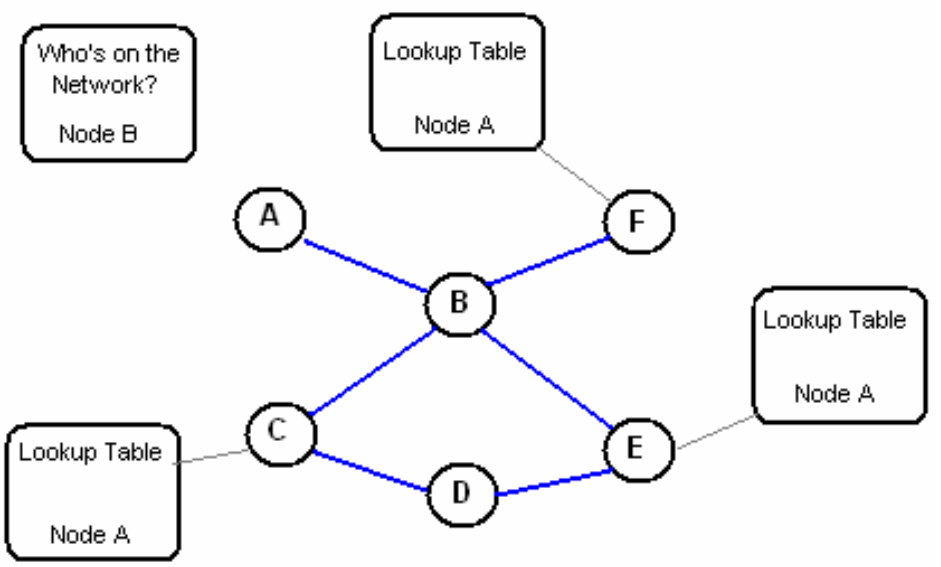


Fig 2.2 Gnutella: Scalable Network

Justin Frankel and Tom Pepper invented Gnutella [1] at Nullsoft, a subsidiary of AOL. The program was released on the 14th March 2000 to the general public from the Nullsoft website.

The Gnutella protocol (current version 0.4) is run over TCP/IP a connection-oriented network protocol. A typical session comprises a client connecting to a server. The client then sends a Gnutella packet advertising its presence. The servers through the network propagate this advertisement by recursively forwarding it to other connected servers. All servers that receive the packet reply with a similar packet about themselves.

Queries are propagated in the same manner; with positive responses being routed back the same path. When a resource is found and selected for downloading, a direct point-to-point connection is made between the client and the host of the resource, and the file downloaded directly using HTTP. The server in this case will act as a web server capable of responding to HTTP GET requests.

Gnutella packets are of the form:

MessageID(16 bytes)	**FunctionID**(1byte)	**TTL**(1byte)	**Hops**(1byte)	**Payload length** (4 bytes)

Where:

Message ID in conjunction with a given TCP/IP connection is used to uniquely identify a transaction.

Function ID is one of: Advertisement [response], Query [response] or Push-Request.

TTL is the time-to-live of the packet, i.e. how many more times the packet will be forwarded.

Hops count the number of times a given packet is forwarded.

Payload length is the length in bytes of the body of the packet.

Connecting

A client finds a server by trying to connect to any of a local list of known servers that are likely to be available. This list can be downloaded from the internet, or be compiled by the end user. The Advertisement packets (also known as Ping or Init) comprise the number of files the client is sharing, and the size in Kilobytes of the shared data. The server replies (Pongs) comprise the same information. Thus, once connected, a client knows how much data is available on the network.

Queries

Queries are propagated the same way as Advertisements. To save bandwidth, servers that cannot match the search parameters need not send a reply.

Downloading

A client wishing to make a download opens a HTTP (hyper-text transfer protocol) connection to the host and requests the resource by sending a "GET >URL<" type HTTP command, where the URL (Uniform Resource Locator) is returned by a Query request. Hence, a client sharing resources has to implement a basic HTTP server.

Firewalls

A client residing behind a firewall trying to connect to a Gnutella network will have to connect to a server running on a "firewall-friendly" port. Typically this will be port 80,

as this is the reserved port number for HTTP, which is generally considered secure and non-malicious.

When a machine hosting a resource cannot accept HTTP connections because it is behind a firewall, it is possible for the client to send a "Push-Request" packet to the host, instructing it to make an outbound connection to the client on a firewall-friendly port, and "upload" the requested resource, as opposed to the more usual client "download" method.

Limitations of the Protocol

The principal shortcomings in the protocol are:

- **Scalability:** The system had been designed in a laboratory and had been set up to run with a couple of hundred users. When it became available on the internet, it quickly grew to having a user base of tens of thousands. Unfortunately at that stage the system became overloaded and was unable to handle the amount of traffic and nodes that were present in the system.
- **Packet Life:** To find other users, a packet has to be sent out into the network. It became apparent early on that the packet life on some packets had not been set right and a build up of these packets started circulating around the network indefinitely. This resulted in less bandwidth being available on the network for users.
- **Connection Speeds of Users**: Users on the system act as gateways to other users to find the data they need. However, not every user had the same connection speed. This resulted in problems as users on slower bandwidth machines were acting as connections to people on higher bandwidth. This resulted in connection speeds being dictated by people with the slowest connection speed, on the link to the data thereby leading to bottlenecks.

Furthermore not the entire network is visible to any one client. Using the standard time-to-live during advertisement and search, only about 4,000 peers are reachable. This arises from the fact that each client only holds connections to 4 other clients and a search/init packet is only forwarded 5 times. In practical terms this means that even though a certain resource is available on the network, it may not be visible to the seeker because it is too many nodes away. To increase the number of reachable peers in the

Gnutella network the time-to-live for packets and the number of connections kept open us increased. But this creates problems that, if increase both the number of connections and the number of hops to eight, 1.2 gigabytes of aggregate data could be potentially crossing the network just to perform an 18 byte search query.

Another significant issue that has been identified is Gnutella's susceptibility to denial of service attacks. For example a burst of search requests can easily saturate all the available bandwidth in the attacker's neighborhood, as there is no easy way for peers to discriminate between malicious and genuine requests. All in all the overall quality of service of the Gnutella network is very poor.

2.2 Structured Peer to Peer Networks

2.2.1 The Chord Protocol

2.2.1.1 Introductory concepts

For the understanding of Chord's behavior hash functions and modular arithmetic are two important concepts. Chord [5] uses SHA-1 as hash function [16].

Hash functions

Each node belonging to the network is assigned a number through the use of a hash function. Each item that is going to be made available (searchable, or retrievable) has such a numeric association too.

Hash functions usually convert an input from a (typically) large domain into an output in a (typically) smaller range.

The domain can be any number, or any data that can be represented in a numeric way. In the case of the IDs of nodes belonging to a Chord [5] network, the IP address, or the <IP,port> pair serve as a value from the domain in the hash function. In the case of items or resources to be shared in the network, the name of a file or resource, or even their contents can also be represented in a numeric way, making its hashing possible.

Hashing is used resides in the fact that these functions randomize and disperse values.

- **Randomization**: given a value **X** from the domain, **hash(X)** will be a value from the range of the function with a certain degree of randomness. This only means that small values of **X** will not necessarily mean small (nor specifically big either) values of **hash(X)**.
- . **Dispersion**: given two similar or close values of the domain, **X** and **Y**, there is high probability that **hash(X)** and **hash(Y)** will be distant one from each other.

Hence, two nodes with similar <IP, port> values (belonging to a certain LAN/WAN, other factors like geographical proximity, or simply resembling values) will end up having very different numeric values after being applied a hash function. The same holds for resources with similar contents previous to hashing.

Modular arithmetic

Chord is a protocol whose behavior is based entirely on the topology that the network forms. Whole topology lies on the Modular arithmetic.

The numeric representation of both nodes and items will belong to a certain range of numbers [0,X). This numbers will be operated in a modulo arithmetic, which means, in "modulo p": 0+1=1, 1+1=2, (p-1)+1=p=0, p+1=1, and so on; Fig 2.3 shows an example, in "modulo 3":

0 (modulo 3) = 0
1 (modulo 3) = 1
2 (modulo 3) = 2
3 (modulo 3) = 0
4 (modulo 3) = 1
5 (modulo 3) = 2
6 (modulo 3) = 0
7 (modulo 3) = 1
etc...

Fig 2.3 Example of numeric equivalences in “modulo 3”

Network topology

Chord’s behavior is defined in terms of the way that nodes organize themselves,

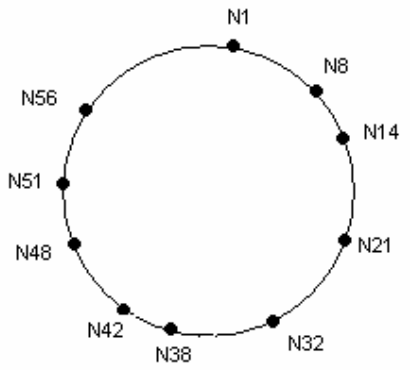
the so-called *topology of the network*. Topology is described in two stages.

a) Basic layout

The two main actors in Chord are nodes and items. Nodes belonging to the Chord network will be referred to as *node* or its identifier *id*, and shared items as *documents* or *keys*. Any of those is a number belonging to the range **[0,2^m)**.Each node in the network will be responsible for a set of *keys*. Chord node is not automatically responsible for the keys it wants to share in the network. When a node shares an item, this item's key will be inserted in the network and will be assigned as a responsibility to (probably) another node.

Nodes and documents organize themselves with respect to each other: identifiers are ordered in a **modulo 2^m** ring. Key *k* is assigned to the first node whose identifier is equal to or follows (the identifier of) *k* in the identifier space, regardless of which node the owner of the file (or resource) was originally that, generated this key. This node is called the *successor* node of key *k*, denoted by *successor(k)*. If identifiers are represented as a circle of numbers **from 0 to 2^m-1**, then *successor(k)* is the first node whose assigned identifier is k or, in the absence of this, the first node found clockwise from *k* in the ring. This circle of identifiers is known as the *Chord ring*.

Fig 2.4(a) below illustrates a Chord ring with 10 nodes. Fig 2.4(b) shows node 14 requesting the insertion of document 24. When inserted, document 24 becomes responsibility of node 32, which is the present successor of key 24, as shown in Fig 2.4(c).



a) Chord ring with 10 nodes

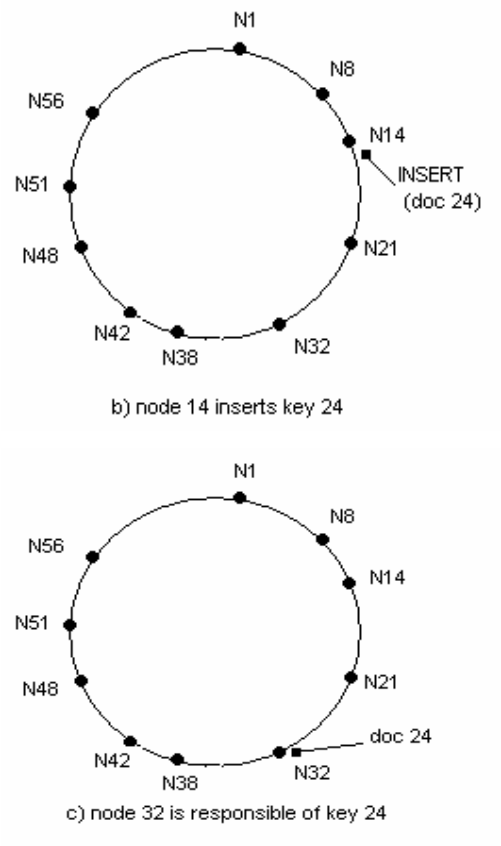
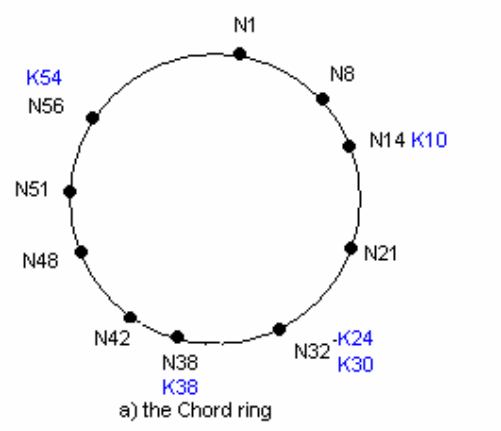


Fig 2.4 Assignment of responsibilities

The basic topology is that every *node* knows its successor, forming the Chord ring. For example , assume Chord ring with **m=6**. Every *identifier* (or *key*) in the network would be a number **0<=X<=2^6**, so **0<=X<64**, or **X€[0,64)**. In this, Chord ring could accommodate a maximum of **N=2^m=64 *nodes***, each one of them with an identifier **X€[0,64)**. Each one of those nodes would be responsible for one *key* at maximum, the *key* being equal to its node i*dentifier*, according to what was illustrated in Fig 2.4. This example has 10 *nodes*, with identifiers: 1, 8, 14, 21, 32, 38, 42, 48, 51 and 56. Some of the *nodes* are responsible for a set of *keys* present in the network, *keys* (or *documents*) 10, 24, 30, 38 and 54 are in the network, available for any peer to be retrieved. Its responsible node holds each one of those documents. A *node* with identifier *id* is responsible for *document d* if *id=successor(d)*. This provides lookup search capabilities with linear cost (the average number of hops necessary to locate a *key* would be **O(N)**). Fig 2.5(b) shows the pseudo code for a lookup operation in RPC format, and Fig 2.5(c) shows a graphical description of its behavior when *node* 8 requests *document* 54, on the

ring previously described and showed in figure Fig 2.5(a)




```
//Node n asks to find successor of id
n.findSuccessor(id){
if (id E(n,successor])
return successor;
else
//forward the query around the circle
return successor.findSuccessor(id);
}
```

b)pseudocode for lookup

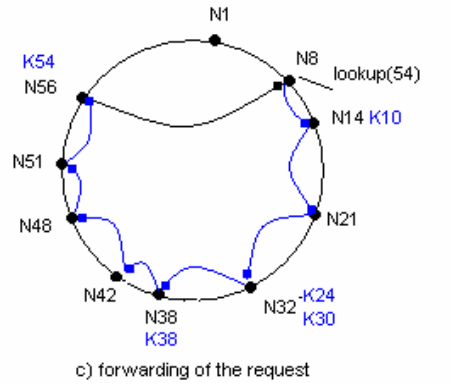


Fig 2.5 Example of lookup in its simplest form: linear forwarding around the ring

b) Further data structures

In order to achieve the goals of the protocol (improved efficiency, performance and scalability, fault tolerance, etc), further data structures are required.

Each node has the following data regarding other nodes in the network:

- **A fingers table with *m* entries**. *m* refers to the number of bits that limit the identifier space. Each given entry *i* in the *finger* table holds, *0<=i<m* the

identifier of *successor(id+2^i)*.This structure offers lookup performance improvement. *fingers*[0] holds *successor* of (id+1), which means the *successor*, the next node found clockwise in the Chord ring; so the variable *successor* is not needed anymore, as its equivalent is now part of the *finger table*. This structure is the one that will ensure that lookups will be performed with cost **O($\log_2$N)**, given that the Chord ring has identifiers belonging to **[0,2^m)**, and the size of the network is at most **N = 2^m**. Fig 2.6 shows a couple of examples of the finger tables of nodes 14 and 38, for the network shown in Fig 2.4. Given that this network had an identifier space limited by **m=6**, the finger tables have 6 entries:

Finger Level	**Aim Node ID + 2 level**			**Successor of aim**
0	14+2^0	14+1	15	21
1	14+2^1	14+2	16	21
2	14+2^2	14+4	18	21
3	14+2^3	14+8	22	32
4	14+2^4	14+16	30	32
5	14+2^5	14+32	48	48
Finger Table of Node 14				

Finger Level	**Aim Node ID + 2 level**			**Successor of aim**
0	38+2^0	38+1	39	42
1	38+2^1	38+2	40	42
2	38+2^2	38+4	42	42
3	38+2^3	38+8	46	48
4	38+2^4	38+16	54	56
5	38+2^5	38+32	68	8
Finger Table of Node 38				

Note that Successor of 68 is 8
68 modulo 26 = 4 , the successor of 4 is 8

Fig 2.6 Finger tables for nodes 14 and 38

- . **The predecessor**. *p=predecessor(n)* means that *n=successor(p)*. Identifiers are represented in a circle of numbers **from 0 to 2^m-1**, the *predecessor* of *n* is the first node found counter-clockwise from *n* in the Chord ring. This is necessary for internal management of the topology as the network changes (nodes joining and leaving).
- **A successors list**. This is a list of the next nodes found clockwise. The longer this list is, the more tolerant to simultaneous failures of nodes is the

network. The successors list will be named “sList”.

- . **A referrers list**. This is a list of the nodes that are pointing to the node from any of their *fingers*. They are useful in the event of a node leaving the network. When a node will leave, it will let all the *referrers* know, so each *referrer* will be able to substitute the *finger* for an appropriate node (which is always the *successor* of the leaving node).

2.2.1.2 Operations in Chord

This section contains information about significant parts of the code that the protocol uses to achieve its goals. Certain routines are called periodically.

a) Join

When a node joins the network, its successor and predecessor are set to none (*null*).

The first thing a node **X** does when being inserted is to request to any node **Y** present in the network who is **X**’s *successor*. When **X** receives a reply, it stores its *successor*’s id.

The *stabilization* and *fixFingers* routines are called for the first time, and will be executed periodically. This will ensure that the *predecessor*, the *fingers,* as well as the *successors list* (sList) and the *referrers list* stay up to date. Given that consistent hashing provides the network with the ability to let nodes enter and leave it with minimal disruption, when a node n enters the network certain keys previously assigned as a responsibility to *n*’s successor now should become assigned to *n*, e.g.: if node 32 is responsible for keys 15, 18 and 30, and now node 20 joins the network, it follows that keys 15 and 18 will be now responsibility of the recently joined node.

b) Lookup

The lookup operation is the heart and core of the protocol. Its performance and reliability stem from the data structures that a node holds and maintains.

Let us assume a node with identifier *n* is interested in locating key *id*: if *id* lies between *n* and *n*’s successor in the identifier circle, the result of the operation is *n*’s successor.

Otherwise, the lookup request is forwarded to the closest preceding node in the network that *n* knows about by inspecting the *fingers table*. This way, by forwarding request, the lookup operation will make steps closer and closer clockwise in the

identifier circle to reach its destination. At each forwarding step, the forwarder node goes as far away in the identifier circle as its data allows to, and that fact will ultimately justify the **O (log_2N)** cost.

When the *successors list* structure is used, it does not only provide robustness in the event of node failures, but also gives a slight performance improvement. When looking for the closest preceding node to forward a lookup request, this structure can be inspected too in order to save some of the last forwarding hops.

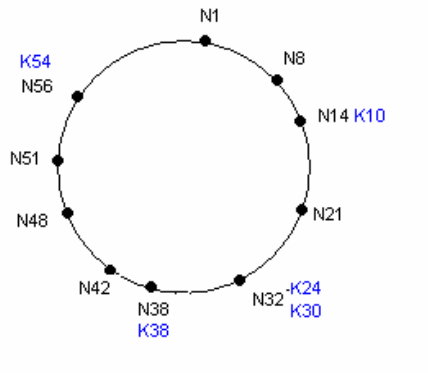


Fig 2.7 Example of a lookup request: node 8 asks for key 54

c) Stabilization

The network is kept stable (or converges to a stable network) by means of two operations: *stabilize* and *fixFingers*.

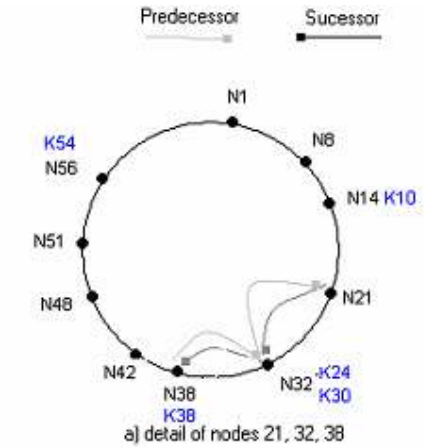
- **The *stabilize* operation:** It ensures that *successor* and *predecessor* pointers are kept up to date. The successor is updated when a new node has been inserted in the identifier circle between the node running the stabilization routine and its successor —this is done by asking for the *successor*'s *predecessor*. Next thing the routine does is requesting the *successors list* of its *successor*, and then build its own by removing the last item and prefixing the successor as first item. Then, the routine lets its *successor* know about its existence, by calling the *notify* procedure. When a node receives a *notify* call, it checks from which node it comes, and updates the *predecessor* pointer if necessary after having checked out that the node who claims to be the predecessor is a better candidate than the existing *predecessor*. The last thing done in the *stabilize* operation is to re-schedule itself to guarantee a periodical execution of the call.

- . **The *fixFingers* operation:** It updates one entry of the *finger* table at a time, scheduling the next update for a later round of *finger table* correction, which happens periodically. It basically consists of making a call to find_successor, and looking for the best fit existing node in the network for the position of the finger table that is being corrected. If the reply to the find_successor call is the same as the content of the fingers table, no correction is needed; when corrections are made the referrers list of both the old finger and the new finger are updated consequently.

d) Failure

Chord's desired strengths is robustness, the nodes need know about other nodes disappearing from the network which is denoted as a node failure , regardless of whether this absence is voluntary or not. The way that nodes know about this is with **negative acknowledgement**. A node sends periodically a message to its *predecessor* to see if it is alive. If the sending node does not receive a reply within a certain timeout, it means that the receiver is not in the network anymore.

The worst case is when a node just drops from the network or disappears without making other nodes in the network notice. In order to make it possible for the network to maintain its invariants of robustness and performance, a node checks periodically for the presence of its *predecessor*. If the predecessor is not present anymore, the predecessor pointer is set to none (*null*), and the *stabilization* routine will make the rest of the job, because nodes send a *notify* call to their *successors* periodically. This is when the *successors list* comes in useful. Fig 2.8 illustrates the example using the same ring, and focusing on nodes 21, 32 and 38.



a) detail of nodes 21, 32, 38

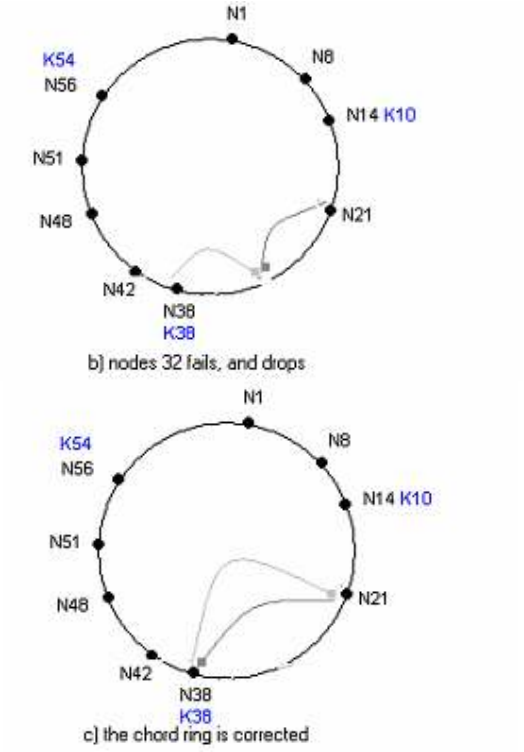


Fig 2.8 Network reorganization: node 32 drops nodes 21 and 38 are corrected

If node 32 fails and drops from the network (Fig 2.8(b)), the next time that 38 checks its predecessor it will realize that 32 is no more in the network. 38 will set its predecessor to null. And also, next time that 21 runs the stabilize routine, it will ask 32 about its predecessor, and 32 will not reply; hence, 21 will understand that 32 is not in the network anymore. Being it so, node 21 will remove 32 from the successors list and the finger table (remember that the first entry of the finger table is the successor). Instead, the next successor it knows about will be used. Say, for example, that every node in the network has a successors list of size 3. This means that each node knows about the 3 next nodes found clockwise counting from their own identifier. Thus, node 21 had this successors list before 32's failure: {32, 38, 42 }. As node 32 has disappeared, the next successor that 21 know about is 38. 32 is then removed from the finger table and the successors list. Now, stabilize will be called again, and 21 will contact 38, asking about its predecessor. 38 will reply that its predecessor is null now, because its former predecessor has failed. This results in 21 not changing its successor (it has already been updated to 38 when 21 noticed that 32 failed). Next thing 21 does is notifying 38, claiming that it may be a proper predecessor for node 38. When 38 receives a notify from 21 it checks its

predecessor; it being null now, node 38 will take 21 as its new predecessor (Fig 2.8 (c)).

e) Leave

A node voluntarily leaving the network can be treated as a node failure, without need to warn other nodes about it. However, performance can be improved through slight additions.

- A node leaving the network tells its *successor* about it. The *successor* takes advantage of knowing who its *predecessor* will be from that moment on. Also, the node can send to its *successor* the set of resources of which it was responsible upon departure, and that will be assigned to the *successor*. This means that the *successor* needs not wait for the *stabilization* routine in order to fix the *predecessor* pointer, and also increases the positive responsiveness of nodes when being asked about keys (documents) that were present in the network and that might have disappeared if the departing node had not passed them on.
- A node leaving the network tells its *predecessor* about it. The node sends along its *successors list*, and the *predecessor* will use it from then on. This implies that the *predecessor* knows who its *successor* is at once, and does not need to wait for the *stabilization* routine to fix it.
- Every node **X** knows which other nodes in the network (**a,b,c**...) are referring to it. When node **X** leaves, it sends a message to each one of the nodes referring to it in the finger tables (**a,b,c**...) so that these nodes can substitute the reference to **X** for a better one. The substitute of **X** in nodes **a,b,c**... will be *successor(**X**)*, which is a value that **X** will send to these nodes in the same message that lets them know that **X** is leaving.

f) Insert

Any node belonging to the Chord network can share a new resource and make it available. The way the protocol works is, when a node *n* inserts a key *k*, it is responsibility of the node with *id = successor(k)* to maintain *k*, until departure.

2.2.2 A Content-Addressable Network

CANs resemble a hash table; the basic operations performed on a CAN [6] are the insertion, lookup and deletion of (key, value) pairs. Each CAN node stores a chunk (called a zone) of the entire hash table.

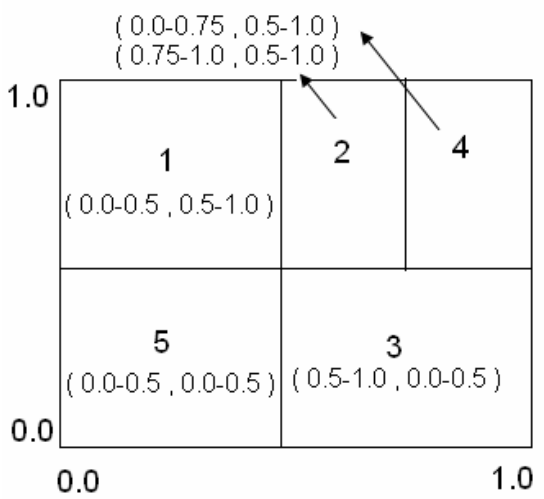


Fig 2.9 Example 2-d coordinate overlay with 5 nodes

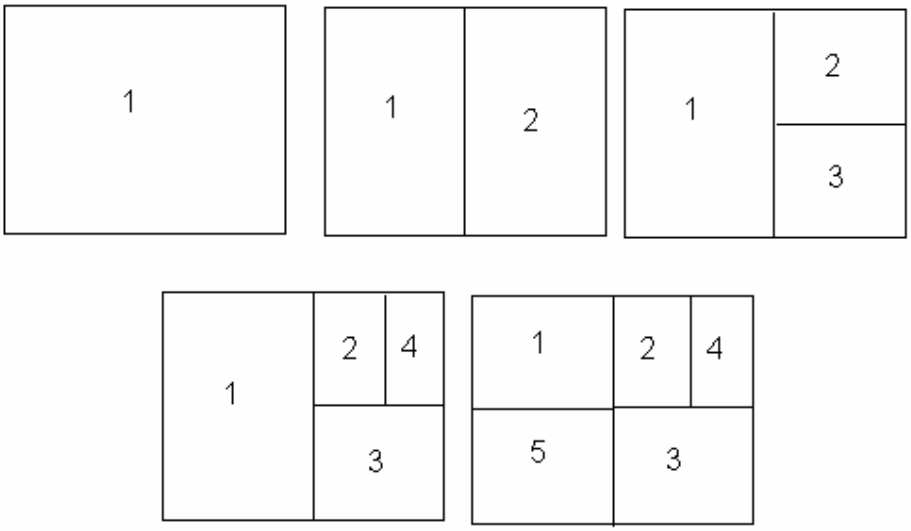


Fig 2.10 Partitioning of CAN space as 5 nodes are joining in Succession

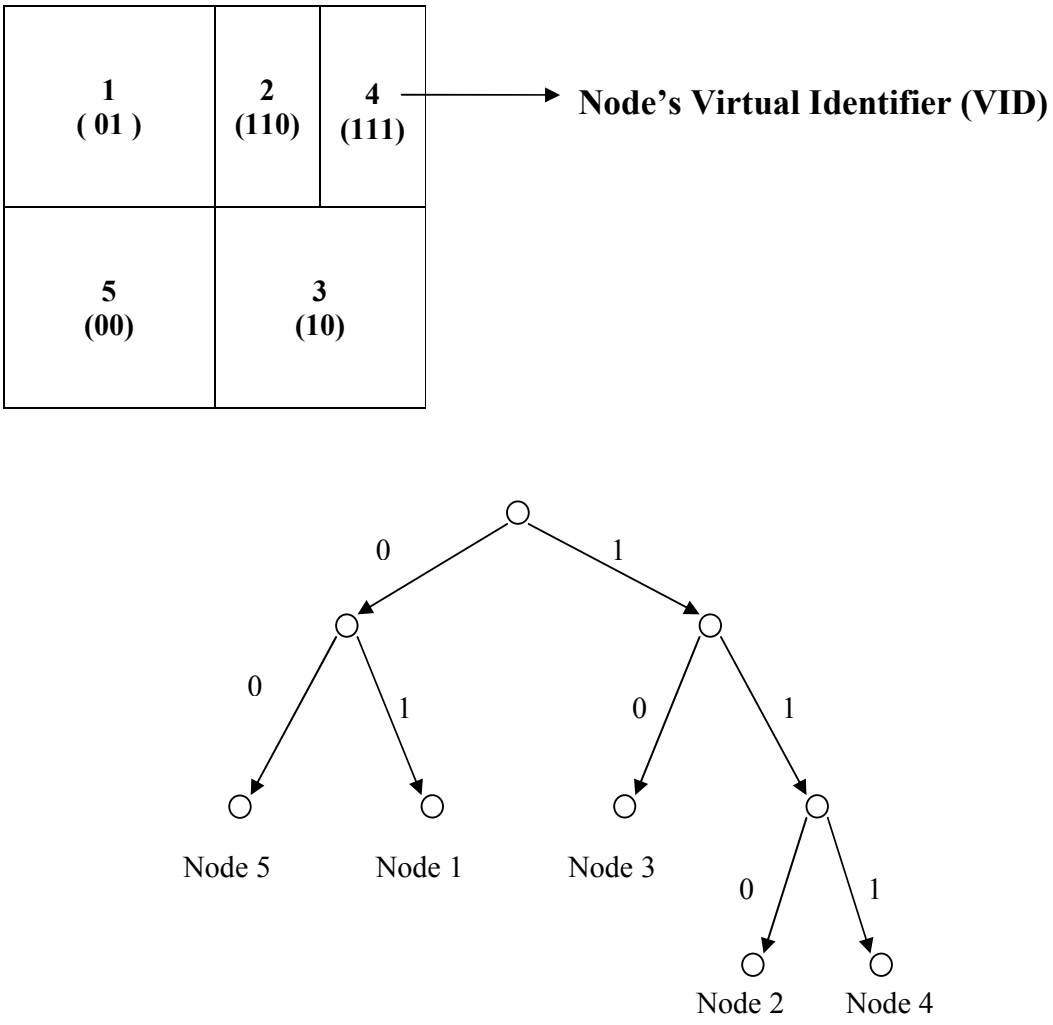


Fig 2.11 5 node CAN and its corresponding partition tree

Providing this hash-table-like interface then requires every node to support a single operation given an input key, a node must be able to route messages to the node holding key. Design primarily addresses the issues related to supporting name-based routing operation in a manner that is completely distributed (requiring no form of centralized control, coordination or configuration), scalable (nodes maintain only a small amount of control state that is independent of the number of nodes in the system), and robust to node and network failures.

2.2.2.1 Design

This design centers around a virtual d-dimensional Cartesian coordinate space on a d-torus. This coordinate space is completely logical and bears no relation to any physical coordinate system. At any point in time, the entire coordinate space is dynamically partitioned among all the nodes in the system such that every node owns its individual, distinct zone within the overall space. For example, Fig 2.9 shows the 2-dimensional [0 1]*[0 1] coordinate space with 5 nodes. Nodes in the CAN [6] self-organize into an overlay network that represents this virtual coordinate space. A node learns and maintains as its set of neighbors the IP addresses of those nodes that hold coordinate zones adjoining its own zone. This set of immediate neighbors serves as a coordinate routing table that enables routing between arbitrary points in the coordinate space.

This Cartesian space serves as a level of indirection. CAN uses a virtual coordinate space to store (key, value) pairs. To store a pair (K, V), key K is deterministically mapped onto a point P in the coordinate space using a uniform hash function. The corresponding key-value pair is then stored at the node that owns the zone within which the point P lies. To retrieve an entry corresponding to key K, any node can apply the same deterministic hash function to map K onto point P and then retrieve the corresponding value from the point P. If the point P is not owned by the requesting node or its immediate neighbors, the request must be routed through the CAN infrastructure until it reaches the node in whose zone P lies. Efficient routing is therefore a critical aspect of CAN.

In the following sections, we describe the three core pieces of CAN design: incorporating new nodes into the CAN, CAN routing, and adjusting to the departure of nodes from the CAN overlay.

2.2.2.2 Node Arrivals

The entire CAN space is divided among the nodes currently in the system. To obtain such a partitioning, each time a new node joins the CAN, an existing zone is split into two halves, one of which is assigned to the new node. The split is done by following a well-known ordering of the dimensions in deciding along which dimension a zone is to be split, so that zones can be re-merged when nodes leave. For example, for a 2-d space, a zone would first be split along the X dimension, then the Y, and then X again followed by Y and so forth. Fig 2.10 depicts the evolution of a 2-d CAN [6] space as 5

nodes join in succession. The first node to join owns the entire CAN space; i.e., its zone is the complete virtual space. When the second node joins, the space is split in two and each node gets one half. The third node to arrive picks one zone and splits it in half, and this process repeats as new nodes arrive. We can thus think of each existing zone as a leaf of a binary partition tree. The internal vertices in the tree represent zones that no longer exist, but were split at some previous time. The children of a tree vertex are the two zones into which it was split. The edges in the partition tree are labeled as follows: an edge connecting a parent and child zone is labeled 0 if the child zone occupies the lower half of the dimension along which the parent zone was split, otherwise (i.e., if the child zone occupies the upper half of the dimension along which the split occurred) the edge is labeled with a 1. Fig 2.11 represents a 5 node CAN and its corresponding labeled partition tree. A zone's position (i.e., the zone's coordinate span along each dimension) in the coordinate space is completely defined by the path from the root of the partition tree to the leaf node corresponding to that zone. Consider for example, the path from the root node to the leaf node 2 in Fig 2.11: the first edge label tells us that the 2's zone lies in the range [0.5,1.0] along the X axis, the second edge indicates that 2's zone lies in the range [0.5,1.0] along the Y axis and the third edge indicates that 2's zone lies in [0.5, 0.75] along the X axis (determined as the lower half of its previously determined span of [0.5, 1.0]). Every node in the CAN is addressed with a virtual identifier (VID) - the binary string representing the path from the root in the partition tree to the leaf node corresponding to the node's zone. Thus a node's VID compactly represents its position in the CAN Cartesian space.

To allow the CAN to grow incrementally, a new node that joins the system must
(a) be allocated its own portion of the coordinate space (i.e., obtain a unique VID) and
(b) discover its neighbors in the space (i.e., discover its neighbors' VIDs and IP addresses).

Briefly, an existing node splitting its allocated zone in half, retaining half and handing the other half to the new node, does this. The process takes three steps:

1. First the new node must find a node already in the CAN.
2. Next, using the CAN routing mechanisms, it must find a node whose zone will be split.

3. Finally, the neighbors of the split zone must be notified so that routing can include the new node.

Bootstrap

A new CAN node first discovers the IP address of any node currently in the system.
To join a CAN, a new node looks up the CAN domain name in DNS to retrieve a bootstrap node's IP address. The bootstrap node then supplies the IP addresses of several randomly chosen nodes currently in the system.

Finding a Zone

The new node then randomly chooses a point P in the space and sends a JOIN request destined for point P. This message is sent into the CAN via any existing CAN node. Each CAN node then uses the CAN routing mechanism to forward the message, until it reaches the node in whose zone P lies.
On receiving the JOIN message, the owner of this zone could directly split its own zone with the new node. However, the owner node knows not only its own zone coordinates, but also those of its neighbors. Therefore, instead of directly splitting its own zone [11], the existing occupant node first compares the volume of its zone with those of its immediate neighbors in the coordinate space. The zone that is split to accommodate the new node is then the one with the largest volume. The effect of this 1-hop volume check is to achieve a more uniform partitioning of the space over all nodes.
The selected occupant node then splits its zone in half; the occupant retains the half occupying the lower end of the dimension along which the zone is split and assigns the

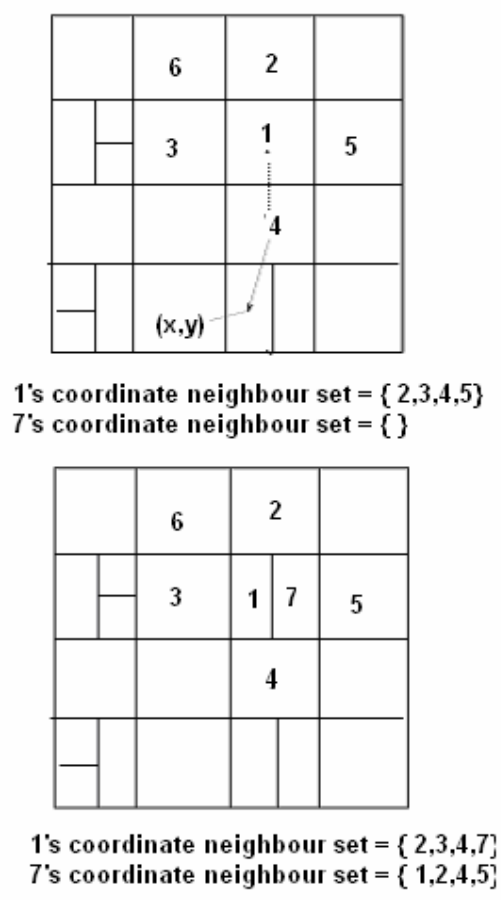


Fig 2.12 Example 2-d space before and after node 7 joins

other (higher-end) half to the new node. The occupant node then appends a "0" to its original VID to reject this shrinking of its zone; the new node acquires its VID by simply appending a “1" to the occupant's original VID. Finally, the (key, value) pairs from the half zone to be handed over are also transferred from the occupant node to the new node.

Joining the Routing

Having obtained its zone, the new node must learns the IP addresses of its coordinate neighbor set. In a dimensional coordinate space, two nodes are neighbors if their coordinate spans overlap along d-1 dimensions and abut along one dimension. For example, in Fig 2.12 , node 6 is a neighbor of node 3 because its coordinate zone overlaps with 6's along the X axis and abuts along the Y-axis. On the other hand, node 2 is not a neighbor of 3 because their coordinate zones abut along both the X and Y axes.

Splitting the previous occupant's zone derives a new node's zone; consequently, the new node's neighbor set is a subset of the previous occupant's neighbors, plus that occupant itself. Similarly, the previous occupant updates its neighbor set to eliminate those nodes that are no longer neighbors. Finally, both the new and old nodes neighbors must be informed of this reallocation of space. Every node in the system sends an immediate update message, followed by periodic refreshes, with its currently assigned zone to all its neighbors. These soft-state style updates ensure that all of their neighbors will quickly learn about the change and will update their own neighbor sets accordingly.

Fig 2.12 shows an example of a new node (node 7) joining a 2-dimensional CAN. As can be inferred, the addition of a new node affects only a small number of existing nodes in a very small locality of the coordinate space. The number of neighbors a node maintains depends only on the dimensionality of the coordinate space and is independent of the total number of nodes in the system. Thus, for a d-dimensional space, node insertion affects only O(d) existing nodes which is important for CANs with huge numbers of nodes.

2.2.2.3 Routing

Routing in a Content Addressable Network works by following the straight line path through the Cartesian space from source to destination coordinates. A CAN node maintains a coordinate routing table that holds the IP address and VIDs of each of its neighbors in the coordinate space. This purely local neighbor state is sufficient to route between two arbitrary points in the space. A CAN message includes the destination coordinates. Using its neighbor coordinate set, a node routes a message towards its destination by simple greedy forwarding to the neighbor with coordinates closest to the destination coordinates.

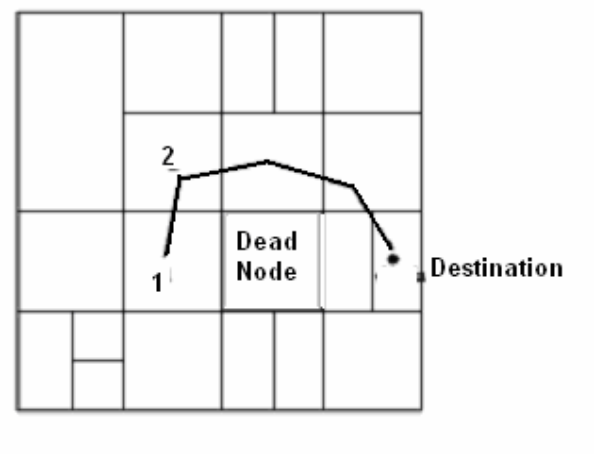


Fig 2.13 1-hop route check allows

Many different paths exist between two points in the Cartesian space and so, even if one or more of a node's neighbors were to crash, a node would automatically route along the next best available path. If however, a node loses all its neighbors in a certain direction, and the repair mechanisms have not yet rebuilt the void in the coordinate space, then greedy forwarding may temporarily fail. In this case, the forwarding node first checks with its neighbors to see whether any of them can make progress towards the destination and if so, greedy routing is resumed through a node two hops away from the current forwarding node. As the example in Fig 2.13 shows, this 1-hop route check is useful in circumventing certain voids, particularly at lower dimensions when a node has fewer options in finding neighbors that make progress to a destination. If, despite the 1-hop route check, greedy routing fails then the message is forwarded using the rules used to route recovery messages (described in the next section) until it reaches a node from which greedy forwarding can resume.

2.2.2.4 Node Departures

When nodes leave a CAN, we need to ensure that the zones they occupied are taken over by the remaining nodes. The normal procedure for doing this is for a node to explicitly hand over its zone state (i.e., its own VID and its list of neighbor VIDs and IP addresses) and the associated (key, value) database to a specific node called the takeover node. If the takeover's zone can be merged with the departing node's zone to produce a valid single zone, then this is done. If not, then the takeover node can temporarily handle both zones.

The CAN also needs to be robust to node or network failures, where one or more nodes simply become unreachable. This is handled through a recovery algorithm that ensures that the takeover node and the failed node's neighbors independently work to construct the routing structure at the failed node's zone. However in this case

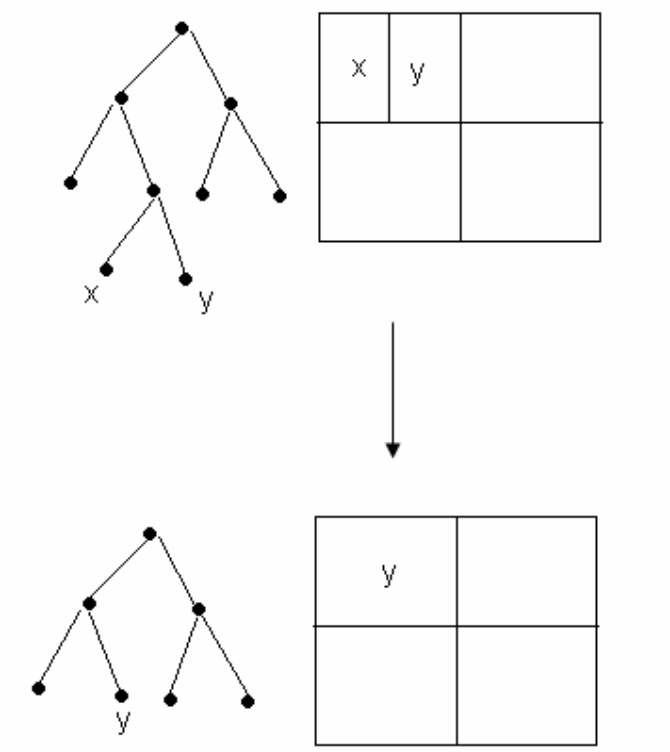


Fig 2.14 CAN before and after departure of node x Case #1

the (key, value) pairs held by the departing node is lost and needs to be rebuilt. This can be achieved in a number of ways. For example, the holders of the data can refresh the state. Alternately, each (key, value) pair might be replicated at multiple points in the CAN and a lost (key, value) pair can be rebuilt from its replicas. The appropriate solution to reconstructing the (key, value) database is largely dependent on application-level issues such as data consistency and availability requirements. CAN recovery comprises two key pieces: the identification of a unique node, called the takeover node, that occupies the departed node's zone and the process by which the departed node's neighbors discover the takeover node and vice versa.

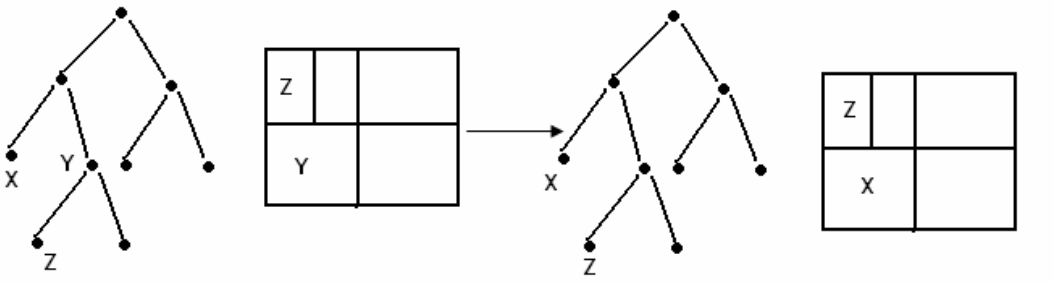


Fig 2.15 CAN before and after departure of node x Case #1

Identification of Takeover Nodes

The unique, well-defined node that takes over for a given departed node is called the departed node's takeover. Conceptually, a given node's takeover can be easily defined using the partition tree. Recall that in a partition tree, the internal vertices represent zones that no longer exist but were split at some time while the children of a tree vertex are the two zones into which it was split.

Now suppose a leaf vertex x leaves the CAN. If the sibling of this leaf is also a leaf (call it y) the departure is easy: y is the takeover node for x and we simply coalesce leaves x and y, making their former parent vertex a leaf, and assign node y to that leaf. Thus zones x and y merge into a single zone which is owned by node y.

If x's sibling y is not a leaf (because the sibling zone has been further split), perform a depth-first in the sub tree rooted at y until a leaf node is found. This leaf, call it z, acts as x's sibling and takes over for x. The zones of x and z cannot be simply merged into a single zone and hence z temporarily owns two distinct zones. Node z retains both zones until contacted by a new node at which point, it simply hands off one zone to the new node (rather than split one of its zones).

The above description uses the partition tree to identify a departed node's takeover. However, nodes do not explicitly maintain the partition tree structure; instead each node maintains only its own VID that summarizes its location in the tree. Nonetheless, as the following description reveals, a node's VID is enough information to identify its takeover.

As stated earlier, a node's VID is a binary string that denotes the path from the root to the node in the partition tree. Thus a VID can be regarded as the prefix of a k-bit string where k is selected to be greater than the depth of any partition tree expected in

practice. A VID of length thus specifies the first l of the k bits; the remaining k-l bits are simply set to zero. Regarded in this manner, every VID has an associated numerical value and examination of the partition tree reveals that for a given node, its takeover (as defined earlier) is simply the node with VID numerically closest to its own VID.
CAN uses two different notions of distance; the first is the Cartesian distance between two points in the CAN coordinate space and the second is the absolute value of the difference between two VIDs as defined above.
To summarize, when a node departs the CAN, its zone is taken over by the node with VID numerically closest to the departed node's VID. We now describe the actual process by which the departed node's neighbors and the takeover node discover each other.

Recovery algorithm

The preceding discussion defined the takeover node that occupies a departed node's zone but did not specify how the departed node's neighbors discover the takeover node and vice versa. We now address this issue of restoring neighbor links - we describe a distributed recovery algorithm by which each of the departed node's neighbors independently discovers the takeover node.

Under normal conditions a node sends periodic update messages to each of its neighbors. The prolonged absence of an update message from a neighbor signals its failure. When a node detects a failed neighbor, it deletes the dead node from its neighbor set and attempts to contact the dead neighbor's takeover. It does so by forwarding a recovery message to its neighbor closest to the dead node in terms of VID-distance. In this manner, the recovery message is routed incrementally closer to the dead node's VID. Ideally, this recovery message arrives at the takeover node from where it cannot be forwarded any closer to the dead node's VID. The takeover node thus infers that it is to occupy the dead node's zone and that the source of the recovery message is a neighbor of its newly acquired zone.
Since each of the dead node's neighbors will independently initiate such a recovery message, the takeover node discovers all its new neighbors and vice versa. In Chord, nodes are assigned unique binary identifiers and each node maintains a successor pointer to the first node with identifier greater than its own, effectively maintaining an

ordered linked list of all the nodes in the system. This linked list, provided it could be maintained in the face of node dynamics, guarantees connectivity between any two nodes. Inspired by Chord's idea of maintaining an ordered link list of nodes, CAN augment algorithms to require every node to know their immediate successor and predecessor in the numerical ordering of VIDs. Provided this chain of VIDs is maintained, a recovery message is guaranteed to arrive at the appropriate takeover node because every node can always make progress (in terms of VID-distance) to an arbitrary VID. In addition to its regular neighbor set, every node now maintains links to its immediate successor and predecessor in the VID space and uses Chord's stabilization algorithm to actively maintain this ordered link list of nodes. When a node dies its zone is taken over by the node closest (in terms of VID-distance) to the dead node. The dead node's neighbors discover this takeover node by greedy routing towards the dead node's VID.

Chapter 3

Hierarchical Peer to Peer Networks

3.1 Introduction to Super Layer and Sub Layer

Inspired by hierarchical routing in the Internet, two-tier exist in DHTs, in which

- Peers are organized in disjoint groups, and
- Lookup messages are first routed to the destination group using an inter group overlay, and then routed to the destination peer using an intra-group overlay.

Hierarchical DHTs have a number of advantages, including**:**

- They significantly reduce the average number of peer hops in a lookup, particularly when nodes have heterogeneous availabilities.
- They significantly reduce the lookup latency when the peers in the same group are topologically close and cooperative caching is used within the groups.
- They facilitate the large-scale deployment of a P2P lookup service by providing administrative autonomy to participating organizations. In particular, in the hierarchical framework, each participating organization (e.g., institutions and ISPs) can choose its own lookup protocol (e.g., Chord, CAN, Pastry, and Tapestry).

3.2 Grapes Network

Grapes [12] provide the hierarchical virtual network infrastructure using physical topology information. In Grapes, nodes physically near each other construct the sub-network and the leaders of the each sub-network form the super-network. The data is inserted in both the sub-network and super-network. When the Grapes node retrieves data, it looks up the sub-network first. If there is no data, and then it looks up the super-network through its leader. At this time, the node caches the data in its sub-network. As a result, a node can find the data in its sub-network with the high probability**.**

Brocade proposed the similar hierarchical virtual network infrastructure based on physical topology. Brocade constructs a secondary overlay to be layered on top of peer-to-peer lookup systems. The secondary overlay builds a location layer between super-nodes. The nodes looking for the data, which is stored in the long distant node, find a local super-node at first. And then the super-node determines the network domain of the destination, and route directly to that domain. A super-node acts as a landmark for each network domain. A super-node is determined independently from the process of constructing peer-to-peer virtual network. Gateway routers or machines close to them are attractive candidates of super-nodes.

Grapes is different from Brocade in that it has a self-organizing mechanism. In Grapes, any node can be a leader or sub-node in the order of node insertion and the nodes construct the hierarchical virtual network in the autonomous manner without any support of a kind of server like landmark. When the node is inserted to the Grapes, it checks physical distance to the bootstrap node. If the distance is shorter than the threshold, it is inserted to the sub-network of the bootstrap node. Otherwise, it is inserted to the super-network. It checks the physical distance to every node (leader) on the path in the process of node inserting to the super-network. If it finds the leader to which the distance is shorter than the threshold, it is inserted to the sub-network of the leader. Otherwise, the new node is inserted in super network as a leader. In each layer of the virtual network, any peer-to-peer lookup algorithm can be used. This hierarchical approach brings two benefits. First, a node can find data in its sub-network with the high probability due to the data replication in its sub-network. Second, hierarchical structure makes the lookup hops shorter than those of the flat one. If the original lookup algorithm has O(log N) logical hops but Grapes have O(log $N^{1/2}$) hops.

Design of Grapes

The Grapes has two layer hierarchical structure, sub network and super-network as shown in Fig 3.1. Nodes physically near each other construct the sub-network and the leaders of the each sub-network form the super network.

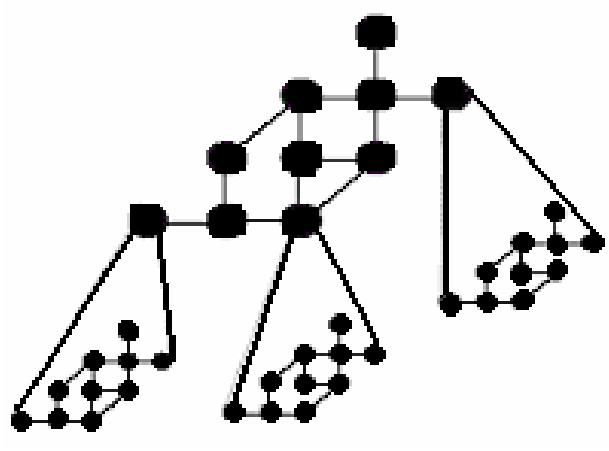


Fig 3.1 Grapes Design

Node Insertion

The new node wants to join the Grapes, it must know the contact point of at least one Grapes node, called bootstrap node. When the new node is inserted to the Grapes, the new node checks the physical distance to the bootstrap node. If the physical distance is shorter than the given threshold, the new node is inserted to the sub-network of the bootstrap node. If the bootstrap node is the leader of the sub-network, the bootstrap node will be the leader of the new node. Otherwise, bootstrap node notifies the new node of its leader. If the physical distance between the new node and the bootstrap node is longer than the threshold, the new node is inserted to the super-network. The new node checks the physical distance to the leaders on the route in the super-network. If it finds the leader to which the physical distance is shorter than the threshold, it is inserted to the sub network of the leader. If there is no leader to which the new node is inserted on the route, the new node is inserted as a node (leader) in the super-network. In each layer of virtual network, any peer-to-peer lookup routing algorithm such as CAN, Chord, Pastry, and Tapestry can be used.

In Fig 3.2 shown, the new node A joins the Grapes with the help of the bootstrap node B. If the distance between A and B is shorter than the threshold, A is located in B's sub-network (A1). Otherwise, A is not inserted to B's sub-network, but to B's super-network. A checks the physical distance to the leaders on the route in the super-network. In the fig 3.2, A is not inserted to the C (and D, E, F)'s sub-network because the distance

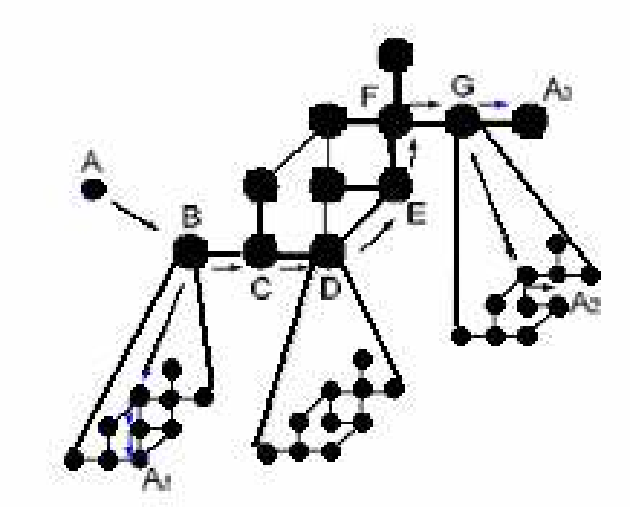


Fig 3.2 Host joining

between A and C(and D, E, F) is longer than the threshold. If the distance A and G is shorter than the threshold, A is inserted to the G's sub-network (A2). If there is no appropriate leader on the route, A is the leader in the super-network (A3).

Data insertion

When the node inserts data, it looks for the node managing the hashed key of the data in its sub-network first. The node inserts data to the node which manages the key of the data. And then the node inserts the data in the super-network. The node, with the help of its leader, looks for the appropriate leader in the super-network. If the destination leader does not have its sub-network, the data is inserted to the destination leader. If the destination leader has its sub-network, the node looks for the appropriate node in the sub-network. In this case, finally, the data is inserted to the target node in the sub network.

Data Retrieval

When the node retrieves data, it looks for the node managing the hashed key of the data in its sub-network first. If the sub-network does not have the data, the node looks up its

super-network with the help of the leader. After the node retrieves the data from the target node, it replicates the data in its own sub-network.

Scalability

In Grapes, a node can find data in its sub-network with the high probability due to the data replication in its sub-network. In addition to this locality benefit, Grapes scales well when the number of the nodes in the virtual network increases. Grapes hierarchical structure makes the lookup hops shorter than those of the flat one. Chord, for example, has O(log N) logical lookup hops. If Grapes uses Chord's lookup algorithm in both layer of the virtual network, Grapes has O(log $N^{1/2}$) logical lookup hops, on the assumption that it constructs the well balanced hierarchy, $N^{1/2}$ leaders and $N^{1/2}$ nodes in each sub network. To achieve this scalability, Grapes only needs a little bit more information in each node. In flat lookup services, the node maintains the neighbor information for lookup routing. In Grapes, the node maintains the neighbor information for its own layer of the virtual network and the leader or the first sub-node information

Leader Delegation

Leader delegation process of Grapes is as follows. Every node in the sub-network maintains its leader information. Therefore, a sub-node can notice that its leader is crashed in the process of node inserting or data insertion/retrieval. The sub-node that detects the leader's crash advertises to all the nodes in its sub network. And then the first sub-node of the leader takes over the sub-network and advertises to all the nodes in its sub-network. The first sub-node is the sub-node that joined the sub-network first of all. It maintains the neighbor information of the leader, which makes it be able to take over the leader's position in the super network. After the first sub-node took over the sub network of the old leader, a neighbor node of the new leader becomes the first sub-node of the sub-network.

Chapter 4

Proposed Scheme-VIM

Virtual Id Based M-way Lookup Protocol for Peer-to-Peer Network

4.1 Introduction

In recent years, peer-to-peer (P2P) systems have been the burgeoning research topic in large distributed system. Gnutella [1] and Napster [2] are the most famous peer-to-peer file sharing systems among these, but both of them have the scalability problem. Peer-to-peer networks like CAN[6], Chord[5], Pastry[7], Tapestry[8] try to address this problem by using Distributed Hash tables (DHT)[18].

Although each of them has different location and routing algorithms, all of them use consistent hashing [16] (like SHA-1) to let the participant nodes and objects be distributed uniformly in its virtual space. These systems can achieve fairly good load balancing, but the primitive DHT schemes have a significant disadvantage that they violate the locality property. During the locating and routing process, the messages choose the next hop to a host regardless of the physical topology information. This produces inefficient effects in response time.

To address this problem, the DHT based approaches should take into consideration the relative physical position of the participant nodes. Grapes[12] provide the hierarchical virtual network infrastructure using physical topology information. It has a two-layered overlay network, the upper layer called super-network, the lower layer called sub-network; in both layers, any DHT routing algorithm can be used. Each sub-network has a leader that forms a part of the super-network and manages the sub-network. The physically nearby nodes construct the sub-network. Moreover to improve performance, during each super-network query, the leader caches the object in its sub-network. As a result, a node can find the object in its sub-network with high probability. Because the physical distance of nodes within a sub-network is short, this infrastructure can greatly reduce the lookup distance. Although hierarchical [17] overlay network like Grapes improves the locality property of DHTs, it disturbs the decentralization property. The leader has to route all the queries of its sub-network and has to manage super-network routing too, thus becoming a performance bottleneck.

Here a scheme is proposed called VIM that solves the problem of decentralization by

distributing the network traffic between multiple hosts. Every node is a cluster of hosts dividing the traffic load among them and saving the network from a single point of failure. VIM also increases the fault tolerance of the system. A "fault" refers the situation in which a query does not reach the relevant node due to the *failure* of intermediate host(s). VIM solves this problem by sending multiple copies of a query through different paths so as to increase the probability of a query reaching its destination. Although the total load on the network gets increased, this does not affect performance because the load is already extensively divided and also it is divided evenly.

4.2 VIM Design

In this section, basic structure of VIM system is described. The overlay has two structures; the nodes having physically proximity constitute a sub-network. Each sub-network is a modified m-way [13] tree. Each node of the tree is a cluster of hosts.

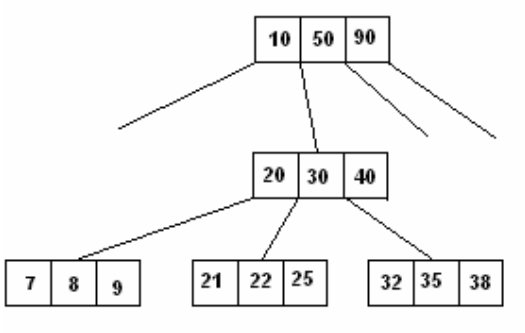


Fig 4.1 VIM Structure

The super-network is composed of the leaders of all the sub-networks. Both these networks can use any of the standard hashing [16] schemes (such as SHA-1) for locating and routing purposes. The entire root hosts (leaders of the sub-networks) are directly connected so as to get fast transmission over large distances. The graph in Fig 4.1 shows the super-network. Each host in the super-network is the root of the sub- network below it. Again, every node is a cluster of hosts. All hosts in the super-network are directly connected so as to get fast transmission over large distances.

4.2.1 The fundamental Hierarchy: Modified M-way Tree

It is basically an m-way search tree [13] with a restriction that a node can have children only after it has 'm-1' elements. We call the above tree as "modified m-way tree". Each sub-network is a modified m-way tree. To insert in VIM at first, we search for the element

to be inserted. Then we try to insert the element into the cluster at which the search terminates. If this cluster already has 'm-1' hosts then the new host is inserted as a child. Otherwise it is inserted into that cluster itself. For deletion in the VIM we replace the leaving host by the host with the largest key in its left sub-tree or by the host with smallest key in its right sub- tree. Sometimes there might be a need of a rearrangement at the leaf level to complete the deletion process. Consider Fig 4.2. If we delete the element with key 50, it is replaced by the largest element in its left sub-tree, which are 40. Now the cluster, which had 40 as an element, will have to do a rearrangement to get 38 at its position as shown in the Fig 4.3.

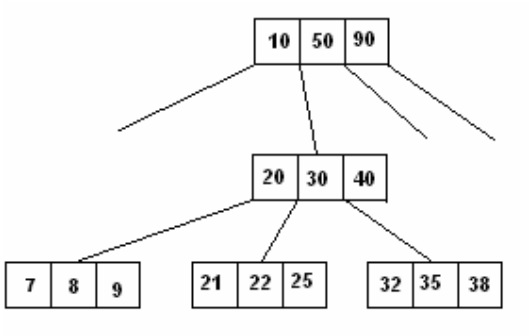


Fig 4.2 The state of the tree when 50 is about to log off

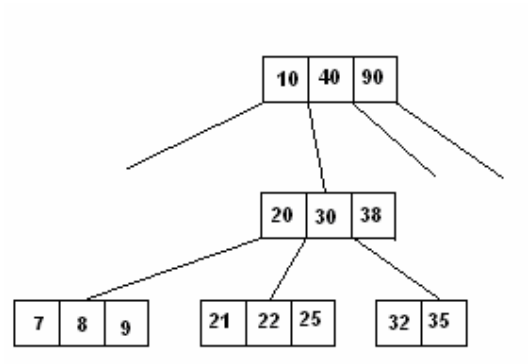


Fig 4.3 The rearrangement after the deletion of 50

Also for the purpose of intra-cluster management like insertion and deletion we have a leader in each cluster. The host with smallest key becomes the leader and while leaving the node, it assigns the leadership to the host with the next smallest key.

4.2.2 The parent child relationship

All the children of a certain node are divided equally among the hosts of that node. "Divided" here is in terms of queries and maintenance. Fig 4.4 illustrates the concept.

Node n_1 is a cluster of hosts A, B and C. The first host of the parent cluster is linked to the first host of each child cluster. In Fig 4.4, A communicates with hosts A1, A2, A3 and A4, B communicates with hosts B1, B2, B3 and B4 and so on.

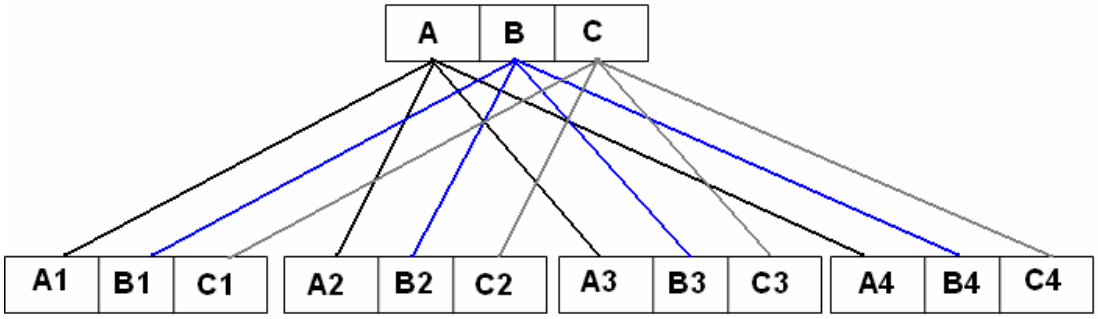


Fig 4.4 Division of Children

This type of arrangement helps in query replication which will further enhance the fault tolerance.

4.2.3 Virtual Address Assignment

Each node in the VIM structure is being assigned a virtual address. In addition to virtual address of nodes, each host is also assigned a virtual address. Root node is assigned as "0" virtual address and "0.0", "0.1" for 1st, 2nd host respectively.

Let node N has virtual address "0201" then its 2nd child node will have "02011" as virtual address and its 2nd host will have "02011.1" virtual address.

The concept of virtual address will lead to reduce the lookup path length as well as it reduces the lookup traffic on the root node.

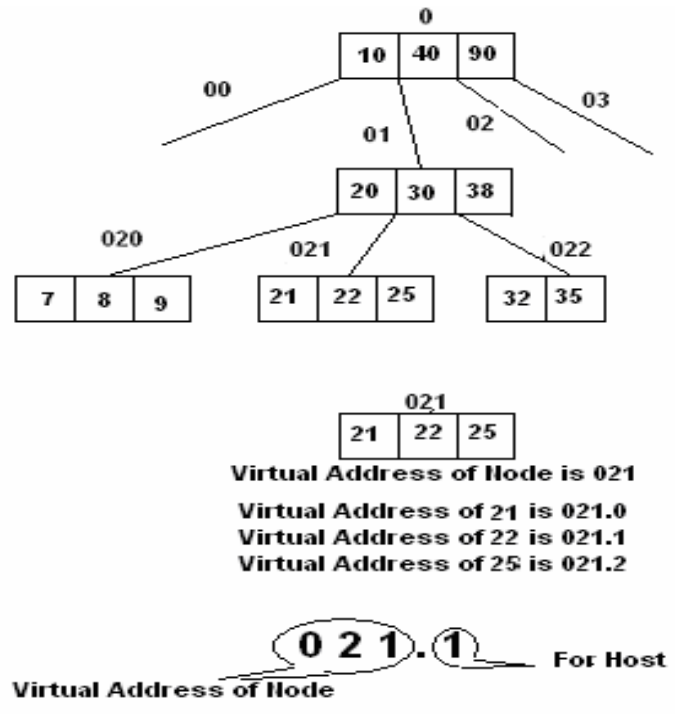


Fig 4.5(a) Virtual Addresses

- Personal Address Book (PAB)

Each node maintains a PAB, which contain virtual address and key pair of all its children, which further reduces the traffic in lower part of the tree, and it also reduces the lookup path length.

Let's take an example in which host 21 searches for host 32. First of all host 21 will search its in its own siblings, if fails then search is transfers to subnet leader of host 21 which is host 30 now host 30 will search host 32 in its own siblings. On failure host 30 will search its own PAB.

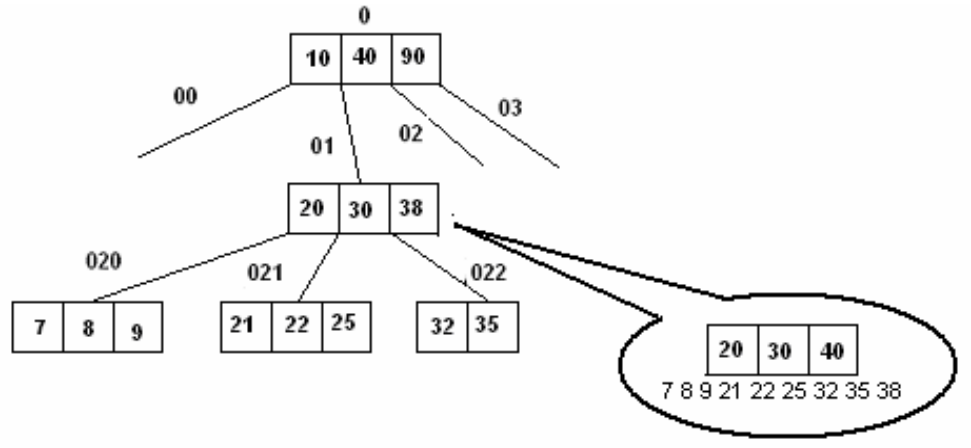


Fig 4.5(b) Personal Address Book

- Common Address Book (CAB)

VIM also maintains CAB, whenever host joins the network it's entry is

registered in the CAB. The data stored in the CAB is sorted according to the key so that we can search the element using binary search, which reduces the searching complexity. Sample of Common Address Book is given below:

Virtual Address	Key
00.0	1
00.1	4
00.2	17
00.3	30
0.0	31
0100.0	33
0100.1	56
0100.2	57

Fig 4.5(c) Common Address Book

4.2.4 Total Decentralization

Each node contains multiple hosts due to which load will be divided among them [14] [15]. Although load gets divided among hosts but there may some chances that the host is unable to handle query. To deal with this situation the host from leaf nodes is requested to share the load which is called the *Assistant host*. As the assistant host will always be leaf host that's why it would be free most of the time. So the choice of leaf host as assistant host will be the best option. As the hosts load will be more than the predefined limit then it well inform its parent that extra queries will be sent to assistant host. As host '20' becomes overloaded then it request host '7' which will behave as assistant to host '20'. Host '20' will inform its parent host '10' about its assistant host '7', so that host '10' will forward the load to host '20' only when its load is below the predefined limit otherwise it will forward the load to host '7' (assistant host of host '20'). Let host '10' is at level h and host '20' is at level h+1 and host '7' is at leaf level. But due to above described scheme host '7' will behave like host at level h+1 (refer Fig 4.6 and Fig 4.7).

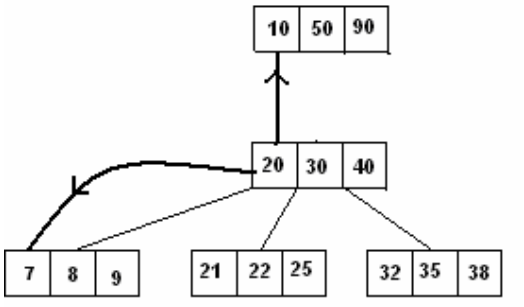


Fig 4.6 B informs its parent about X and sends its address book to X

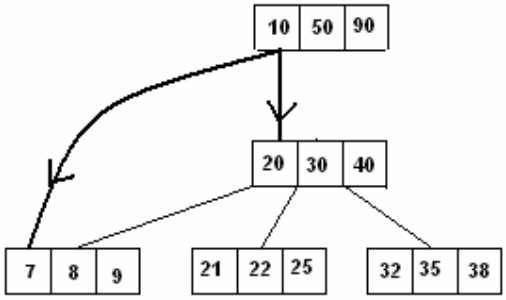


Fig 4.7 Network traffic gets distributed between ‘20’ and ‘7’.

The above scheme will be performed until the load on host ‘20’ become affordable. If host ‘7’ will also gets overloaded then it will also search for another leaf level host which will be its assistant host or we can call the new host as assistant host of host ‘20’ of degree two.

4.2.5 Query replication

Query replication means that a single query will turn into ‘r’ (replication factor) queries, which will search for the desired host in parallel. The source of the query will perform an operation (GET_OFFSET), on source’s virtual Id and destination’s virtual Id, which results into the offset from the source host so that the destination host will be searched fast. Now from resultant location in the structure, query is replicated by ‘r’ factor and the replicated query will be forwarded to its corresponding child. Consider an example Fig 4.8 host X (80) generates a query for ‘6’. Now at first the search will start from the Node Y (Node Y is computed from the GET_OFFSET operation). In the given figure replication factor is 3. Replicated query starts from three hosts i.e. 40, 50 and 60 in Node Y. These hosts then find the appropriate child node and pass it to their respective children in that node. Again these hosts pass it to the relevant child node. Finally, when the query reaches the destination node, brothers 7, 8 pass it to 6. The above scheme shows that the query fails only when one host on each of the paths fail simultaneously. This mechanism greatly reduces the probability of a fault.

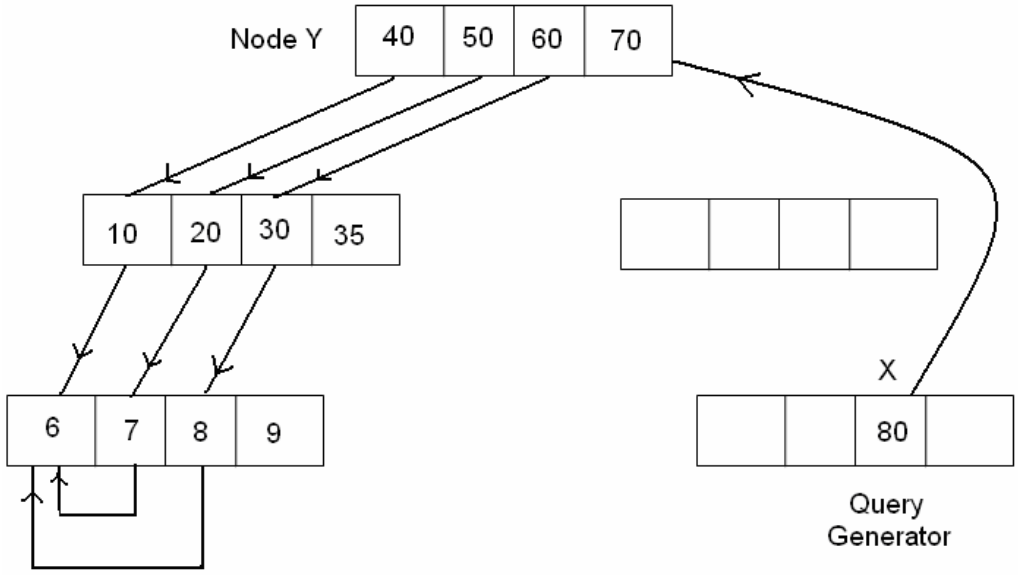


Fig 4.8 The flow of a query, 80 originate a query for 6, which follows the above path.

Fig 4.8 is self-explanatory. Let us assume that host 10, 20 fails then first two replicas of query will not reach to the destination but query through 30 will reach to 8 and then host 8 will forward the query to the host 6 which is the desired destination of 80(X).

Let r_q is Number of Query Replicas

n_f is Intermediate hosts fails (one host per path)

With the above example we come the conclusion that

$$r_q < n_f \Rightarrow \text{No Result}$$

$$r_q > n_f \Rightarrow \text{Result}$$

4.3 THE VIM PROTOCOL

This section will discuss the entire procedures of insertion and deletion in VIM.

4.3.1 Host Insertion

Any host can join the network only when it has got invitation from existing host in the network. Existing hosts continuously send invitation to the near by host. If any external host is interested then it will send a request back to the existing host. After this existing host will request its sub network leader to join the new host. The sub network leader search for the key of the host. Where the search terminates new host will be inserted at that location. This mechanism also reduces the overhead for joining the new host. As the existing host can send the invitation to it's near by hosts which are having distance less than threshold value. So

finally the new host will be inserted at proper position. Figure 4.9 describes the pseudo code for the host insertion.

```
hostInsert (node.host.key)
{
        if (node==NULL)
        {
                node.Assign(host.key);
                host.Assign(Virtual Address);
                Update (node.PersonalAddressBook);
                Update (CommonlAddressBook);
        }
        else if ( node.cnt < Order(M) )
        {
                pos=binsearch(node.ht.node.cnt);
                Insert the host.key at "pos" Position in "node";
                Node.cnt++;
                Update (node.PersonalAddressBook);
                Update (CommonlAddressBook);
        }
        else
        {
                Pos=binsearch(node.ht.node.cnt);
                hostInsert(node.ptr[pos].host.keys;
        }

}
```

Fig 4.9 Pseudo code for the host Insertion

4.3.2 Host deletion

When a host 'H' logs off the network, it carries out the following procedure: it informs all the brothers and the parent host about its departure. In case the leaving host is the cluster-leader it appoints a new leader (which is the host with the smallest key). Now the cluster-leader searches for a replacement 'R' for the leaving host 'H' (from a leaf node, no replacement is required if the leaving host is already in a leaf node). The replacement host, R before leaving its old node informs all its relatives about its departure. R takes its new address book from H and finally informs its new relatives about its arrival. Figure 4.10 describes the pseudo code for the host Deletion.

```
hostDelete(node.pos.key)
{
        if( ChildPresent (Right Siblings))
        {
                Shift First most key to the key to be deleted;
                hostDelete ( node.pos.current.key );
                Update(node.PersonalAddressBook);
        }
```

```
        else if (ChildPresent ( Left Siblings ) )
        {
                Shift Last most key to the key to be deleted;
                Rearrange(node.key[ ] );
                hostDelete ( node.pos.current.key );
                Updaste(node.PersonalAddressBook);
        }
        else
        {
                Delete key in the node;
                Rearrange(node.key[ ] );
                Updaste(node.PersonalAddressBook);
        }

        Inform( Siblings );
        Update( CommonAddressBook);
}
```

Fig 4.10 Pseudo code for the host Deletion

4.3.3 Host Failure

If a host H goes off the network without informing any other host, such a situation is referred to as 'host failure'. In such a situation the host that discovers its failure first, X has to inform all other related hosts. The brothers and the parent of a host ping it at regular intervals so a brother or its parent either discovers a failure. Under the first possibility, the brother X informs about the leaving host H to the H's relatives and then to the leader of the node. If H was the leader then the host with the smallest key becomes the new cluster leader. If X is the parent of H then it informs all hosts in H's node. Now, the cluster-leader of H's node carries out all the operations as in the case of host deletion (finding a replacement and then giving it all the information about H). The total time that is required from the point of failure to the moment when finally the replacement R informs everyone about its arrival is called T_R (or recovery period). The fault tolerance of the network is directly dependent on this parameter. The higher the time it takes to replace the failed node higher will be the probability of a query getting lost or stuck somewhere in the path. Figure 4.9 describes the pseudo code for the host insertion. Figure 4.11 describes the pseudo code for the host failure.

```
hostFailure ( node.host )
{
        node.H.failFound( host );
        node.H.inform( All Siblings);
        node.H.inform(Parent);
        node.h.inform(host);

        if(IsLeader(host) )
```

```
        {
                Nominate ( leader );
        }

        For ( All host in the node )
        {
                Update ( All Pointer);
                Update ( node.PersonalAddressBook);
                Update ( CommonAddressBook );
        }
}
```

Fig 4.11 Pseudo code for the host Failure.

4.3.4 Query

The originator of a query sends it to 'r' (replication factor) hosts in the node computed by GET_OFFSET operation. Every host on receiving a query checks its brothers and forwards it to him if his key matches the search otherwise forwards it to the relevant child. While forwarding a query to the child, a host checks if it has already forwarded more queries than the child's bandwidth limit (the child is loaded). If it is so, it sends the query to the assistant host in the leaf node, which is sharing the load of the child host. The query searching mechanism is the same as that in an m-way search tree, but the query proceeds through r parallel paths to increase fault tolerance. This query is first searched in the sub-network and on failing to get a positive response from the sub-network; the leader then forwards the query to the super-network. Figure 4.12 describes the pseudo code for the Query.

```
Lookup( srcNode , key )
{
        destVID = SearchCAB ( key );
        srcVID = srcNode.VID;
        offset = GET_OFFSET ( destVID,srcVID );
        searchStartNode = moveUp ( srcNode , offset );
        pathLength = lookupKey ( searchStartNode , key , QueryRepCnt);
}
```

Fig 4.12 Pseudo code for the Query.

Chapter 5

Performance Analysis of VIM

In this chapter simulation results and analysis of VIM is presented. The VIM simulation software was implemented in C++. Some of the computations have been done using java. The following parameters are used to evaluate VIM:

- Path Length
- Fault tolerance
- Effect of Order of tree (M) on percentage of success

While conducting experiments on the simulation the following parameters were taken into account:

Number of hosts (N)	This is an important parameter, which shows the scalability of the network.
Cluster Size (m-1):	This is a crucial parameter which has its own tradeoffs under different requirements and can significantly affect the performance of the network, especially the number of hops (path length) and consequentially the look up [15] latency. Also, this parameter affects the fault tolerance of the system.
Replication Factor (r):	This factor indicates the number of copies of a query that is originally sent to the sub-network leader so as to facilitate multiple parallel paths resulting in a fault tolerant system.

5.1 Path length

Effect on the path length by the failure of the hosts is shown in this section. Results are shown for 10, 20 and 30 percent failure of hosts. These experiments reveal that with the use of Common Address Book, the path length of the query become shorter. The following graph contains data for Chord, VIM with CAB, and VIM without CAB. Results shows that Chord gives the longest path length and VIM with CAB results in shortest path length. So, by using CAB in VIM path length drastically reduces. Fig 5.1, 5.2, 5.3 shows the results for 10%, 20% and 30% failure respectively. First set of data in all the three graphs (250,500,750…2000) is corresponding to order (modified

m-way tree) 5, second for 7, third for 9 and fourth for 11. These graphs reveal that as the order of tree increases, the path length decreases.

Order of Tree ∝ 1/Path Length
Path Length ∝ No of Hosts in Network

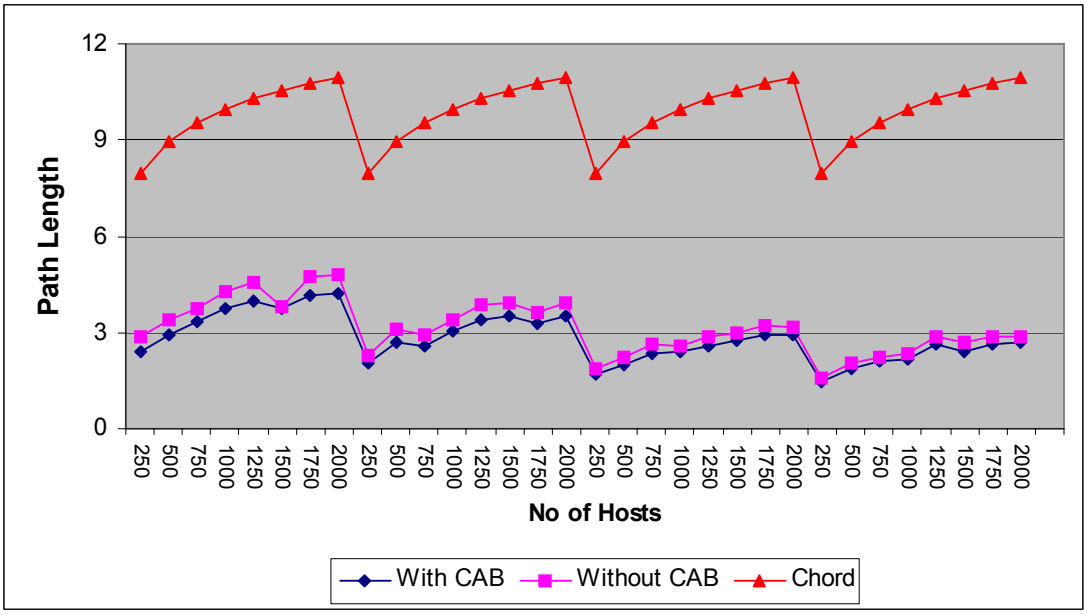


Fig 5.1 Path Length with 10% Host Failure

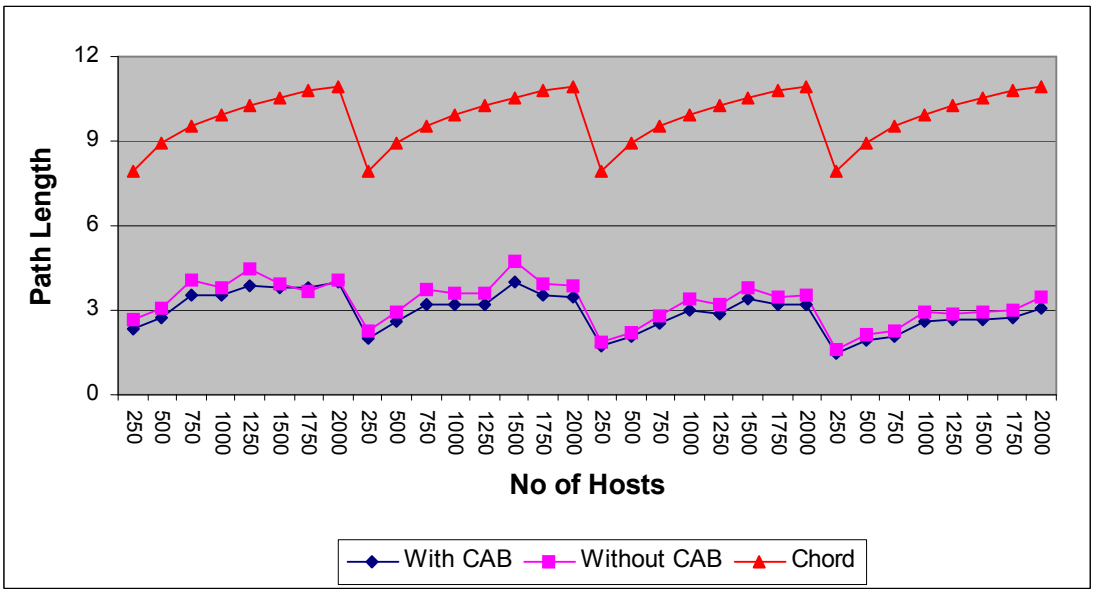


Fig 5.2 Path Length with 20% Host Failure

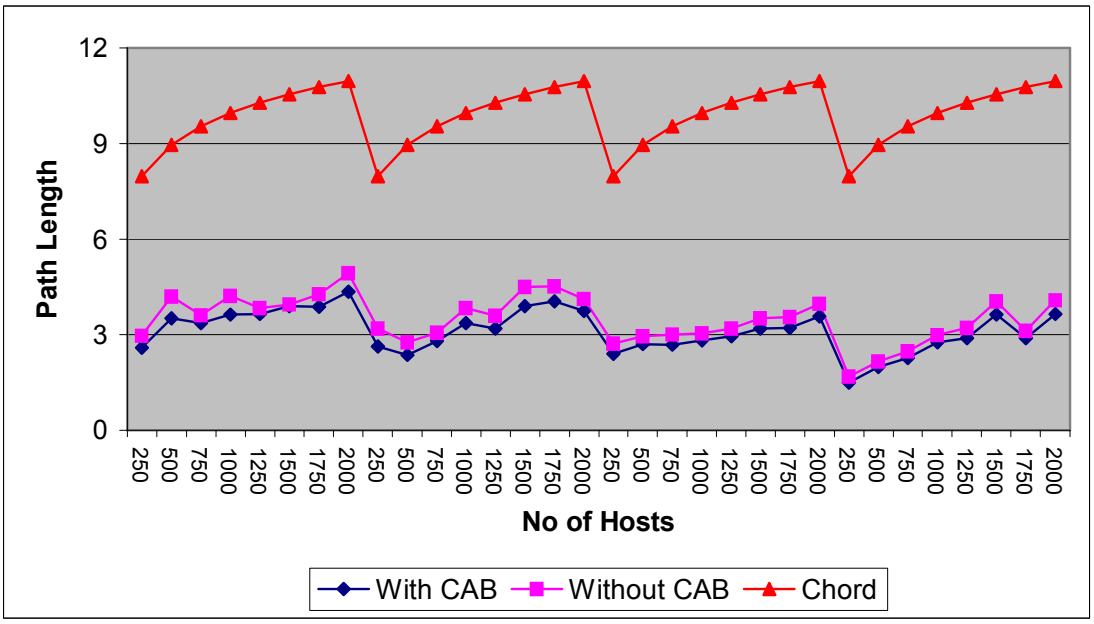


Fig 5.3 Path Length with 30% Host Failure

From these graphs we concluded that as the number of host failure increases the path length also increases. One more result that comes out of these graphs is that as the order of tree increases path length decreases. Higher the order of tree shorter will be the height of tree and finally shorter will be the path length.

5.2 Fault Tolerance

Next we evaluated the impact of a failure on VIM's performance and on its ability to perform correct lookups. Fig 5.4, 5.5, 5.6 shows the graphs with % Success for 10%, 20%, 30% failure respectively. First set of data in all the three graphs (250, 500, 750… 2000) is corresponding to order (modified m-way tree) 5, second for 7, third for 9 and fourth for 11.

% Success ∝ Order of Tree

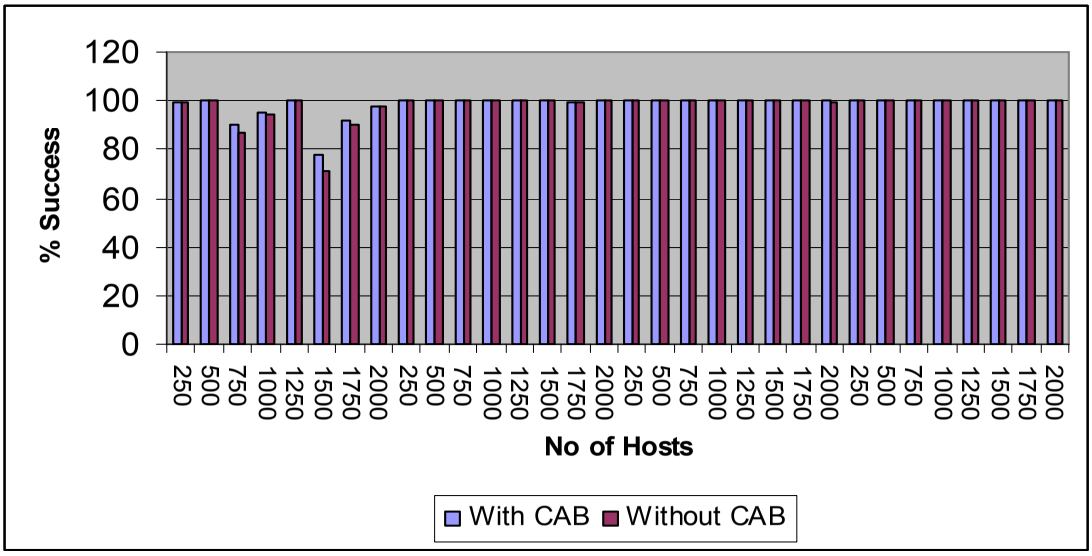


Fig 5.4 % Success 10% Hosts Failure

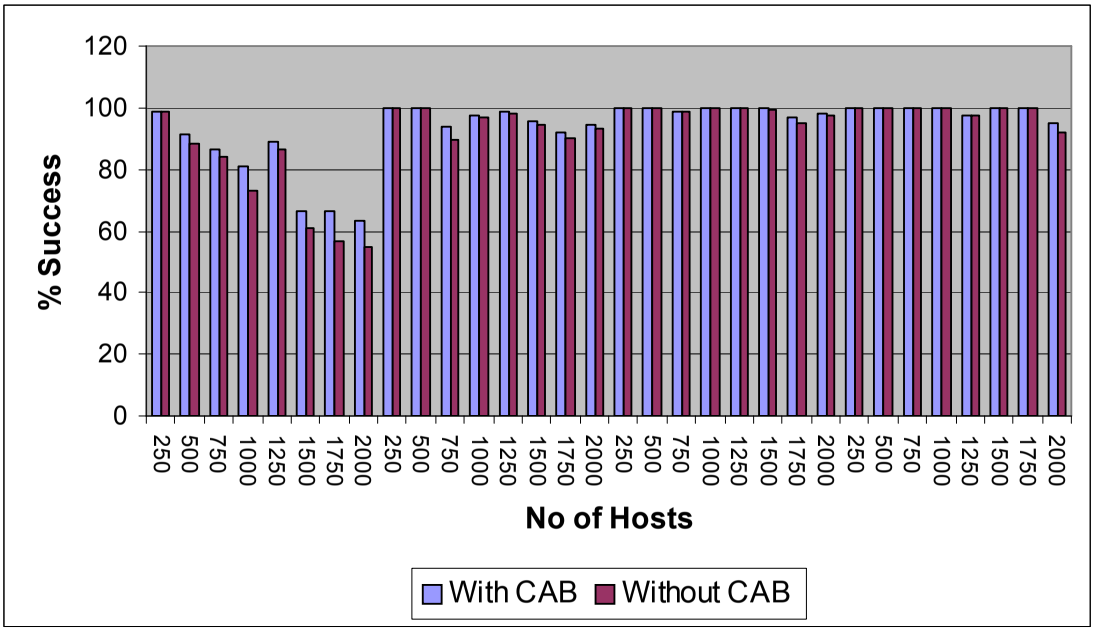


Fig 5.5 % Success 20% Hosts Failure

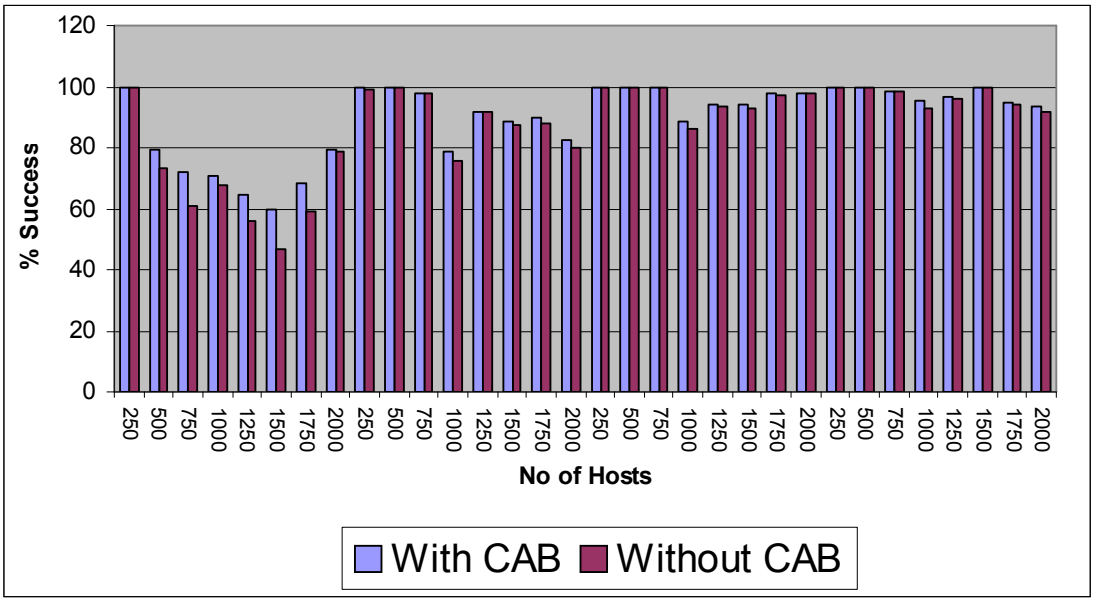


Fig 5.6 % Success 30% Hosts Failure

As order of tree increases, height of tree decreases which in turns reduces the path length of the queries. As path length of queries reduces then chances of failure of intermediate hosts will be reduced and hence percentage of success will increases.As the number of failure increases % of success decreases, but percentage success of query increases with the use of the CAB.

5.3 Effect of Order of tree (M) on Percentage of Success

Next we evaluated the impact of order of tree (m) on VIM's performance and on its ability to perform correct lookups. Fig 5.7, 5.8, 5.9 shows the graphs with % Success for 1000, 1500, 2000 host respectively. First set of data in all the three graphs (10, 20,30,40,50 %) is corresponding to order 5, second for 7, third for 9 and fourth for 11.

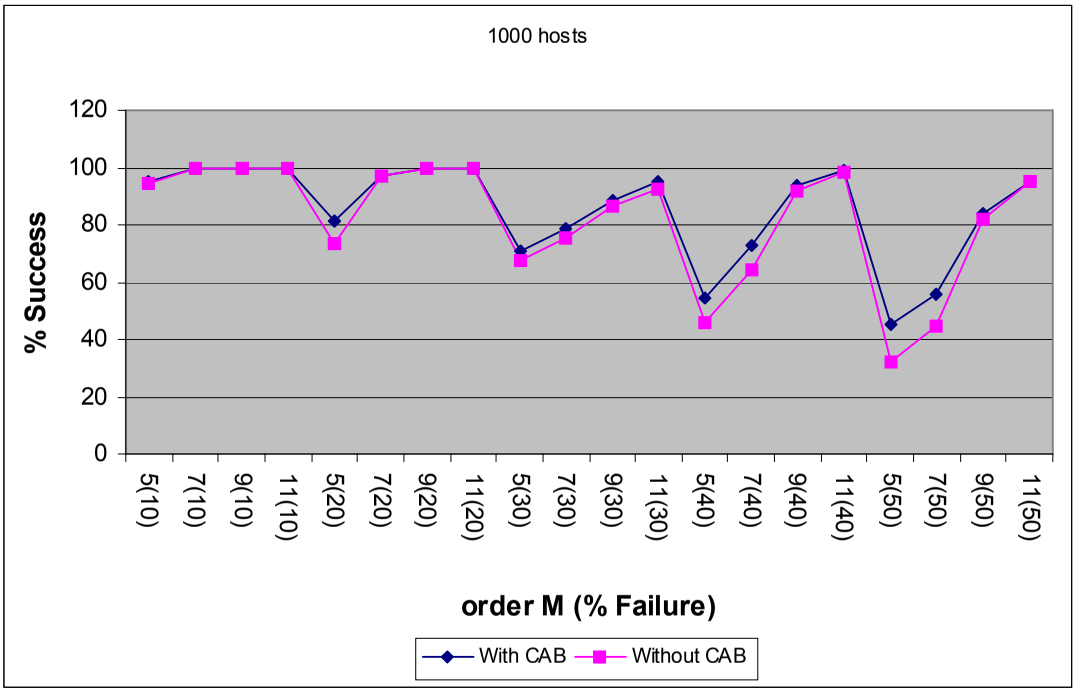


Fig 5.7 1000 Hosts

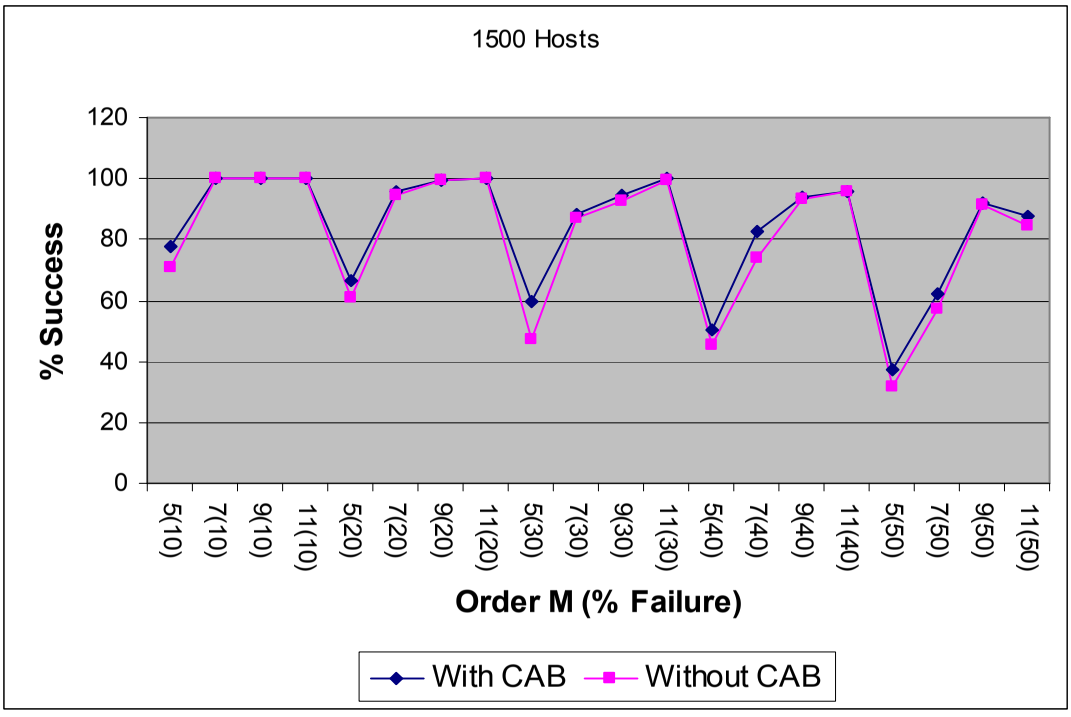


Fig 5.8 1500 Hosts

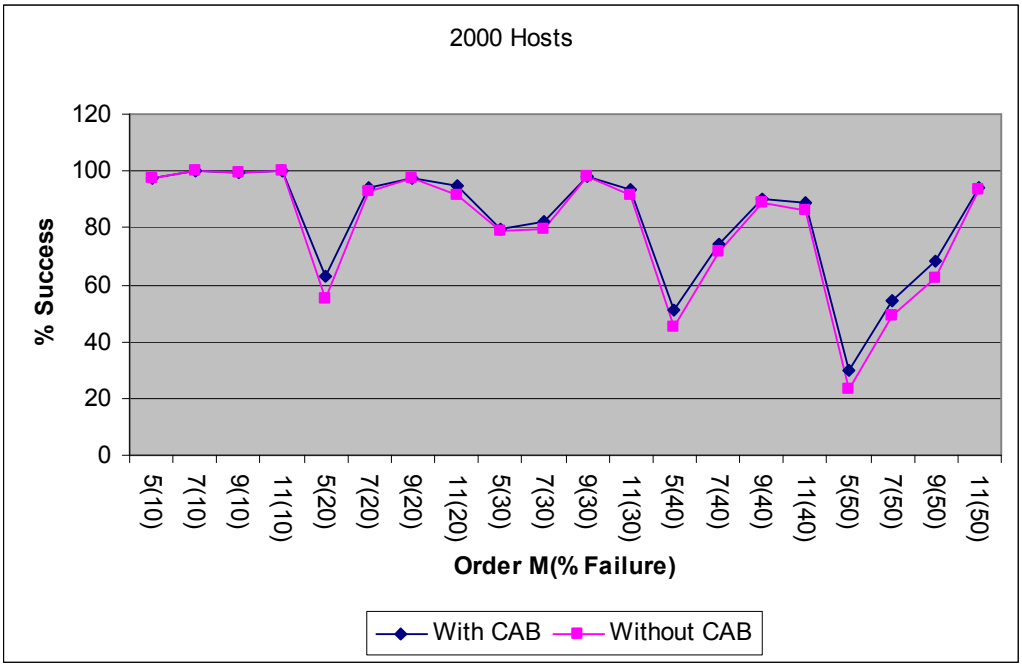


Fig 5.9 2000 Hosts

Above graphs also illustrates that as % failure increases for same order of tree (m) then % success decreases. Order of tree (m) increase for same % failure then % success will increase.

Finally % success for VIM with CAB is better than VIM without CAB and Chord.

Chapter 6

Conclusion and Future work

Conclusion

In this dissertation, we have proposed a scheme which is based on M-way search tree. Along with this scheme, five lookup schemes have been discussed here. Two of them are based on the non-DHT pattern for peer-to-peer systems and three of them are based on the DHT pattern.

The proposed scheme is also based on DHT and hierarchical topology which exploits the M-way search tree in modified style. The proposed scheme enhances the fault tolerance and decentralization which are two important requirements of peer-to-peer systems.

The proposed scheme exploits the proximity between hosts and also provides fault tolerance through query replication. Geographically closer hosts form the sub-network. Query forwarding is done is some different way where n^{th} host will forward the query only to n^{th} host in the child node (cluster), this pattern of query forwarding has given the way for query replication. Query replication and its passage through different paths results in a high degree of fault tolerance.

Use of multiple hosts at each node, distributes the network load between hosts and hence an appreciable degree of decentralization is achieved. Further, the concept of *assistant host* also ensures total decentralization among participant hosts.

The use of virtual identity for each node (cluster) as well as for each host enables us to use the concept of Personal Address Book (PAB) and Common Address Book (CAB). Each node maintains a PAB which contains information about its children which results in satisfying the query early. CAB is common for all the hosts which maintain the key and virtual Id of the hosts present in the networks. Whenever a query is fired, first of all it will search the PAB of that node and then on failure it will use the CAB to extract the VID of the destination host, now source VID and destination VID will used by GET_OFFSET function to compute the offset, by which query will move up in the tree to start the search. This procedure will reduce the path length as well as increases the percentage of success in lookup of host.

To simulate this scheme, we have used modified M-way search tree in which a node can not have child until it has (m-1) hosts. In addition to maintain links between parent node and child node this tree maintain links between parent host and child host which helped in query replication.

With performance analysis, we have shown the comparison between path length of CHORD, VIM without CAB and VIM with CAB (% of success for queries of CHORD, VIM without CAB and VIM with CAB) which clearly indicates that VIM with CAB is having better performance.

We have shown in chapter 5 that this scheme will successfully search the destination host even if 50% of hosts have failed.

Future Work

In this dissertation, we have used common address book which is being used by the entire host (available in network) for extracting the VID of destination host. To extend the same scheme in the future, we can incorporate the concept of replica of CAB which will reduce the traffic at CAB. In addition to this, we can consider for designing an adaptive replicated hierarchical network which may further reduces the path length as well as increases the percentage of success for lookup of hosts.

References

[1] Gnutella: Link: http://www.gnutella.com

[2] Napster: Link http://www.napster.com

[3] Wikipedia, the free encyclopedia: “Peer-to-Peer” Implementing M-Way Search Trees, Data Structures and Algorithms with Object-Oriented Design Patterns.

[4] L. Garc´es-Erice, E.W. Biersack, P.A. Felber, K.W. Ross, and G. Urvoy-Keller: Hierarchical Peer-to-peer Systems.

[5] I. Stoica, R. Morris, D. Karger, M F. Kaashoek and H. Balakrishnan: Chord: A Scalable - Peer-to-peer Lookup Service for Internet Applications. Proceedings of SIGCOMM 2001, ACM.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker: A Scalable Content Addressable Network. Proceedings of SIGCOMM 2001, ACM.

[7] A.Rowstron and P.Druschel: Pastry: Scalable Distributed Object Location and routing for Large-scale Peer-to-peer Systems, In Proceedings of IFIP/ACM Middleware 2001.

[8] B. Y. Zhao, J. D. Kuibiatowicz and A. D. Joseph: Tapestry: An Infrastructures for Fault-tolerant Wide-area Location and Routing, Tech. Rep.UCB/CSD-01-1141, UC Berkeley, EECS, 2001.

[9] DanielA.Menascé, George Mason University: Scalable P2P Search, IEEE internet computing, 2003

[10] B. Yang, and H. Garcia-Molina: Improving Search in Peer-to-Peer Networks, ICDCS 2002, Vienna, Austria, July 2002.

[11] Daisuke Takemoto, Shigeaki Tagashira and Satoshi Fujita: Distributed Zone Partitioning Schemes for CAN and Its Application to the Load Balancing in Pure P2P Systems, IPSJ Digital Courier 2005.

[12] K. Shin, S. Lee, G. Lim, H. Yoon, and J. S. Ma: Grapes: Topology-based Hierarchical Virtual Network for Peer-to-peer Lookup Services. Proceedings of the International Conference on Parallel Processing Workshops (ICPPW' 02).

[13] B.B Karki: B-trees, csc3102, Link: www.csc.lsu.edu/~karki/DA-07/DA09.pdf

[14] J. Byers, J.Considine, and M. Mitzenmacher: Simple Load Balancing for Distributed Hash Tables, In Proceedings of the 2nd International Workshop on peer-to-Peer Systems, 2003.

[15] Ananthapadmanabha Rajagopala-Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, Ion Stoica: Load Balancing in Structured P2P Systems, In Second International Workshop on Peer-to-Peer Systems (IPTPS), February 2003, Berkeley, CA.

[16] FIPS 180-1: Secure Hash Standard. U.S. Department of Commerce /NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[17] Roman Kurmanowytsch, Mehdi Jazayeri and Engin Kirda: Towards a Hierarchical, Semantic Peer- to-Peer Topology, In Proceedings of the Second International Conference on Peer-to-Peer Computing (P2P'02) 2002.

[18] Ming Xie: P2P systems based on distributed hash table, University of Ottawa, September 2003.

[19] Hari Balakrishnan, M. F. Kaashoek, David Karger, Robert Morris, Ion Stoica: Looking Up Data in P2P Systems, Communications of the ACM, Vol. 46, No. 2, February 2003, pp. 43-48.

[20] Richard Hsiao and Sheng-De Wang: Jelly: A Dynamic Hierarchical P2P Overlay Network with Load Balance and Locality, In Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSW'04), 2004.

[21] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee and Pete Keleher: Adaptive Replication in Peer-to-Peer Systems, Department of Computer Science, University of Maryland, College Park

Appendix

Source Code

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<math.h>
#define M 10
#define S 3*M
#define MM M*(M+1)
#define H 30000
#define LK 5

struct addrBook
{
      char vaddrnd[S];
      char vaddrht;
      int key;
};
addrBook CentralABook[H];
int per,perQ;
int cntCABook=0;
FILE *fp=fopen("output.txt","w");
FILE *fp1=fopen("output-Book.txt","w");

FILE *fp2=fopen("percentage.txt","w");
FILE *fp3=fopen("result.xls","a");
int dHost[H]={0};
int dcnt=0;
struct vaAddr
{
      char vand[S];
      char vaht;
};
struct host
{
      int key;
      host *lnk[M+1];
      vaAddr vahost;
};

struct node
{
      int cnt;
      host ht[M];
      node *ptr[M+1];
      node *par;
      char vanode[S];
      addrBook abook[MM];
} ;

struct useHostList
{
      node *ptr;
      int key;
}UHL[H],ADUHL[H];

int cntUHL=0;
int cntADUHL=0;

struct query
{
      int src,dest;
}qList[H];
int cntQList=0;

node *root;
```

```
node* initNode()
{
    int i,j;
    node *nd;
    nd=new node;
    nd->cnt=0;
    nd->par=NULL;
   // nd->vanode="N";
    for(i=0;i<M;i++) {  (nd->ht[i]).key=-1;  }
    for(i=0;i<=M;i++) {  nd->ptr[i]=NULL; }
    for(i=0;i<M;i++)
    {
       for(j=0;j<=M;j++)
       {
          (nd->ht[i]).lnk[j]=NULL;
       }
     }

    return nd;
}

int binsearch(int x, host *a, int n)
{
int i, left, right;
                if (x <= a[0].key)
                                return 0;
                if (x > a[n-1].key)
                                return n;
                left = 0;
                right = n-1;
                while (right -  left > 1){
                                i = (right + left)/2;
                                if (x <= a[i].key)
                                                right = i;
                                else
                                                left = i;
                }
                return(right);
}

void LocalAddrBookUpdate(host h,node **p)
{
        for(int j=0;j<MM;j++)
        {
          if( !(strcmp((*p)->abook[j].vaddrnd,h.vahost.vand)) && (*p)->abook[j].vaddrht == h.vahost.vaht)
          {
             (*p)->abook[j].key=h.key;

             break;
          }
        }
}

void createLocalABook(addrBook *abk,node **p)
{

  char st[S],sth[S],stn[S],ht,nd;
  int i,j,k,t,h,n;

  k=0;
  for(i=0;i<M+1;i++)
  {

    for(j=0;j<M;j++)
    {
     strcpy(st,(*p)->vanode);
     n=i+48;
     h=j+48;
     nd=n;
     ht=h;
     stn[0]=n;
     stn[1]='\0';
     strcat(st,stn);
     strcpy(abk[k].vaddrnd,st);
     abk[k].vaddrht=h;
```

```
      abk[k].key=-1;
      k++;
     }
    }
}


int ABSearch(int x)
{
       int l=0,m,i,u=cntCABook-1;
       i=-1;

       while(u>=l)
       {
        m=(u+l)/2;
        if(x==CentralABook[m].key)
        {
         i=m;
         break;
        }
        else
        {
           if(x>CentralABook[m].key)
             l=m+1;
           else
             u=m-1;
         }

       }
  return (i);




}


void updateCentralABook(node * t)
{
              int i, *k, n,j;
              int z;

              if(t==root)
                            cntCABook=0;

              if (t != NULL)
              {
                            for (i=0; i<t->cnt; i++)
                            {
                                          strcpy(CentralABook[cntCABook].vaddrnd,t->ht[i].vahost.vand);
                                          CentralABook[cntCABook].vaddrht= t->ht[i].vahost.vaht;
                                          CentralABook[cntCABook].key=t->ht[i].key;
                                          cntCABook++;
                            }

                            for (i=0; i<=t->cnt; i++)
                            updateCentralABook(t->ptr[i]);
              }
}

void sortCentralABook()
{
    addrBook temp;
    for(int i=0;i<cntCABook-1;i++)
    {
      for(int j=0;j<cntCABook-1-i;j++)
      {
         if(CentralABook[j].key > CentralABook[j+1].key)
         {
          temp=CentralABook[j];
          CentralABook[j]=CentralABook[j+1];
          CentralABook[j+1]=temp;
         }
       }
```

```
    }
}



void insert(int i,node **nd,node **par,int t)
{
    int p,k,l,m,z,n;
    node *tmp;
    char ch;
    char st[2],st1[2],st2[2];
    addrBook abk[MM];

    if(*nd==NULL && *nd == root)
    {
       *nd=initNode();
      (*nd)->cnt=1;
      (*nd)->ht[0].key=i;
      (*nd)->par=*par;
      strcpy((*nd)->vanode,"0");

     for(m=0;m<M;m++)
      {
         z=m;
         z=z+48;
         ch=z;
         strcpy((*nd)->ht[m].vahost.vand,(*nd)->vanode);
         (*nd)->ht[m].vahost.vaht=ch;
      }
      createLocalABook((*nd)->abook,nd);
      updateCentralABook(root);
      sortCentralABook();
    }
    else if(*nd ==NULL && *nd !=root)
    {
        *nd=initNode();
       (*nd)->cnt=1;
       (*nd)->ht[0].key=i;
       for( k=0;k<M;k++)
       {
         (*par)->ht[k].lnk[t]=&(*nd)->ht[k];
       }
       (*nd)->par=*par;

       strcpy((*nd)->vanode,(*par)->vanode);

       t=t+48;
       ch=t;
       st[0]=ch;
       st[1]='\0';
       strcat((*nd)->vanode,st);

       for(m=0;m<M;m++)
       {
        z=m;
        z=z+48;
        ch=z;

        strcpy((*nd)->ht[m].vahost.vand,(*nd)->vanode);
        (*nd)->ht[m].vahost.vaht=ch;
        }

        createLocalABook((*nd)->abook,nd);
        LocalAddrBookUpdate((*nd)->ht[0],par);
                              updateCentralABook(root);
                              sortCentralABook();
    }
     else if( (*nd)->cnt < M)
     {
         p= binsearch(i,(*nd)->ht,(*nd)->cnt);
                               for (int j =(*nd)->cnt; j > p; j--)
         {
                                          (*nd)->ht[j].key=(*nd)->ht[j-1].key;
                                 if( (*nd) != root)
```

```
            {
              LocalAddrBookUpdate((*nd)->ht[j],par);
            }
          }
          (*nd)->ht[p].key=i;
          (*nd)->cnt =(*nd)->cnt+1;
          if( (*nd) != root)
          {
            LocalAddrBookUpdate((*nd)->ht[p],par);
          }
                              updateCentralABook(root);
                              sortCentralABook();
                    }
      else
      {
          p= binsearch(i,(*nd)->ht,(*nd)->cnt);
          if ( i == (*nd)->ht[p].key)
            cout<<"Duplicate found at :"<<(*nd)->ht[p].vahost.vand<<"."<<(*nd)->ht[p].vahost.vaht<<"\n";
          else
            insert(i,&(*nd)->ptr[p],nd,p);

      }
}


void fillUHL(node * t)
{
              int i, *k, n,j;
              int z;
              if (t != NULL)
              {
                        for (i=0; i<t->cnt; i++)
                        {
                        UHL[cntUHL].ptr=t;
                        UHL[cntUHL].key=t->ht[i].key;
                        cntUHL++;

                        ADUHL[cntADUHL].ptr=t;
                        ADUHL[cntADUHL].key=t->ht[i].key;
                        cntADUHL++;
                        }
                        for (i=0; i<=t->cnt; i++)
                        fillUHL(t->ptr[i]);
              }
}

void sortUHL()
{
    useHostList temp;
    for(int i=0;i<cntUHL-1;i++)
    {
      for(int j=0;j<cntUHL-1-i;j++)
      {
         if(UHL[j].key > UHL[j+1].key)
         {
          temp=UHL[j];
          UHL[j]=UHL[j+1];
          UHL[j+1]=temp;
         }
      }
    }
}


void sortADUHL()
{
    useHostList temp;
    for(int i=0;i<cntADUHL-1;i++)
    {
      for(int j=0;j<cntADUHL-1-i;j++)
      {
         if(ADUHL[j].key > ADUHL[j+1].key)
         {
          temp=ADUHL[j];
          ADUHL[j]=ADUHL[j+1];
```

```
            ADUHL[j+1]=temp;
          }
        }
      }
    }


void printUHL()
{
    for(int i=0;i<cntUHL;i++)
    {
       fprintf(fp,"%u (%d)\n", UHL[i].ptr,UHL[i].key);
    }
}

void printtree(node * t)
{
                        static int position=0;
                        int i, *k, n,j;
                        int z;
                        if (t != NULL)
                        {
                                position += 6;
                                fprintf(fp,"%*s", position, "");
                                for (i=0; i<t->cnt; i++)
                                        fprintf(fp," %d(%s.%c) ", t->ht[i].key,t->ht[i].vahost.vand,t->ht[i].vahost.vaht);
                                fprintf(fp,"\n");
                                for (i=0; i<=t->cnt; i++)
                                        printtree(t->ptr[i]);
                                position -= 6;
                        }
}

void printhost()
{
        int i,j,x;
        node *tmp=root;
        for(j=0;j<M;j++)
        {
                cout<<"\n"<<tmp->ht[j].key<<" -> ";
                for(i=0;i<=M;i++)
                {
                        if( (tmp->ht[j]).lnk[i] !=NULL)
                                cout<<((tmp->ht[j]).lnk[i])->key <<" ";
                        else
                                cout<<" NULL ";
                }
        }
cout<<"\n";

}

void printCentralABook()
{
    int i;
    for(i=0;i<cntCABook;i++)
    {
      fprintf(fp,"%d)- %s.%c : %d\n",i+1,CentralABook[i].vaddrnd,CentralABook[i].vaddrht,CentralABook[i].key);
    }
}

void error(char *str)
{
        printf("\nError: %s\n", str);
        exit(1);
}

void DelLocalAddrBookUpdate(node **p)
{
    for(int i=0;i<M+1;i++)
    {
     if((*p)->ptr[i] != NULL)
      {
        for(int j=0;j<M;j++)
```

```
        {
          for(int k=0;k<MM;k++)
          {
            if( !(strcmp((*p)->ptr[i]->ht[j].vahost.vand,(*p)->abook[k].vaddrnd)) && (*p)->ptr[i]->ht[j].vahost.vaht == (*p)->abook[k].vaddrht)
             {
              (*p)->abook[k].key=(*p)->ptr[i]->ht[j].key;
             }
          }
        }
      }

    }
}



node * DelBinSearch(int key,node **nd,int *k,int *pos)
{
    int i;
    node * temp;
    *pos=*k;
    for(i=0;(*nd)->ht[i].key<key && i < (*nd)->cnt;i++);
      *k=i;
    if((*nd)->ht[i].key == key)
                            return (*nd);
    else
       return (DelBinSearch(key,&(*nd)->ptr[i],k,pos));
}

int foundLeft(node *nd,int k)
{
   int i;
   for(i=k;i>=0;i--)
   {
    if(nd->ptr[i] != NULL)
    return i;
   }
   return -1;
}

int foundRight(node *nd,int k)
{
   int i;
   for(i=k+1;i<=nd->cnt;i++)
   {
    if(nd->ptr[i] != NULL)
    return i;
   }
   return -1;
}

void DelHost(node **nd,int pos,int k,int key)
{
    int loc1=-2,loc2=-2,i;
    loc1=foundLeft(*nd,k);
    loc2=foundRight(*nd,k);
    if(loc1 != -1)
    {
      for(i=k;i>loc1;i--)
      {
        (*nd)->ht[i].key=(*nd)->ht[i-1].key;
       }

       (*nd)->ht[loc1].key=(*nd)->ptr[loc1]->ht[(*nd)->ptr[loc1]->cnt-1].key;
        if(*nd)!=root)
        {
        DelLocalAddrBookUpdate( &(*nd)->par);
        }
       DelHost(&(*nd)->ptr[loc1],loc1,(*nd)->cnt-1,(*nd)->ptr[loc1]->ht[(*nd)->ptr[loc1]->cnt -1].key);
    }
    else if (loc2 != -1)
    {
      for(i=k;i<loc2;i++)
```

```
        {
          (*nd)->ht[i].key=(*nd)->ht[i+1].key;
        }

        (*nd)->ht[loc2-1].key=(*nd)->ptr[loc2]->ht[0].key;
        if((*nd)!=root)
        {
        DelLocalAddrBookUpdate( &(*nd)->par);
        }
        DelHost(&(*nd)->ptr[loc2],loc2,0,(*nd)->ptr[loc2]->ht[0].key);
      }
      else
      {
         for(i=k;i<((*nd)->cnt)-1;i++)
         {
           (*nd)->ht[i].key=(*nd)->ht[i+1].key;
            if((*nd)!=root)
            {
              DelLocalAddrBookUpdate( &(*nd)->par);
            }
         }
         (*nd)->ht[((*nd)->cnt)-1].key=-1;
         if((*nd)!=NULL)
         {
          DelLocalAddrBookUpdate( &(*nd)->par);
         }
         ((*nd)->cnt)--;
         if((*nd)->cnt ==0 && (*nd) == root)
         {
          root=NULL;
         }
         else if((*nd)->cnt ==0 && (*nd) != root)
         {
             (*nd)->par->ptr[pos]=NULL;
         }

          updateCentralABook(root);
          sortCentralABook();

      }

 }


int lookupchild(node *nd,host *ht,int key,int **count)
{
   int i,j,k,u,v,diff;

   if(ht!=NULL && nd !=NULL && ht->key != -1)
   {
   k=binsearch(key, nd->ht, nd->cnt);
   for(i=0;i<M;i++)
   {
     if(nd->ht[i].key==key)
     {
      return 1;
     }
   }
   if(per!=100)
   {
      for(int r=0;r<MM ;r++)
      {
        if(nd->abook[r].key==key)
        return 1;
      }
    }

     (**count)++;
     return(lookupchild(nd->ptr[k],ht->lnk[k],key,&(*count)));
   }
   else
     return 0;
}

int lookup(node *nd,int key,int *count,int cntup)
```

```
{
   int u,v,diff,flag=0;
   while(cntup>0)
   {
    nd=nd->par;
    cntup--;
   }

   if(nd!=NULL )
   {

   for(int k=0;k<nd->cnt;k++)
   {
     if(nd->ht[k].key==key) return 1;
    }

   int k=binsearch(key, nd->ht, nd->cnt);
   int i,sum=0;
              flag=1;
   if(per!=100)
   {
     for(int r=0;r<MM ;r++)
      {
       if(nd->abook[r].key==key)
       return 1;
      }
   }

   if(nd->ht[k].key==key && flag==1)
   {
    return 1;
   }
   for(i=0;i<LK && flag==1 ;i++)
   {
     (*count)++;
     int dd=lookupchild(nd->ptr[k],nd->ht[i].lnk[k],key,&count);
     if(dd==1)
     {
     sum=sum+dd;
     break;
     }
    }

    return sum;
    }
    else
    return 0;
}


int lookupFast(node *nd,int key,int *count)
{
   int u,v,diff,flag=0;

   if(nd!=NULL )
   {
   int k=binsearch(key, nd->ht, nd->cnt);
   int i,sum=0;
   for(int j=0;j<M;j++)
   {
    if(nd->ht[k].lnk[j]!=NULL)
     flag=1;
    }
   if(per>=90)
   {
      if(nd->ht[k].key==key && flag==1)
      {
       return 1;
      }
      else
      return 0;
   }
   else
   {
      int check=0;
```

```
        for(int r=0;r<MM ;r++)
        {
        if(nd->abook[r].key==key)
        return 1;
        else
        check++;
        }
        if(check==MM)
        return 0;
    }
   }
     else
     return 0;
}

void downHost(int key)
{
    node *nd;
    int k=-1,pos=-1;
    nd=DelBinSearch(key,&root,&k,&pos);
  for(int i=0;i<=M;i++)
    {
    nd->ht[k].lnk[i]=NULL;
    }

}

int convert(char *s)
{
                int sum=0,i;
                for(i=0;s[i]!='\0';i++)
                {
                                sum=sum*10+(s[i]-48);
                }
                return sum;
}

void genRandom()
{
  int i,n,x,j,flag=0,y,z;
  for(i=0;i<H;i++) dHost[i]=-1;
  n=cntCABook*per/100;
  dcnt=0;
  i=0;
  srand(time(0));
  for(i=0;i<n;)
  {
    flag=0;
     x=rand()%cntCABook;
    for(j=0;j<dcnt;j++)
    {
     if(dHost[j]==CentralABook[x].key) { flag=1; }

    }
    if(flag==0)
    {
     dHost[dcnt]=CentralABook[x].key;
     fprintf(fp1,"\n Key failed at %d : %d ",x,CentralABook[x].key);
     downHost(CentralABook[x].key);
     dcnt++;
     i++;
    }
  }
}

int BSearchADUHL(int x)
{
        int l=0,m,i,u=cntADUHL-1;
        i=-1;

        while(u>=l)
        {
          m=(u+l)/2;
          if(x==ADUHL[m].key)
          {
```

```
          i=m;
          break;
        }
        else
        {
          if(x>ADUHL[m].key)
            l=m+1;
          else
            u=m-1;
        }

      }
 return (i);
}



void updateADUHL()
{
    int i,j,pos;
    for(i=0;i<dcnt;i++)
    {
     pos = BSearchADUHL(dHost[i]);
     for(j=pos;j<cntADUHL-1;j++)
     {
       ADUHL[j].key=ADUHL[j+1].key;
        ADUHL[j].ptr=ADUHL[j+1].ptr;
     }
     cntADUHL--;
    }
}

void printADUHL()
{
        fprintf(fp1,"\n\n ----------ADUHL----------\n" );
    for(int i=0;i<cntADUHL;i++)
    {
        fprintf(fp1,"\n %u : %d ",ADUHL[i].ptr,ADUHL[i].key);
    }
}

void genQuery()
{
 int i,n,x,j,flag=0,y,z,flag1=0;
 for(i=0;i<H;i++)
 {
     qList[i].src=-1;
     qList[i].dest=-1;
 }

 n=cntADUHL*perQ/100;  //UHL is used host list which stores ptr and key
 cntQList=0;
 i=0;
 srand(time(0));
for(i=0;i<n;)
 {
  flag=0;

   x=rand()%cntADUHL;

  for(j=0;j<cntQList;j++)
  {
   if(qList[j].src==ADUHL[x].key ) { continue; }

  }
   qList[i].src=ADUHL[x].key;
  again:
  flag1=0;

   y=rand()%cntADUHL;

  for(j=0;j<cntQList;j++)
  {
   if(qList[j].src==ADUHL[y].key || ADUHL[y].key == -1 ) { goto again; }
```

```
    }
     qList[i].dest=ADUHL[y].key;
     cntQList++;
     i++;
  }

}

void printQList()
{
  int i;
  fprintf(fp1,"\n---------- Query List---------- \n");
  for(i=0;i<cntQList;i++)
  {
      fprintf(fp1,"<SRC: %d # DEST: %d> \n",qList[i].src,qList[i].dest);
  }
}


int binsearchQlist(int x)
{

      int l=0,m,i,u=cntUHL-1;
      i=-1;

      while(u>=l)
      {
        m=(u+l)/2;
        if(x==UHL[m].key)
        {
          i=m;
          break;
        }
        else
        {
           if(x>UHL[m].key)
             l=m+1;
           else
             u=m-1;
         }

      }
  return (i);

}

int xorr(char svid[],char dvid[])
{
   int ls,ld,len,i,j,res=0,diff;
   char svid1[10],dvid1[10];
   ls=strlen(svid);
   ld=strlen(dvid);
   strcpy(svid1,svid);
   strcpy(dvid1,dvid);


   if(ls<ld)
   {
    i=0;
    dvid1[ls]='\0';
    while(strcmp(svid1,dvid1)!=0)
    {
      i++;
      res++;
      dvid1[ls-i]='\0';
      svid1[ls-i]='\0';
    }

   }
    else if(ld<ls)
   {
    i=0;
    svid1[ld]='\0';
    while(strcmp(svid1,dvid1)!=0)
    {
```

```
        i++;
        res++;
        dvid1[ld-i]='\0';
        svid1[ld-i]='\0';
      }
      diff=ls-ld;
      res=res+diff;

    }else if(ld==ls)
     {
         i=0;

         while(strcmp(svid1,dvid1)!=0)
         {
            i++;
            res++;
            dvid1[ls-i]='\0';
            svid1[ls-i]='\0';
         }
     }
   return res;
}

main(int argc, char *argv[])
{
              int i,j,code=0,x,k=-1,pos=-1,sum,dn,z,count=-1,num;
              int noOfHosts=convert(argv[4]);
              float pathLen;
              root=NULL;
              node *dummy=NULL;
              node *nd=NULL,*tempnd=NULL;
              char ch, treefilnam[51], *inpfilnam;
              char svid[S],dvid[S];

              FILE *fpinp;
              ch='y';
              if (toupper(ch) == 'Y')
              {
                        inpfilnam=argv[1];
                        if ((fpinp = fopen(inpfilnam, "r")) == NULL)
                                  error("file not available");
                        while (fscanf(fpinp, "%d", &x) == 1)
                        {
                                  insert(x,&root,&dummy,0);
                        }
                        fclose(fpinp);
              }
              cout<<"\n";
              printtree(root);
              code=1;
              ch='L';
              switch (ch)
              {

                        case 'I':
                                                            insert(x,&root,&dummy,0);
                                                            break;

                        case 'D':
                                                            nd=DelBinSearch(x,&root,&k,&pos);
                                                            DelHost(&nd,pos,k,x);
                                                            break;

                        case 'L':
                                                            per=convert(argv[2]);
                                                            perQ=convert(argv[3]);
                                                            fillUHL(root);
                                                            sortUHL();
                                                            sortADUHL();
                                                            genRandom();
                                                            updateADUHL();
                                                            genQuery();
                                                            printADUHL();
                                                            printQList();
                                                            num=0;
```

```
                                        pathLen=0;
                                        int count1,sum1=0,pathLen1=0;
                                        float num1=0;
                                        for(int q=0;q<cntQList;q++)
                                        {
                                                count1=0;
                                                int dd=qList[q].dest;
                                                int cntup=0;
                                                sum1=lookup(root,dd,&count1,cntup);
                                                if(sum1 > 0)
                                                {
                                                count1++;
                                                pathLen1=pathLen1+count1;
                                                fprintf(fp1,"\n key  %d Top Found,Hops=%d  \n",dd,count1);
                                                num1++;
                                                }
                                                else
                                                fprintf(fp1,"\n key  %d  Not Found,Hops=%d \n",dd,count1);
                                        }
                                        fprintf(fp1,"\n PATHLEN=%f NUM=%d  \n",pathLen1,num1);

                                        float res1=num1*100.0/(cntQList);
                                        for(int q=0;q<cntQList;q++)
                                        {
                                                count=0;
                                                int tt=qList[q].src;
                                                int dd=qList[q].dest;
                                                int d=binsearchQlist(tt);
                                                int locsrc=ABSearch(tt);
                                                int locdest=ABSearch(dd);
                                                strcpy(svid,CentralABook[locsrc].vaddrnd);
                                                strcpy(dvid,CentralABook[locdest].vaddrnd);
                                                int cntup=xorr(svid,dvid);
                                                int tempcnt=cntup;
                                                tempnd=UHL[d].ptr;
                                                sum=lookup(tempnd,dd,&count,cntup);
                                                if(sum > 0)
                                                {
                                                        count++;
                                                        pathLen=pathLen+count;
                                                        fprintf(fp1,"\n < %s(tt) - %s(dd) = %d >   key  %d
Found,Hops=%d  \n",svid,dvid,cntup,dd,count);
                                                        num++;
                                                }
                                                else
                                                        fprintf(fp1,"\n key  %d  Not Found,Hops=%d
\n",dd,count);
                                        }

                                        float res=num*100.0/(cntQList);//-(cntCABook*per/100) );

                                        if(num !=0 && num1 !=0)
                                        {
                                                fprintf(fp2,"Lookup Sucess=%f, \n Avg. Path
Length=%f\nLookup Sucess=%f,\nAvg. Path Length=%f\n",res,pathLen/num,res1,pathLen1/num1);

        fprintf(fp3,"%d\t%d\t%d\t%d\t%d\t%f\t%f\t%f\t%f\t%f\n",cntCABook,M+1,LK,per,perQ,res,pathLen/num,res1,pathLe
n1/num1,log(noOfHosts)/log(2));
                                        }
                                        else
                                                fprintf(fp2,"%f   %f\n No Path Length ",res,res1);
                                        break;
        }

        fprintf(fp,"\n\t\tTree Structure with Local Address Book\n");
        printtree(root);
        fprintf(fp,"\n\t\tCentral Address Book\n");
        printCentralABook();
        printUHL();
        fclose(fp);
        fclose(fp1);
        fclose(fp2);
        fclose(fp3);

}
```