A RELIABLE AND SCALABLE ADDRESSING SCHEME FOR DATA DISTRIBUTION

By

SRAVANTHI PERURU

Bachelor of Technology In Computer Science and Engineering

Jawaharlal Nehru Technological University

Hyderabad, India

2008

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2010

A RELIABLE AND SCALABLE ADDRESSING

SCHEME FOR DATA DISTRIBUTION

Thesis Approved:

Dr. Subhash Kak

Thesis Advisor

Dr. Johnson Thomas

Dr. Michel Toulouse

Dr. Mark E. Payton

Dean of the Graduate College

ACKNOWLEDGMENTS

I owe a great deal of gratitude and appreciation to my major advisor Dr. Subhash Kak for his guidance, motivation, dedication and valuable instruction during my thesis work.

I wish to thank my other graduate committee members, Dr. Johnson Thomas and Dr. Michel Toulouse. Their time and efforts are greatly appreciated.

I would like to thank many other people who taught me the basics of computer programming and its concepts: my undergraduate teachers and my graduate teachers,
Dr. Samadzadeh, Dr. Nophill Park, Dr. K.M.George. I would also like to thank our systems manager Mr. Terry Wright for helping me during the time of system failures.

I would like to thank my friends Pranava Kuchu, Renuka Kandregula for supporting me during the difficult times and for their emotional support and caring they provided. I wish to thank my entire family members for their loving support.

Lastly and most importantly, I would like to thank my parents for their extended support, both financially and morally, which made me come to the United States and pursue my graduate degree. It was their examples of hard work over years that gave me the inspiration and motivation to complete my graduate studies. I will be forever indebted to them for this.

.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

Data centers store, manage, process, and exchange digital data and information. The client/server model is based on a fixed number of servers for many clients. As a result, the resources available in this model are only likely to decrease, as the load of the network increases. It has a single point of failure because the number of servers is static, and tends to be un-scalable. Conversely in a distributed model (where the data may be distributed over many systems), the more the number of nodes in the network, the more capacity and load the system can handle. The resources and the load are distributed among all the nodes to reduce the overall load of the system. In order to achieve this, the distributed data systems must be highly scalable and capable of modifying and adapting its topology as nodes are entering or leaving the system.

Peer to peer systems have distributed data storage architecture. They are loosely organized systems without any centralized control or any hierarchical structure. Data centers and peer to peer systems share a common communication model in which each node acts as both client and server. This contributes a system that has no centralized data and has its entire load almost equally distributed over the network ensuring the load balancing in the network.

1.1 BACKGROUND AND MOTIVATION

Data centers are being increasingly used due to the increasing popularity of cloud computing. Such usage requires an efficient and reliable addressing scheme to store and communicate the data. This very need for an addressing scheme motivated me to pursue my thesis. There are some theoretically ideal protocols in peer to peer networks. But a simple theoretical model does not necessarily help in achieving efficient communication.

Chord [1] is a theory-inspired protocol which belongs to the family of peer to peer protocols. But it is not used in real time applications because of its enormous overhead. Although the Chord protocol will resolve all the lookups via O(log N) messages to other nodes, there can be a more efficient lookup service which can take less than O(log N) messages [1]. Also the stabilization overhead in Chord prevents it from being used in real time.

Gnutella [2] is another peer to peer protocol, although widely used; it has its own drawbacks. In case of Chord, data needs to be moved back and forth among the nodes continuously as nodes join and leave the network, making it to be unacceptable for the practical implementation. The main drawback of Gnutella protocol is its heavy request traffic. The passing of request messages in Gnutella often leads to network congestion and slow response making the quality of service poor, since the response to the queries is delayed. A study showed that most of the traffic in Gnutella systems is mainly due to messages for establishing initial connections and queries [3].

In view of these drawbacks, we propose a new protocol called CHORELLA that inherits the features of both Chord and Gnutella, which explains its name.

1.2 RESEARCH GOALS AND APPROACH

Distributed hash tables (DHTs) [4] are a class of decentralized distributed systems that provide a lookup service similar to a hash table. Chord is one of the four original distributed hash table protocols, (along with CAN, Tapestry, and Pastry [5]), and it was developed in 2001 as the bottom layer of a DHT, for routing and locating peers in the overlay network. It provides support for just one operation and given a key, it maps the key onto a node [10]. The Chord protocol is designed to efficiently locate the node where the key is stored, but it remains a theoretical model.

Gnutella[2] is another fully distributed alternative to the centralized systems. It uses a flooding query model. All the queries are flooded to all nodes until the data is located. It limits the query to a maximum number of hops (used like a time-to-live) to prevent infinite loops. Ripeanu reported that the traffic generated in Gnutella consists of approximately 92% Query messages, 8% Ping messages and hence other messages constitute less than 1% of the traffic [7].

The above problems call for an efficient and a reliable addressing scheme. The core problem confronting distributed data storage and peer to peer applications is to efficiently locate the node where the requested data is present [8]. The goal of this thesis is to find an addressing scheme which goes beyond the theoretical model. CHORELLA, the new addressing scheme is built upon the ideas of Chord and Gnutella and it finds the location of the requested data in the network.

CHORELLA avoids the heavy network traffic and minimizes the number of lookup messages. It is similar to Chord in the way it organizes its nodes on an identifier space. We take into consideration that there can be a maximum of 2^{20} (this number is an arbitrary choice based on an expectation of the number of nodes in a large network) nodes that can be a part of the network. CHORELLA identifies a 2^{20} bit identifier space to place the nodes on the identifier circle. It is assumed that all these nodes are alive when the network is established initially. Each node's IP address is mapped to a 20 bit address using one to one mapping. A Manager node, which sits on the top of the overlay is responsible for assigning these 20 bit nodeIDs to the nodes. A list of

all these pairs (the nodeID and its corresponding IP address) of all the nodes present in the network forms a mapping table which is maintained by the Manager node.

Each file is placed in the network in the following manner: The file name is hashed to 160 bit hash value using SHA-1 hashing algorithm. Least significant 20 bits of the 160 bits are extracted. This file will now be stored at a node whose nodeID is same these least significant 20 bits. When a client requests a particular file, it first hashes the filename and gets the 160 bits hash value. From that 160 bit hash value, it extracts the least significant 20 bits. The client (node that is posed with query) now knows that, the requested file should be present with a node whose nodeID is equal to these least significant 20 bits. The client will look up for the IP address corresponding to the nodeID in the mapping table. The client will then request that node for the file. File mirrors are also stored at the other locations in the network, making the protocol more reliable even in case of node failures.

CHAPTER II

REVIEW OF LITERATURE

2.1 DISTRIBUTED HASH TABLES

A distributed hash table (DHT) is a reliable, scalable, wide area data storage system that frees programmers from many of the complications of building a distributed system. DHTs store blocks of data on hundreds or thousands of machines connected to the Internet, replicate the data for reliability, and quickly locate data despite running over high latency, wide area links. The DHT addresses problems of locating data and replicating it for reliability, which are common to many distributed systems, without additional work by the application.

The DHT provides a generic interface, which makes it easy for a wide variety of applications to adopt DHTs as a storage substrate: PUT: stores data in the system under a key; GET: retrieves the data. Distributed hash tables fill a gap in the design space of storage systems. DHTs occupy middle place between small systems with strong guarantees on the service they provide (such as distributed file systems) and large unorganized, best effort systems (such as the World Wide Web or file sharing systems). DHTs are able to operate over a large and previously unoccupied area of the design space; the existence of a single, practical system that operates throughout this regime makes it easier to write new distributed applications.

Inspired by small, LAN based systems (which we will call transparent distributed systems), DHTs provide probabilistic guarantees on the success of a GET or PUT operation. At the same time DHTs can also operate on the large scale systems designed to run on the wide

area (which we will call Internet systems). Transparent distributed systems, such as Harp [18] and DDS [19], combine the resources of a number of systems, while trying to hide any evidence from the user which reveals that the system is distributed.

In order to obtain this clarity, these systems provide strong guarantees on the results of operations. For example DDS, guarantees that a GET operation will always see the results of the last PUT operation. This guarantee is provided because transparent distributed systems are designed to run on a small number of machines connected to a local area network and it is assumed to have a high bandwidth, low latency, reliable interconnect. DDS could not make the same guarantees on the wide area because in its case the above assumptions are violated and assumes that the network is never partitioned. The techniques used by these systems in order to provide consistency are expensive to run over unreliable, high latency links.

The other research ancestors of DHTs, Internet systems along with transparent distributed systems, have the goal of aggregating the resources of multiple machines. However, Internet systems mainly focus on maximizing scale but not on the transparency. These systems are designed to run on a large number of geographically distributed nodes on the wide area network. The scalability of these systems comes at a cost of some fixed guarantees: a file sharing service like Gnutella offers a keyword search interface but it may fail to return the location of a file even if the file exists in the system [20].

DHTs are developed as a result of the need for a system which is scalable with the Internet systems (large number of machines, large amount of data, large inter-node latency). DHTs overlook some of the of the guarantees offered by the transparent distributed systems and

cannot be scaled as well as the loosely organized systems like the web. But the system will run in a wide range of environments including the machine room clusters to a collection of cable modems, and provide a generic interface that is a suitable for a large number of distributed storage applications that can operate with consistency.

The lack of strong consistency has not limited the usefulness of DHTs. Scenarios that requires strong consistency, such as write sharing, are rare and applications designed to run on DHTs can be designed to minimize the amount of mutable data in the system and eliminate the possibility of multiple writers [21]. Because DHTs work in a variety of deployment scenarios, they provide the possibility of a near universal storage infrastructure. Any wide area distributed system must be able to sustain with the challenges of node scale, network delay, and node failure. DHTs implement mechanisms such as replication and efficient location algorithms.

The DHT interface may prove to be the storage equivalent to the IP abstraction for networks. DHTs are widely used because they can be adopted even in small systems where the systems need not maintain full membership information in order to route the queries. DHTs.

2.2 CHORD PROTOCOL

Chord presents a new way out to the problem of efficient location of the node. It uses routed queries to locate a key with a small number of hops, which remains small even if the system contains a large number of nodes. Essentially, It supports just one operation: given a key, it maps the key onto a node [1]. Data localization can be implemented by associating each key with a data item.

2.2.1 OVERVIEW OF THE CHORD PROTOCOL

Distributed computation of a hash function that is provided by the Chord protocol, uses consistent hashing [2], [3], [9], which balances the load. The same number of keys is received by all nodes. In a N node network, each node maintains information only about O(log N) other nodes, and a lookup requires O(log N) messages. When an Nth node joins (or leaves) the network, an O(1/N) fraction of the keys are moved to a different location, which maintains a balanced load. A join or leave requires $O(\log^2 N)$ messages to update the routing information[1].

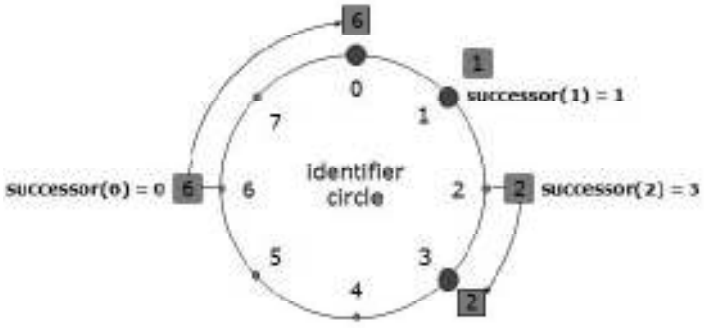


Figure 2.1 Identifier circle with m=3.

The Chord circle shown in figure contains three nodes: 0, 1 and 3. The successor of node 2 is node 3.

Chord uses the routed queries to locate a key, using identifiers to refer to nodes and keys “to let nodes enter and leave the network with minimal disruption" [1]. Each node and key is

assigned an m-bit identifier using SHA-1[11] as a base function. NodeID (identifier) is obtained by hashing the IP address of the node, which is a unique id assigned for each node. The 160 bits obtained by hashing the IP address will form its corresponding nodeID. KeyID (identifier) is obtained by hashing the filename.

Length of the identifier should be large enough so that the probability of two keyIDs hashing to the same identifier is negligible. In consistent hashing keys are assigned in the following way: Identifiers are ordered on an identifier circle modulo 2^m. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. The node is called the successor node of key k, denoted by successor(k). Every node n maintains a routing table with up to m entries (where m is the number of bits of the identifiers) called finger table. The ith entry in the table at node n contains the identity of the first node S that succeeds n by at least 2^i-1 on the identifier circle. This node S is called the i^{th} finger of node n.

2.2.2 KEY LOOKUPS

The principle function of Chord protocol is the key look-up function. For a better knowledge an insight is provided into the simple key lookup and scalable key lookup.

2.2.2.1 SIMPLE KEY LOCATION

When a node receives a query, it responds with a reply if it contains the requested file else, it will forward the query to its successor. The successor in turn will search for the existence of that file with it. If it finds it, then the successor, will simply respond to the request. In this way each query passes through all the successors, until it reaches the node where the file is located.

In the worst case, the query needs to be forwarded N times in a circle with N nodes, so the cost of a lookup is linear with the number of nodes. In systems with a large number of nodes, lookups would be too slow. Therefore, Chord uses a scalable key location function which will provide more efficient lookups.

2.2.2.2 SCALABLE KEY LOCATION

Routing information is stored to accelerate lookups so as to provide a better and efficient lookups. Every node n maintains a routing table with up to m entries (where m the number of bits of the identifiers) called the finger table.

The i^{th} entry in the table at node n contains the identity of the first node S that succeeds n by at least 2^{i-1} on the identifier circle. This node S is called the i^{th} finger of node n.

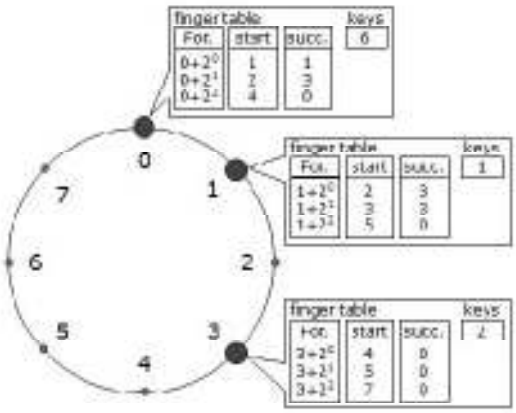


Figure 2.2: Shows the routing table for nodes. Finger table entries are calculated by the formula

$$finger[i] = successor\ (n+2^{i-1})$$

Each Chord needs “routing” information about only a few other nodes. Because the routing table is distributed, a node resolves the hash function by communicating with few other nodes.

Important characteristics of Scalable key location:

- Each node stores information about a small number of nodes (m).
- Each node knows more about nodes closely following it than about nodes farer away.
- A finger table generally does not contain enough information to directly determine the successor of an arbitrary key k. A node has to contact other nodes in order to resolve the hash table.
- In the steady state, in a N-node system, each node maintains information only about O(log N) other nodes, and resolves all the lookups via O(log N) messages to other nodes.

When a node is asked to find a particular key, it will first find the highest predecessor of this key in its routing table and forward the key to that node. This procedure will, iteratively determine the node responsible for the key. Since each node has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target identifier. The lookup time is O(log N). Figure 4 shows the scalable key location in a Chord ring.

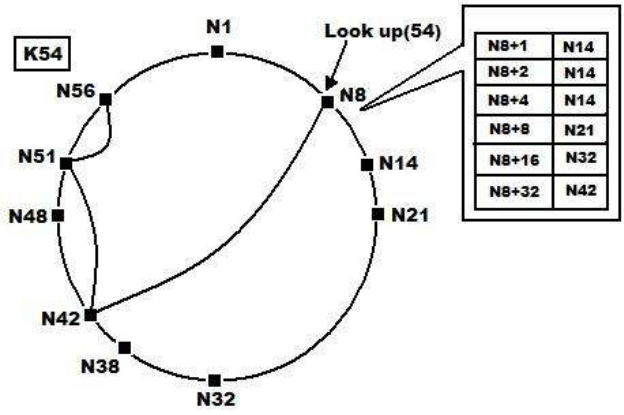


Figure 2.3 Scalable Key location

2.2.5 NODE JOINS

When a node joins the system, the successor pointers of some nodes will have to change. It is important that the successor pointers are up to date at any time else the correctness of lookups is not guaranteed. The Chord protocol uses a stabilization protocol running periodically in the background to update the successor pointers and the entries in the finger table.

The following example shows how any node joins a Chord ring.

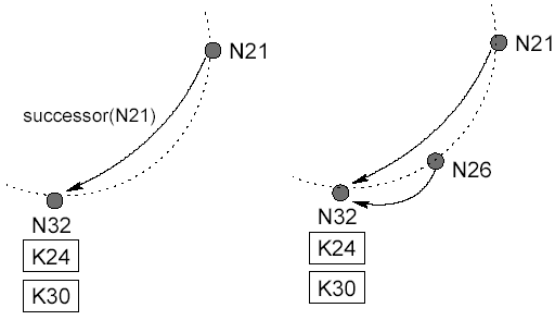


Figure 2.4 (a) N26 joins the Chord ring. It sets its successor pointer to N32 and notifies N32, so N32 sets its predecessor pointer to N26.

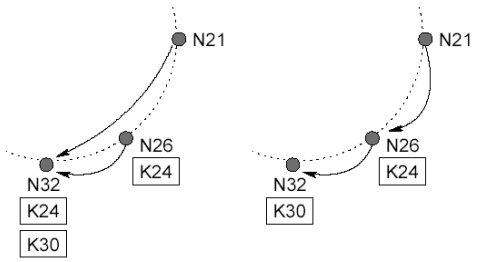


Figure 2.4 (b) N26 copies K24. Next time N21 runs stabilize(), it detects N26 as its new successor. N21 changes its successor pointer to N26 and notifies N26, so N26 sets its predecessor pointer to N21.

2.2.6 STABILIZATION

Stabilization protocol is used to keep nodes successor pointers up to date, which is sufficient to guarantee correctness of lookups. Those successor pointers are then used to verify and correct finger table entries, which allow these lookups to be fast as well as correct.

2.2.7 NODE FAILURE

The correctness of the Chord protocol entirely depends on the fact that each node knows its successor. When a node fails, it is possible that existing node does not know its new successor, and that may have no chance to learn about it. Figure 2.5 demonstrates an example.

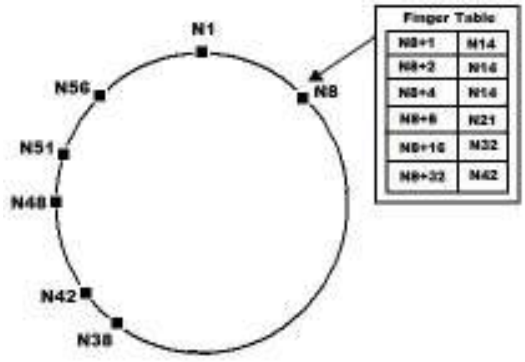


Figure 2.5 Node Failure

When nodes N14, N21 and N32 fail simultaneously, N8 now has no chance to know about its new successor N38, because it does not show up in the finger table of N8.

In order to deal with this situation, each node maintains a successor list of size r, containing the node's first r successors. When the successor node does not respond, the node simply contacts the next node on its successor list. Assuming that each node fails with a probability p, the probability that every node on the successor list fails is p^r. Increasing r makes the system more robust. With this parameter tuning, any degree of robustness can be achieved.

2.2.8 QUANTITATIVE ANALYSIS

- For any set of N nodes and K keys, with high probability:
    1. Each node is responsible for at most (1+e)K/N keys
    2. When an (N+1)st node joins or leaves the network, responsibility for O(K/N) keys changes hands (and only to or from the joining or leaving node) [1].
- If we take a stable network with N nodes, and another set of up to N nodes joins the network with no finger pointers (but with correct successor pointers), then lookups will still take O(log N) time with high probability.

- If we use a successor list of length r=O(log N) in a network that is initially stable, and then every node fails with probability 1/2, then with high probability find_successor returns the closest living successor to the query key.
- If we use a successor list of length r=O(log N)in a network that is initially stable, and then every node fails with probability 1/2, then the expected time to execute find successor in the failed network is O(log N) [1].

2.3 GNUTELLA

In Gnutella [3], servants are the nodes that perform tasks normally associated with both clients and servers. They provide client side interfaces through which users can issue queries and view search results, while at the same time they also accepts queries from other servants, check for matches against their local data set, and respond with applicable results.

Gnutella consists of a set of descriptors which are used for communicating the data between servants and a set of rules governing the inter-servant exchange of descriptors [11] which are given below:

DESCRIPTOR	DESCRIPTION
Ping	A ping message is used to discover hosts on the network. According to the expectation a servant that receives a ping message is responded with one or more Pong descriptors
Pong	It is a response to a Ping message. This message will include the address of a connected Gnutella servant and also the information about the amount of data it is making available to the network.
Query	This is the primary mechanism for searching the distributed network. A servant that receives a Query descriptor will respond with a QueryHit if a match is found against its local data set
QueryHit	It is response received for a query. This descriptor provides the recipient with sufficient information to acquire the data matching its corresponding Query.
Push	This will allow a firewalled servant to contribute file-based data to the network.

Table 2.1 Descriptors of Gnutella

A Gnutella servant connects itself to the network by establishing a connection with another servant which is currently a part of the network.

2.3.1 DESCRIPTORS AND THEIR ROLES IN DETAIL

Ping: A servant uses 'Ping' descriptors to search for the other servants in the network. A servant receiving a Ping message may respond with a Pong descriptor.

Pong: They are sent in response to an incoming Ping descriptor. More than one Pong messages may be sent in response to a single Ping descriptor.

Query: Query is a request message for a particular file in the network. A servant forwards this Query descriptor to all its neighbors. These servants will in turn forward this Query descriptor to their corresponding neighbors except to the originating server.

QueryHit: These descriptors are sent in response to an incoming Query descriptor. A servant will respond to the Query with a QueryHit if it contains data which strictly meets the Query search criteria.

Push: If there exists a servant in the network that sends the servant a QueryHit descriptor, (indicating that it has the data which meets the query search criteria sent by the originating servant), but doesn't support incoming connections, implying that the servant sending the QueryHit [12] is behind the firewall then the originating servant would send a Push descriptor. The servant that is the target of the Push request attempts to establish a new TCP/IP connection to the requesting servant upon the receipt of the Push descriptor.

2.3.2 ANALYSIS OF GNUTELLA MESSAGES

The subsection below describes some interesting results regarding the messages that were routed through a cluster - X. The sample that is analyzed exceeds 56 million messages.

Figure 2.6 presents the message breakdown by Message Type. Only a very few Push messages (i.e. 3; 000) were routed through the cluster - X and hence they are not presented in the pie chart. Figure 2.6 confirms that Gnutella has actually a huge communication overhead (i.e. ping/pong messages), since it is averagely 63% of all the messages. That is due to the fact that Gnutella connections are relatively unstable which leads peers in a endless effort of discovering new peers rather than to the fact that peers are joining and leaving the network at fast paces. Pong Stealing might be a reason why a peer is attracting more network load than the load it can really handle. Pong Stealing is when an intermediate node B is obtaining the IP address, from a Pong message which is routed through it. In this situation a host C which replied positively to a ping request of a host A might get a connection request from both A and B although its initial intention was to accept only 1 connection. This phenomenon might destabilize Cs current network connections since C is forced to; at least, abort the one of the two requested connections [24].

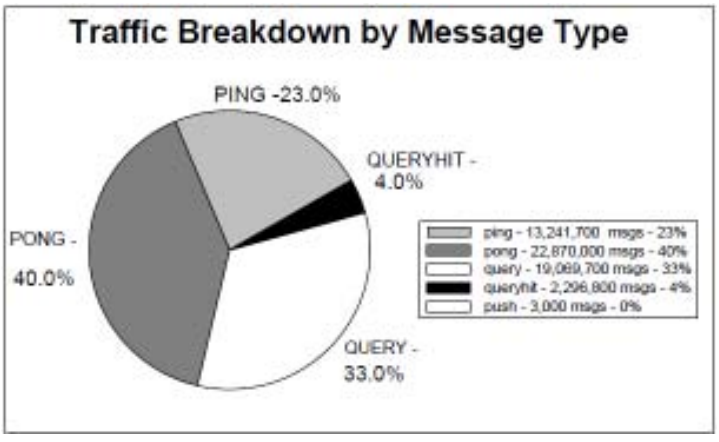


Figure 2.6 Messages Breakdown by Message Type.

Another reason for network instability might be that some "powerful" host A is routing many messages to another "weak" host "B" with a result that B is kept so busy that it is not able to handle its rest connections. if B is implemented appropriately (i.e. keep different queues for different connections) then of course such a thing won't occur but the point here is that this really depends on the implementation of each Gnutella client.

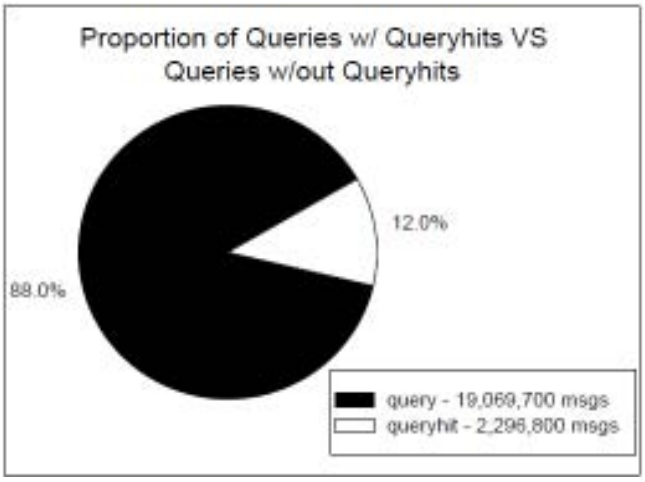


Figure 2.7 Proportions of Queries w/ QueryHits VS Queries w/out Queryhits.

Figure 2.7 presents the amount of queries routed through our cluster versus the amount of queries that were routed with queryhits. A similar measurement was obtained in [17] where the percentage of Queries with Queryhits varied between 10-12% [24].

2.3.3 COMMUNICATION IN GNUTELLA

A node that is posed with a request sends the query to all of its neighbour nodes, who will inturn send the query to all of their neighbour nodes and this query broadcasting continues until the query reaches a node that has a file which matches the query, or until a certain predefined maximal number of forwards is reached. If the destination node (node where the requested file is present) is reached, then that node sends back a reply to the query containing its address, the size of the file, speed of transfer, etc. The reply travels through the same path by which the query arrived, but in the reverse order back to the node that posed the query. In this manner each message is propogated to upto n^p other nodes where n is the number of neighbour nodes and *p* is the maximum number of forwards called “time to live” (TTL) of the query [15].

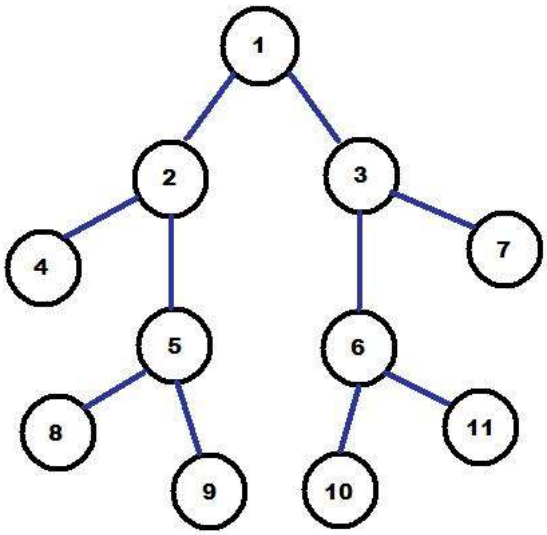


Figure 2.8 Communication between nodes in Gnutella when node 1 queries the network.

Figure 2.8 represents the communication that takes place in Gnutella. When node 1 recieves a query, it broadcasts the query to its neighbour nodes 2 and 3. 2 and 3 nodes in turn broadcast the request to their respective neighbours which are 4, 5 and 6, 7 respectively. Nodes 4, 5, 6, 7 in turn broadcast their requests to their neighbours. In this way the request traverses the network until it hits a nodes with the required response or until a maximal number of forwards is reached.

CHAPTER III

THE CHORELLA FRAMEWORK

INTRODUCTION

The prime goal behind designing CHORELLA is to provide an efficient way of addressing a node in the distributed data storage systems, to avoid the heavy network traffic and minimize the number of lookup messages. It is an overlay network that sits on the top of the physical network and supervises and controls the communication. CHORELLA shares some of its characteristics with Chord. It is similar in the way it organizes its nodes on the identifier space. We consider that there can be a maximum of 2^{20} nodes that can be a part of the network; CHORELLA identifies a 2^{20} bit identifier space to place the nodes on the identifier circle.

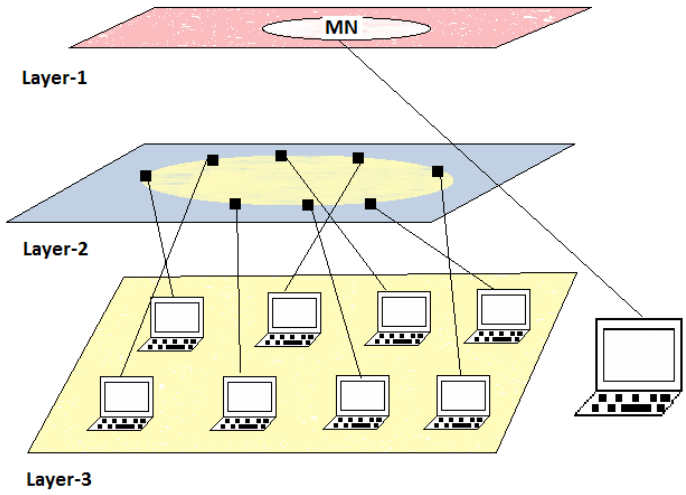


Figure 3.1 CHORELLA sitting on the top of physical Network.

3.1 MAPPING OF NODES

The traditional method of assigning nodeIDs using the hash function is not useful for CHORELLA. Assume the nodes were assigned the nodeIDs in the following manner:

- IP address of each node that comes in is hashed to 160 bits using SHA-1 hashing algorithm [13] [14].
- Least significant 20 bits of the 160 bits are extracted and these 20 bits form the nodeID of the nodes.

It leads to the problem of collision of the nodeIDs arises. That is more than one node has a same nodeID, although the nodeIDs are supposed to be unique. If more than one node has the same nodeID, then it would not be possible for a querying node to find a specific node that contains the requested data/file.

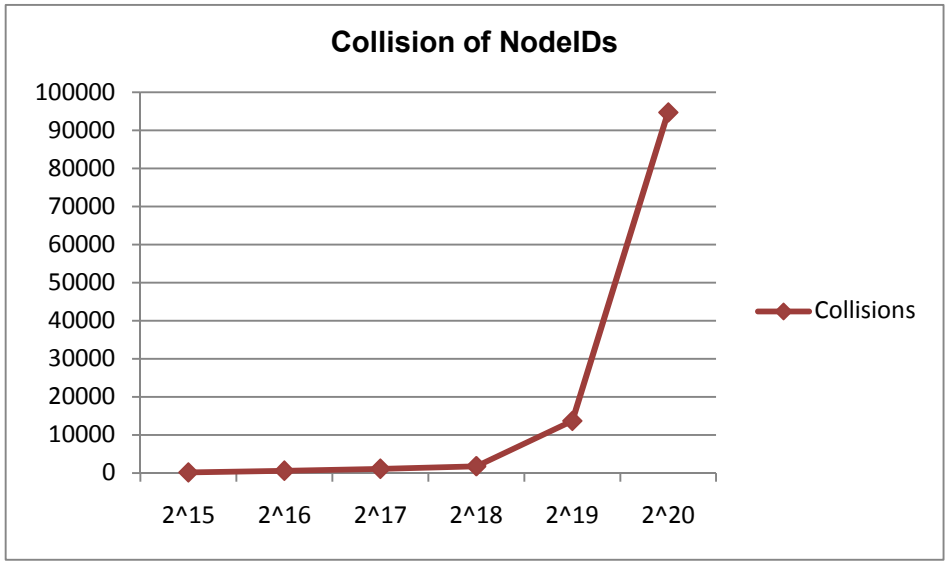


Figure 3.2 Plot of the number of collisions against the number of nodes in the network when the NodeIDs are assigned using the SHA-1 hash function.

Figure 3.2 shows that there is a fair chance for collisions when the nodeIDs are obtained by hashing the IP addresses using SHA-1 hash algorithm and then extracting the least significant 20 bits of those 160 bits and hence forming the nodeID

Every node that wants to be a part of network, must register itself with the Manager node. The Manager node is a supervising node taken to be always alive that sits on the top of network managing and supervising the communication among the nodes. The Manager node assigns to each node a 20 bit ID, and this assignment is a one to one mapping. A node is assigned a unique ID in the same order as it joins the network. Since ID space ranges from 0 to 2^{20} , there can be a maximum number of 2^{20} nodes in the network.

Let this 20 bit address assigned by the Manager node be the nodeID of the corresponding node.

- Place this node on an identifier circle of identifier space 2^{20}. Identifier circle can be thought of as a circle which represents values between 0 and 2^{20}-1
- The Manager node is responsible for storing and updating the mapping table. Mapping table contains a set of IP addresses and their corresponding 20 bit mapped value or the nodeID, of all the nodes present in the network.

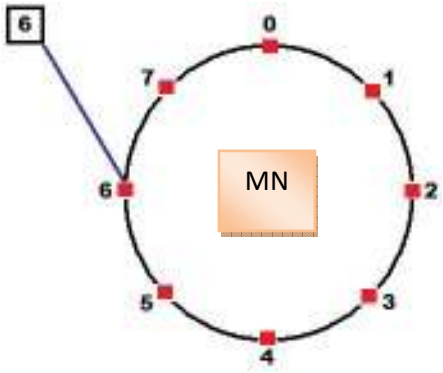


Figure 3.3 Identifier circle with an identifier space of 2^2. MN is the Manager node

Figure 3.3 shows how a node is placed on an identifier circle. When a node with nodeID 6 wants to join the network, it is placed at an identifier space 6 on the identifier circle.

Figure 3.1 shows how the CHORELLA looks like. The top most layer, Layer-1 has the Manager Node, the supervisor of the communication. The middle layer, Layer-2 has the CHORELLA, nodes which are mapped to the physical nodes on Layer-3.

3.2 FILE DISTRIBUTION

Any file that is to be placed on a node:

- Approaches any random node in the network. It is the responsibility of that node to hash the file key using the SHA-1[13] hashing algorithm.
- The node hashes it keys (Key of the file depends upon the application using it, CHORELLA doesn't define this) to a 160 bit hash value using SHA-1, a consistent hashing [6], [14] algorithm.
- Least significant 20 bits of the 160 bit hash value, together with 160 bit hash value and file key forms the KeyID.
- KeyID = [least significant 20 bits of the 160 bit hash value ; 160 bit hash value of key ; filekey]
- Though the 20 bit key hash of each file may not be unique but, the 160 bit hash value together with 20 bits and file name makes it unique.
- This file is placed at a node whose nodeID is equal to the keyID (least significant 20 bits of the 160 hash of the filekey).

IP Address	20 bit mapped value
139.78.67.161	00000000000000000000
74.195.252.115	00000000000000000001
148.76.90.87	00000000000000000010
32.7.165.117	00000000000000000011
19.248.167.1	00000000000000000100
27.68.69.178	00000000000000000101
169.178.197.1	00000000000000000110
147.248.167.87	00000000000000000111

Table 3.1 illustrates the mapping table which holds the IP addresses and their corresponding nodeIDs.

3.3 FILE SEARCH

When a client (one among 2^{20} nodes) requests for a file:

- The client itself converts the key associated with the file into a 160 bit hash value using SHA-1 algorithm.
- It extracts the least significant 20 bits of the 160 bit hash value and forms the keyID
- The client then obtains the IP address of the node whose nodeID is equal to the first value (20 bits) of the keyID searching from the mapping table. It searches for the IP address corresponding to the node whose nodeID is equal to the 20 bits in keyID.The client then requests that node for that particular file.

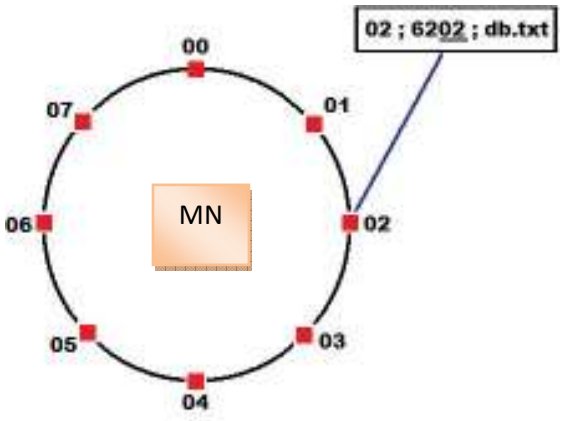


Figure 3.4 File search in CHORELLA

Figure 3.4 represents an example for the file search operation. When there is a query for file name db.txt (i.e., a file, named db.txt is requested by the client), the client will first convert the key of the file into an integer value of length 4 (In case of CHORELLA it is the 160 bit hash value) which is 6202. The least significant 2 digits of the 6202, i.e., 02 is extracted. The client now knows that the file should be present with the node whose nodeID is equal to 02. So the client now searches for the IP address corresponding to nodeID 02 from the mapping table and obtains it. After obtaining the IP address the client, directly sends the request to that IP address and obtains the file.

3.4 NODE FAILURE

Nodes may leave the network at their own will, without even informing any other nodes in the network. But if the nodes become unavailable, the data present with node is lost forever. If this happens then the data availability is not guaranteed. To ensure data availability in case of node failure, every file that comes in, is stored at more than one location. This replication ensures the data availability even in case of node failures making the protocol reliable. The mirrors are stored at various locations in the following manner.

- Each file on a node is replicated [6] and placed at other nodes.
- File is stored at a node whose nodeID is equal to the keyID in the identifier circle. Every file is replicated and stored at two or more locations.
- Copy of the file is stored at locations or nodeID's which are obtained by rehashing the original 160 bit hash value obtained by hashing the filename, and extracting the least significant 20 bits from it. This mirror file is stored at a node whose nodeID is equal to these 20 bits, obtained. Second mirror will be stored at a location obtained by rehashing the second set of 160 bits (which in turn are obtained by rehashing the 160 bits of the file name) and extracting the least significant 20 bits of those 160 bits. Number of replications or the number of mirrors depends on the probability of node failure in the network.
- The client first requests the owner node for the file. If the owner doesn't respond to the request submitted by the client in a stipulated time interval, the client forwards its request

to that node whose nodeID is obtained by rehashing the 160 bit hash of the file key. In this manner the client searches for the file with every mirror until it finds the requested file. Thus ensures the availability of the consistent data even in case of node failure.

- There is a manager node which is always alive, to keep track of modifications in mapping table. If any client submits its request and the node doesn't respond within some stipulated time interval, the client assumes node to be absent in the network and thus reports to the manager node about the absence of that node. The manager node keeps track of number of absences for each node. When the count reaches (log N) the manager node will update the mapping table by deleting the tuple corresponding to that node.
- Number of mirrors is choosen in such a way that the probability of all the mirrors leaving the network is almost null.

3.5 NODE JOINS

Assuming the probability of node joins is not too high; any node that wants to join the network is accommodated at the first available empty space in the mapping table. That is, the incoming new node is given that nodeID which the node that left the network had starting from the beginning of the mapping table. If all the nodes are present in the network, then the incoming new node is assigned the same nodeID beginning from the identifier space '0'.

In this case the mapping table holds two IP addresses corresponding to one nodeID. This node shares the same data that was held with the older node with the same nodeID. If one of the nodes fails, the other simply replaces it.

Although, the probability that all the nodes in network be alive is low, most times new nodes just fill in the empty spaces in the mapping table left by the nodes that left the network.

The following example will give a better idea of node joins:

IP Address	20 bit mapped value

139.78.67.161	00000000000000000000
74.195.252.115	00000000000000000001
148.76.90.87	00000000000000000010
19.248.167.1	00000000000000000100
27.68.69.178	00000000000000000101
169.178.197.1	00000000000000000110
147.248.167.87	00000000000000000111

Table 3.2 illustrates the mapping table which holds the IP addresses and their corresponding nodeIDs when a node left the network

In the table 3.2 the node with nodeID 00000000000000000011 left the network. Any new node that wants to join the network will be assigned a nodeID 00000000000000000011. The Manager node checks for the availability of any empty space in the Mapping table. The first blank space it encounters from the beginning of the table will now host the nodeID of the new node.

CHAPTER IV

RESULTS

4.1 NUMBER OF HOPS

Any request just takes one hop to reach the node which contains the file requested. In case of node failure any request may take a maximum of those many hops as the number of mirrors for any file in the network.

4.2 NUMBER OF COLLISIONS OF NODEIDS

If the nodeIDs are assigned using the following technique then there is a possibility of collision of nodeIDs:

- Hash the IP address of node to a 160 bit value using SHA-1 hashing algorithm
- Extract the least significant 20 bits of the 160 bits, and these least significant 20 bits form the nodeID of the node with corresponding IP address.

When the above technique is implemented practically to assign the nodeIDs to the nodes then there were some collisions found. The following is the data obtained after the simulation:

Number of Nodes	No of collisions
2^{15}	187
2^{16}	623
2^{17}	1134
2^{18}	1814
2^{19}	13712

Table 4.1 showing the number of Collisions.

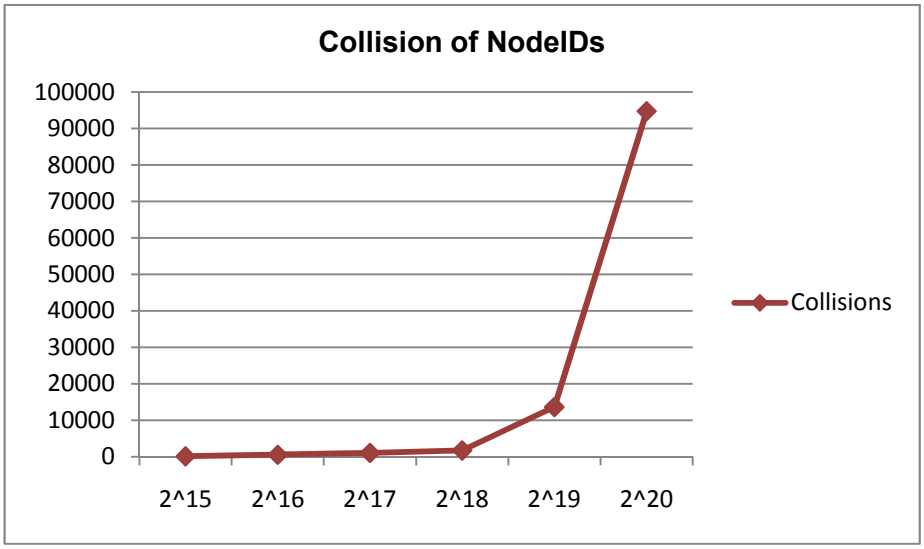


Figure 4.1 Plot of the number of collisions against the number of nodes in the network when the NodeIDs are assigned using the SHA-1 hash function.

When the nodeIDs collide then it is difficult to store and retrieve the data from the which has its nodeID shared with other nodes. Inorder to avoid the problem of collisions, the nodes are assigned the nodeIDs by one to one mapping by the manager node.

4.3 NUMBER OF KEYS PER NODE

Keys are evenly distributed among the nodes to ensure the load balancing. The following results obtained from the simulation show that the files are evenly distributed. Each node is responsible for at most K/N keys (in a network with N nodes and K keys).

AVERAGE NUMBER OF FILES PER NODE:

NUMBER OF FILES	NO OF NODES	AVG NUMBER OF KEYS PER NODE
2^{20}	2^{20}	2
2^{22}	2^{20}	4.8
2^{25}	2^{20}	31
2^{28}	2^{20}	226
2^{30}	2^{20}	997

Table 4.2 showing the average number of keys per node.

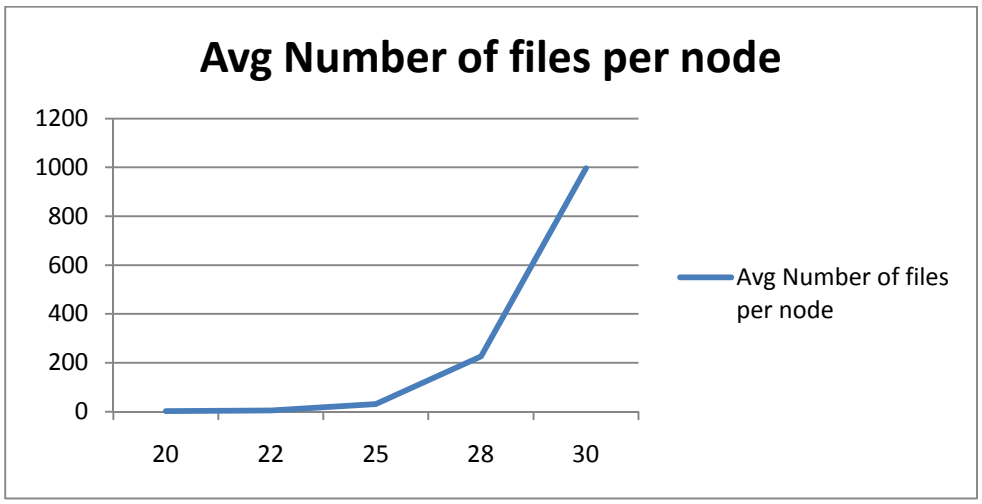


Figure 4.2 Plot of average number of keys per node when 1-1 mapping is used to assign node ids and number of nodes is 2^{15} as the number of files varies from 2^{15} to 2^{20}

Maximum number of keys held by a node

NUMBER OF FILES	NO OF NODES	MAXIMUM NUMBER OF FILES PER NODE

2^{15}	2^{13}	4
2^{15}	2^{14}	5
2^{15}	2^{15}	6
2^{15}	2^{16}	10
2^{15}	2^{17}	15
2^{15}	2^{18}	23

Table 4.3 showing the maximum number of keys per node.

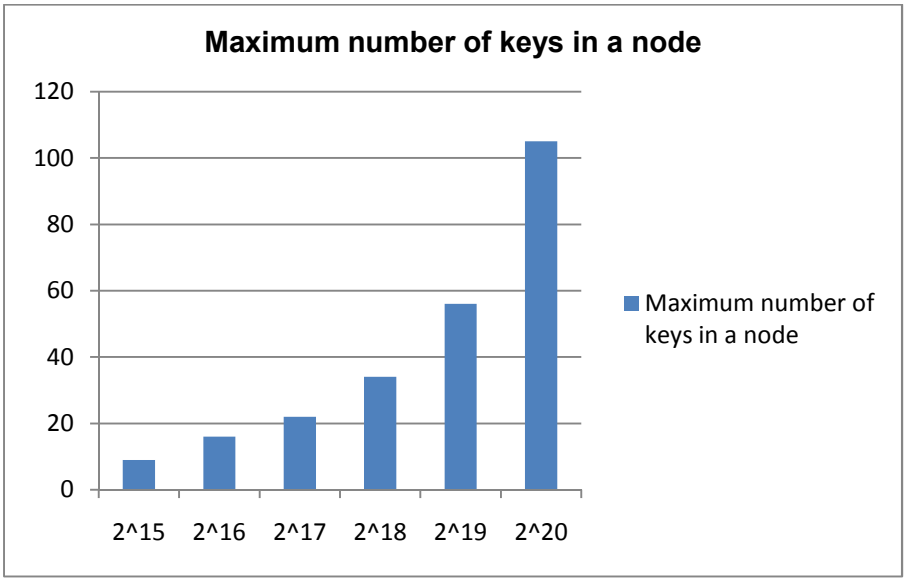


Figure 4.4 Plot of the maximum number of keys a node can have as the number of nodes is 2^{15}, while number of keys varies from 2^{15} to 2^{20}.

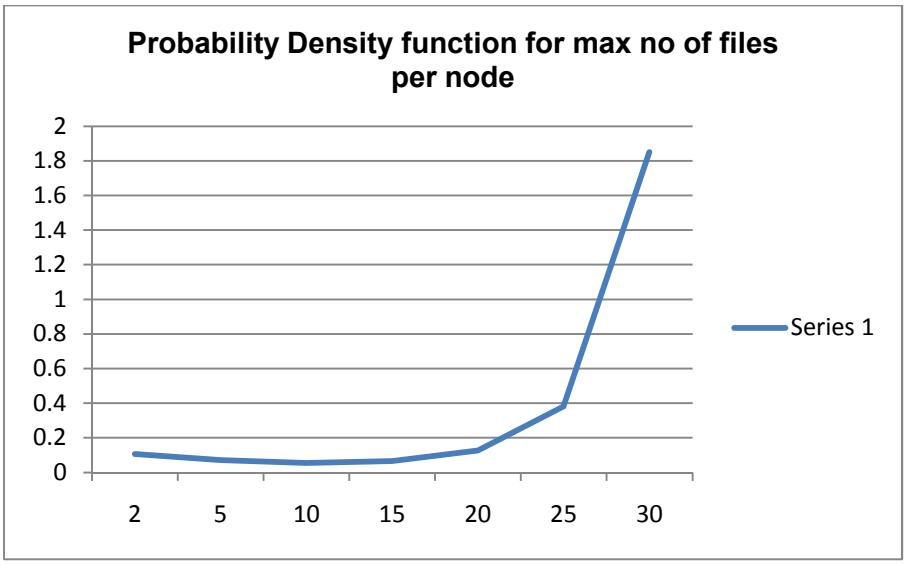


Figure 4.5 Probability Density function (PDF) of the maximum number of keys per node

4.4 NUMBER OF MIRRORS

Query hit is when a node gets a reply for its query for a file. Query miss is when a node doesn't get the file it queried for. The following are the results of the simulation obtained when the nodes leave the network with various probabilities.

Fraction of nodes failed	No of files searched	No of existing files	No of non existing files	Total no of files on system	No of Nodes in the system
0	100000	63490	36510	2^{20}	2^{15}
0.05	100000	60203	39797	2^{20}	2^{15}
0.10	100000	57004	42996	2^{20}	2^{15}
0.15	100000	54201	45799	2^{20}	2^{15}
0.20	100000	52040	47960	2^{20}	2^{15}
0.25	100000	49247	50753	2^{20}	2^{15}

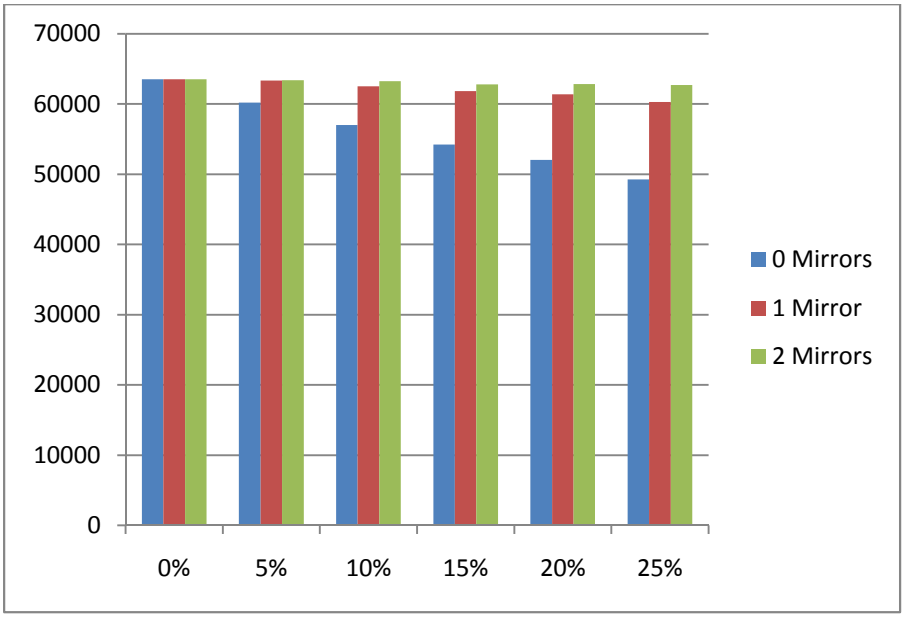
Table 4.4 Data obtained when the network stores no mirrors

Fraction of nodes failed	No of files searched	No of existing files	No of non existing files	Total no of files on system	No of Nodes in the system
0	100000	63171	36829	2^{20}	2^{15}
0.05	100000	63321	36679	2^{20}	2^{15}
0.10	100000	62512	37488	2^{20}	2^{15}
0.15	100000	61818	38182	2^{20}	2^{15}
0.20	100000	61388	38612	2^{20}	2^{15}
0.25	100000	60268	39732	2^{20}	2^{15}

Table 4.5 Data obtained when the network stores 1 mirror that is each file is replicated once.

Fraction of nodes failed	No of files searched	No of existing files	No of non existing files	Total no of files on system	No of Nodes in the system
0	100000	63183	36817	2^{20}	2^{15}
0.05	100000	63366	36634	2^{20}	2^{15}
0.10	100000	63248	36752	2^{20}	2^{15}
0.15	100000	62770	37230	2^{20}	2^{15}
0.20	100000	62829	37171	2^{20}	2^{15}
0.25	100000	62684	37316	2^{20}	2^{15}

Table 4.6 Data obtained when the network stores 2 mirror that is each file is replicated twice



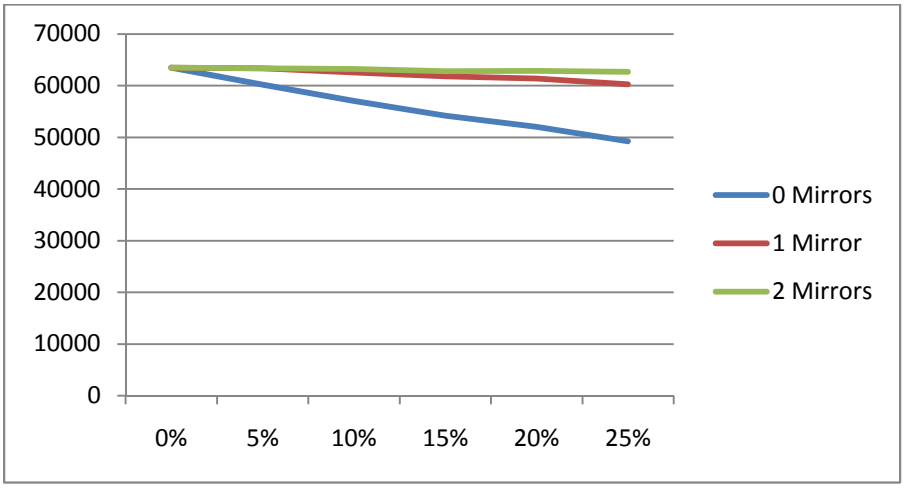


Figure 4.6 Plot of Queryhit against the fraction of node failure for a network of size 2^{20} nodes and the number of queries = 100000

X-axis represents the percentage of node failures. Y-axis represents the Number of files found in the network.

The above results are obtained for a network size of 2^{15} with 2^{20} files stored on them. When the network is searched for 100000 files, the above are the files found in the network. This shows that the success rate of file search is very high in the network with 1 mirror when compared with the network with no file replication. As compared with the network with 2 mirrors and 1 mirrors there is only a little difference between the corresponding success rates.

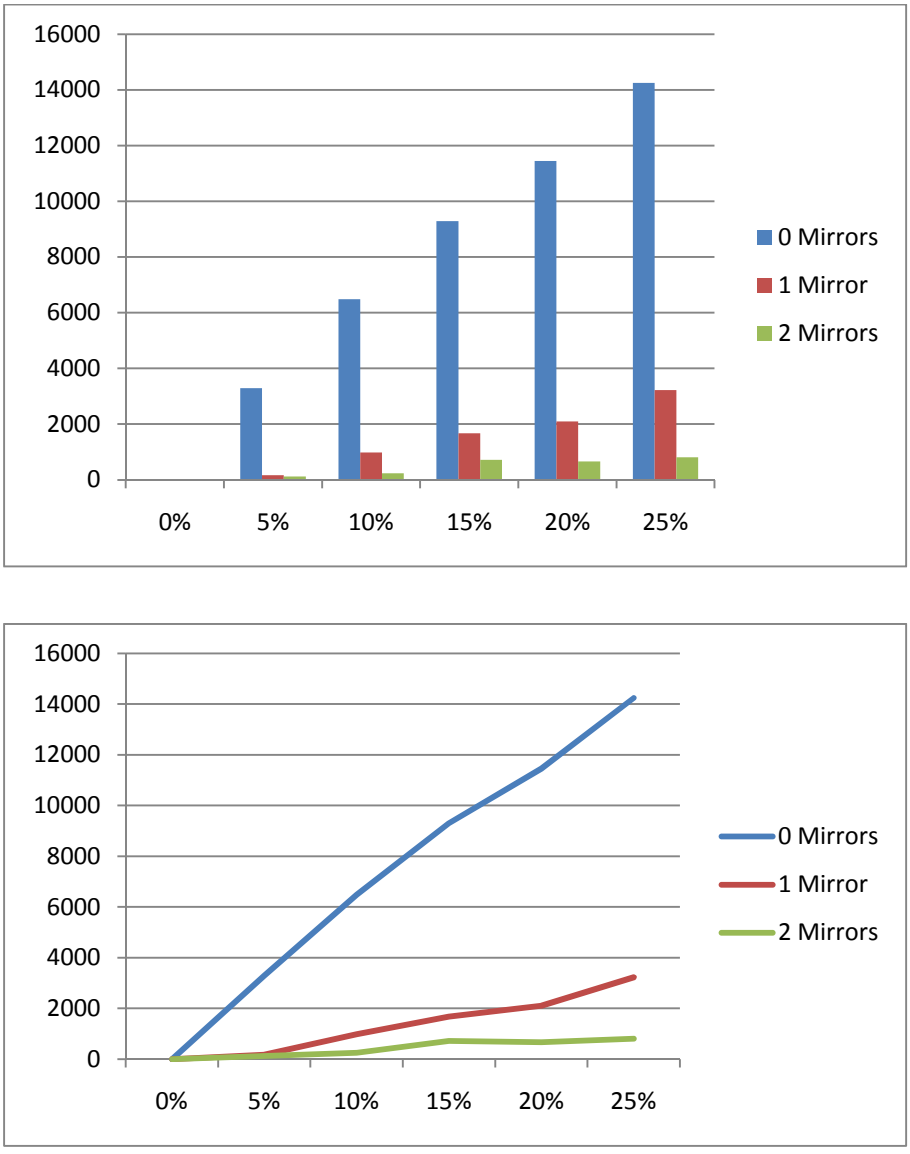


Figure 4.7 Plot of Querymiss against the fraction of node failure for a network of size 2^{20} nodes and the number of queries = 100000

From figure 4.7 it can be seen that the node failure is very low when there is a file replication. The above results show that when there are 2 mirrors in the network, the rate of misses due to node failures is very low. If the network stores 2 mirrors for each file, even when 25% of the nodes in the network fail, only less than 1% of the queries are not answered. This means having 2 mirrors for each file will make the network, resistant to the node failures.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

CHORELLA is a distributed data addressing scheme built upon the ideas of Chord and Gnutella. The data is evenly distributed among all the nodes in the network, thus ensuring load balancing.

In CHORELLA, any file search operation takes just one hop. Only in case of node failures if the target or the owner load is not present in the network, then the protocol may take at most as many hops as there are number of mirrors in the network.

To ensure data availability the data is replicated and stored at more than one location. This makes the CHORELLA reliable and tolerant to node failures.

The main characteristics of the proposed scheme are as follows:

- For any set of N nodes and K keys, with high probability, each node is responsible for at most K/N keys
- When an (N+1)st node joins or leaves the network, responsibility for keys need not change hands.
- Any file search operation in network will take only one hop in case of no node failure
- If we take a stable network with N nodes, and another set of up to N nodes joins the network, then lookups will still take only one hop with high probability.

- The network maintains 2 mirrors for each file. Experimentally it is shown that if each file is replicated twice, even in case of massive node failures only 1% of the queries are unanswered which is negligibly small.
- In case of node failures each file search may at most take 3 hops to reach the destination node.

FUTURE WORK

Further research needs to be done to analyze the performance of CHORELLA and improve its scalability. Also the mirror nodes in CHORELLA can be used to store different shares of a file. When the shares are brought together the file can be recreated. One can see this most clearly when the error rate in the CHORELLA network is assumed to be zero. In that case, all the mirror nodes -- or a subset of them can be used to store files using algorithms of implicit security [22], [23].

REFERENCES

[1] R. Morris, M. F. Kaashoek, D. Karger, H. Balakrishnan, I. Stoica, D. Liben-Nowell, F. Dabek : Chord: A scalable peer-to-peer lookup protocol for internet applications http://pdos.csail.mit.edu/papers/Chord:sigcomm01/Chord_sigcomm.pdf

[2] GNUTELLA Website: http://gnutella.wego.com. (Retrieved on October 2009)

[3] J. Vaucher, G. Babin, P. Kropf, and Th. Jouve, “Experimenting with Gnutella Communities”, Distributed Communities on the web (DCW 2002), Sydney, Australia. LNCS 2468, Springer Berlin, pp.85-99, 2002.

[4] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. 2nd International Workshop on Peer-to-Peer Systems (IPTPS), pages 80–87, 2003.

[5] Frank Dabek, E. Brunskill, Franks Kaashoek, and David Karger. Building peer-to-peer systems with Chord, a distributed lookup service. Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany, May 2001.

[6] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. Proceedings of the 29th Annual ACM Symposium on Theory of Computing (El Paso, TX), pp. 654–663, 1997.

[7] Ripeanu, M., "Peer-to-Peer Architecture Case Study: Gnutella Network", Proceedings of IEEE 1st International Conference on Peer-to-Peer Computing Linkoping, Sweden 2001.

[8] Distributed Systems—Architecture and Implementation, ed. Lampson, Paul, and Siegert, Lecture Notes in Computer Science 105, Springer, pp 246-265 and pp 357-370 1981.

[9] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In ACM Symposium on Theory of Computing, pages 654–663, 1997

[10] Ion Stoica, Robert Morris, David Karger, Francs Kaashoek, and Hari Balakrishnan. Chord a scalable peer-to-peer lookup service for internet applications. Proceedings of the 2001 ACM SIGCOMM Conference, pages 149–160. ACM Press, 2001.

[11] www.systemexperts.com/assets/tutors/CryptographicHashUpdate.pdf

[12] Andy Oram (ed.) Peer-To-Peer: Harnessing the Power of Disruptive Technologies, Edited by published by O'Reilly, 2001

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

[14] D. Lewin, Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Department of EECS, MIT, 1998.

[15] The Gnutella protocol specification v4.0 -http://dss.clip2.com/GnutellaProtocol04.pdf (Retrieved on October 2009)

[16] C. Plaxton, R. Rajaraman, And A. Richa, Accessing nearby copies of replicated objects in a distributed environment. Proceedings of the ACM SPAA (Newport, Rhode Island), pp. 311–320 1997.

[17] Evangelos P. Markatos: Tracing a large-scale Peer to Peer System: an hour in the life of Gnutella. Proceedings of the CC Grid 2002: the second IEEE International Symposium on Cluster Computing and the Grid, pages 65-74, 2002.

[18] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M. Replication in the Harp file system. In Symposium on Operating Systems Principles pp. 26–38, 1991.

[19] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, Scalable, distributed data structures for Internet service construction. Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000), 2000.

[20] M. Castro, M. Costa, and A. Rowstron Debunking some myths about structured and unstructured overlays. Proceedings of the 2nd Symposium on Networked Systems Design and Implementation 2005.

[21] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, Ivy: A read/write peer-to-peer file system. Proceedings of the 5th OSDI 2002.

[22] A. Parakh and S. Kak, "Online data storage using implicit security." Information Sciences, vol. 179, pp. 3323-3331, 2009.

[23] A. Parakh and S. Kak, "A Key Distribution Scheme for Sensor Networks Using Structured Graphs." IEEE ELECTRO-09, Varanasi, Dec 22-24, 2009.

[24] Kelsey Anderson, "Analysis of the Traffic on the Gnutella Network", University of California, San Diego CSE222 Final Project, 2001.

VITA

Sravanthi Peruru

Candidate for the Degree of

Master of Science

Thesis: A RELIABLE AND SCALABLE ADDRESSING SCHEME FOR DATA DISTRIBUTION

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Master of Science in Computer Science at Oklahoma State University, Stillwater, Oklahoma in July, 2010.

Completed the requirements for the Bachelor of technology in Computer Science at Jawaharlal Technological University, Hyderabad, AndhraPradesh, India in 2008.

Experience:

Research Analyst at Center for Telecommunication and Network Security, Oklahoma State University, Stillwater, OK

Webmaster at Political Science and Zoology Departments at OklahomaState University, Stillwater, OK.

Name: Sravanthi Peruru Date of Degree: July, 2010

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: A RELIABLE AND SCALABLE ADDRESSING SCHEME FOR DATA DISTRIBUTION

Pages in Study: 38 Candidate for the Degree of Master of Science

Major Field: Computer Science

A central problem faced in distributed data storage and in peer to peer applications is the efficient location of the node where the requested data is present. Currently popular peer to peer protocols such as Chord and Gnutella require enormous overhead. In Chord protocol data needs to be moved back and forth in order to stabilize the network and in Gnutella the requests flood the network. The above problems call for an efficient and reliable addressing scheme. The protocol proposed in this thesis (CHORELLA) can accommodate a maximum of 2^{20} nodes in the network. It takes just one hop to find any file in the network and in case of node failures, it may take a maximum of as many hops as the number of mirrors or the replication rate in the network. The files are evenly distributed among the nodes using SHA-1 hashing function.

ADVISER'S APPROVAL: Dr. Subhash Kak