Peer to peer systems: An overview

Gaurav Veda (Y1148)
gveda@cse.iitk.ac.in
Computer Science & Engineering
Indian Institute of Technology
Kanpur, UP, INDIA - 208016

***Abstract*— Peer-to-peer (p2p) systems is an exciting and emerging field. It is much beyond enabling people to illegally share copyrighted music and video files. As we progress towards a world with better and ubiquitous connectivity, more and more people are switching to a p2p solution from a centralized server solution.**

In this report, I first define what is meant by a p2p system and when should one go for a p2p solution as compared to a centralized server solution [1]. I then describe what is a Distributed Hash Table (DHT) and present two popular routing protocols based on DHT's - Chord [2] and CAN [3]. After this, I move on to describe how the underlying geometry of the DHT based routing algorithm affects its static resilience and proximity properties [4]. This will suggest that for designing DHT based routing protocols, ring geometry is the best choice.

I. Introduction

In recent years there has been a sudden rise in interest in p2p systems. Peer to peer systems caught the world's attention because of the much publicized case of Napster [6] versus the music and record companies [7]. For many people, a peer to peer system is simply a medium that enables content sharing. However, p2p systems have many more uses and applications than simply enabling illegal sharing of media files. Infact, now more and more people are gravitating towards deploying a peer to peer solution for various problems that till now had centralized server based solutions. p2p systems is also a hot topic for research as can be gauged from the large number of recent highly cited publications and major university projects in this domain.

The purpose of this report is to familiarize the reader with peer to peer systems and the research taking place in the direction of routing in p2p systems. In the sections that follow, we will first understand what is meant by a p2p system. We will then take a look at a decision tree and various issues that help decide the viability of a p2p solution, as compared to a centralized server solution, for the problem at hand. After this, we move on to the issue of efficiently routing in a peer to peer system. Most of the recent p2p routing protocols are based on Distributed Hash Tables (DHT's). We will look at two popular routing protocols that use DHT's, namely Chord and CAN. As will be clear after reading about these protocols, DHT's can use a variety of underlying geometries like ring, hypercube etc. So, we will try to address the question of which geometry to choose for a new DHT routing algorithm, by exploring the relation between the underlying geometry of a DHT routing protocol and its static resilience and proximity properties.

II. Peer to Peer Systems

Let us first understand what is meant by a peer to peer system. The philosophy behind p2p networks is to have a system that enables end-point resources to be shared. The resources might be files (in a file-sharing system), storage space, CPU cycles etc. The defining feature of a p2p infrastructure, is the ability of end systems (ie. peers) to communicate with each other. In this sense, a p2p system can be viewed as an overlay network over the internet. Beyond this, there is no clear-cut definition of a p2p system. Here, I present the view held by a large number of people.

According to [1], there are three defining characteristics of a p2p system:

- *Self-organizing:* Nodes must organize themselves to form an overlay network. There should be no assistance from a central node. Also, there should not be any global index that lists all the peers and/or the available resources.
- *Symmetric communication:* All nodes must be equal ie. no node should be more important than any other node (so according to this definition, systems like the KaZaa file sharing application [8], that have a notion of super-peers do not qualify as a p2p system). Also, peers should both request and offer services ie. they should act as both clients and servers, rather than having a role as only a client or only a server.
- *Decentralized control:* There should not be a central controlling authority that dictates behaviour to individual nodes. Peers should be autonomous and must determine their level of participation in the network on their own.

III. 2 P2P or Not 2 P2P

The title of this section has been taken from [1] and essentially is a summary of that paper. We first look at the parameters that govern whether we should go for a p2p solution or not. These are:

- **Budget:** The single most important factor that dictates whether we go for a p2p solution or not, is the budget available to us. A p2p solution inherently suffers from inefficiencies, latencies and testing problems. Hence, if the budget is adequate for a centralized solution, a person is likely to choose that. However, if the budget is low, p2p systems offer the advantage that they can be build incrementally. As and when more budget/resources become available, they can be added on to the system.

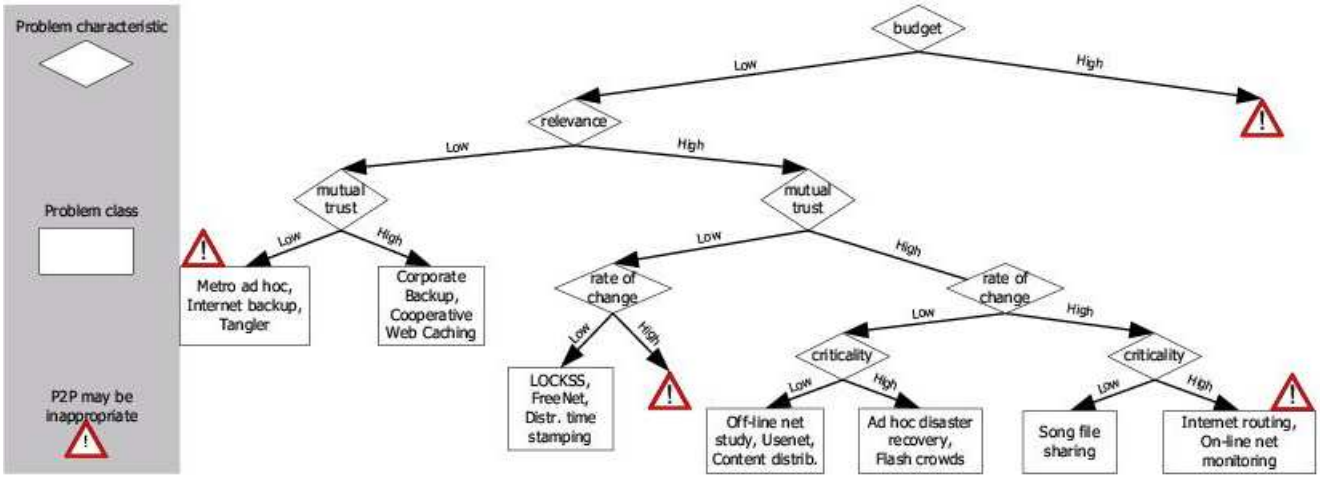

Fig. 1. Diagram taken from [1]. This decision tree can be used to analyze the viability of a p2p solution. A few sample applications for various possible problem classes are mentioned in the square boxes.

So, although the final total cost of a p2p solution might come out to be higher, it is much easier to build.

- **Resource Relevance:** If a single "unit of service" of the problem is of interest to many participants, then they are likely to cooperate and this makes it much easier to go for a p2p solution. An example might be a file sharing system. A single file is of relevance to many participants and hence they all cooperate. If the relevance is less, then achieving cooperation between hosts (for a p2p solution) would require other incentives, and it might lead to increased costs.
- **Trust:** If the peers do not trust each other, we might have to go for external trust mechanisms such as encryption etc. which would lead to additional costs (in terms of efficiency etc.). However, in certain cases, some amount of mistrust might be desirable, since it would lead to mechanisms that would ensure that the damage caused by a misbehaving peer is limited and that there can be fault isolation.
- **Rate of system change:** It the problem at hand has high requirements of consistency and timeliness while the participants in the system, the data or the system parameters have a high rate of change, a p2p solution becomes very difficult. This is because a p2p solution has inherent inconsistencies and latencies.
- **Criticality:** If the problem in question has a very high importance for the users and is critical to them, then they might demand a centralized control and accountability mechanism. This more or less rules out a p2p solution. If we still want a p2p solution, it might be necessary to do massive over-provisioning and take steps for fault tolerance, which might make the system very costly and very slow.

Figure 1 (courtesy [1]) is a decision tree that shows the aforementioned parameters graphically. We now look at some of the problems (mentioned in square boxes in the diagram) that have proposed P2P solutions, to see how good the above factors are, in determining the viability of a P2P solution.

A. Ad-hoc Routing in Disaster Recovery

At the site of a calamity, all the previously existing communication infrastructure becomes useless and communication is re-established using transient resources (like the wireless devices of recovery crews). The range of such devices is small and so they must route each other's packets. Since this is a highly relevant and critical application, and there is a large amount of mutual trust between participants (which can be strengthened using pre-configured security mechanisms), a p2p solution is quite effective. Moreover, once the network is established, there is little movement of the resources (ie. low rate of change).

B. Metropolitan-area Cell Phone Forwarding

In this problem, we want to bypass the base station to save on communication cost. This is achievable if a cell phone forwards the packets of other cell phones. However, unlike in the previous example of ad-hoc routing, here there is little trust between individual peers. The task is also not very critical and the rate of change is high. Therefore, as expected from the decision tree, this application hasn't been quite successful.

C. Backup

Backup procedures can be broadly classified into Internet Backup and Corporate Backup. In internet backup, the peers don't know each other and hence there is little trust between peers. There is no guarantee that a peer p_1 will not tamper/delete the backed up data of another peer p_2, that is supposed to be stored at p_1. Because of these factors, internet backup hasn't been quite popular.

In a corporation, since all machines are owned by a single authority, there is a large amount of trust between individual peers and because of this corporate backup is much better suited for a p2p solution as compared to internet backup.

D. File Sharing

This was the application that brought peer-to-peer systems in focus. Here, resource relevance is high since a user is typically interested in downloading various files from others. The user also trusts the other nodes to deliver the advertised content. Moreover, this is not a critical application and is very low cost (all it needs is an internet connection). Because of these reasons, inspite of the high rate of system change (nodes continuously enter and exit the system) file sharing is one of the most successful p2p applications.

IV. Consistent Hashing

The technique of hashing provides a nice way to efficiently store and retrieve information based on a key. A hash function is an important requirement is most applications. As we will see later, it is required by all the routing protocols that we will look at later in this report. In this section, we will be looking at a special kind of hash function, namely, a consistent hash function, that forms the basis of the Chord protocol that we will look at in section 6. This section can be viewed as a summary of [9] and [10].

A typical hash function, such as $ax + b \ modulo \ p$ (where a and b are integers and p is a prime), changes dramatically when the range of the hash function is changed (here, this is equivalent to changing the prime p). Even a very small change causes an almost complete re-mapping of the domain elements to the range set. On the internet, machines frequently come and go as they are brought into the network or taken out (or they crash). Information about which machines are currently present on the network and which are not, travels slowly in the network. As a result, different machines may have differing views of the machines that are currently online. In other words, one machine might think that a machine x is on the network currently, while another machine might think that x is not present on the network.

If we consider a client-server scenario where different clients have different views of the servers that are currently available, this leads to various problems. As an example, let us suppose that the servers are web-page caching servers. When a client wants a particular web-page, it looks at the available servers (ie. the servers it thinks are online), and using a hash function, decides which server should it contact for the web-page. If the server does not have the page, it contacts the main internet server, and gets a copy of the web-page. Now if we use a typical hash function, then since the range ie. the servers that appear online to a client, keeps on changing, the server that a client contacts for a particular web-page, will keep on changing. So, as soon as the client sees that one more server is online or some server goes offline, it will recompute which server to contact for each and every web-page and suddenly, all the cached data might become useless ! Moreover, since the servers that one client thinks are available might be different for each client (although different views might differ in only one or two servers), every web-page must reside on as many servers as the number of clients or if the number of clients is more than the number of servers, each page must reside on each server. This is a colossal waste of resources, since the views for most clients will differ only in a very small number of servers and we would like that many clients should contact the same server for a particular web-page (so that the total information that is to be stored in all the web servers, comes down). To overcome these and other problems, we define a consistent hash function.

Before we look at a particular consistent hash function, let us define some of the desirable properties of such a function. In what follows, we will assume that each client knows about atleast $1/t$ of the total number of caches (or servers) that are currently online/active.

Let $\mathcal{I}$ be the total set of keys (or items) that any client is interested in. So, for each client, we have to map every element in the set $\mathcal{I}$ to some machine or bucket. Let $\mathcal{B}$ be the set of buckets (ie. the total number of servers - whether online or not). A *view* is any subset of machines out of the set $\mathcal{B}$. Every client has an associated view, which is essentially the client's view of the world viz. the currently active servers (according to the client). A function f is called a ranged hash function if it is of the form $f : \mathcal{V} \times \mathcal{I} \mapsto \mathcal{V}$, where $\mathcal{V}$ is a view. In other words, given a view $\mathcal{V}$, the function f maps every item i onto a bucket in the view. Let us denote the term $f(\mathcal{V}, i)$ as $f_{\mathcal{V}}(i)$. A ranged hash family $\mathcal{F}$, is a family of ranged hash functions. The properties that are desired from a consistent hash function are:

- **Balance:** A ranged hash family $\mathcal{F}$ is said to be balanced, if, for a particular view $\mathcal{V}_i$ and a randomly chosen function f ($f \in \mathcal{F}$), the fraction of items mapped to each bucket is O(1/$\mid \mathcal{V}_i \mid$), with high probability.
- **Monotonicity:** Suppose we have a set of buckets $\mathcal{V}_1$ and we map every item to this bucket. Now we add a few more buckets to form $\mathcal{V}_2$. Now the mapping of item changes. A ranged hash function f is monotonic, if, an item may move from an old bucket to a new bucket, but not from one old bucket to another.
- **Spread:** If we have a set of views, an item might map to different buckets in different views. The spread of an item is the number of buckets that it maps into. The spread of a hash function is the maximum spread of an item. A good hash function should have low spread so that the total storage needed is small.
- **Load:** The load of a bucket is the number of distinct items that map to it across all the views. Basically, this denotes the items for which this machine will be considered responsible (by at-least one client) and hence must store information about. The load of a hash function is the maximum load of a bucket. Since we desire load-balancing among the nodes, a good hash function should distribute load almost equally among all the nodes.

Construction: Now we look at a particular consistent hash function that satisfies the above properties. Consider a unit circle. Let us map items and buckets independently and randomly onto points on the circle. The hash function maps an item (key) to the bucket (machine) that is next to the item

in the clockwise direction (on the circle). In order to ensure a uniform allocation of items to buckets, a bucket is mapped onto 'm' random points on the circle. If we the number of buckets in N, then the value of m will be $k.log(N)$, where k is a small constant. Lets call this family UC_{random}.

Analysis: Following are some of the theorems presented in the paper [9] and the thesis [10]. The proof of all these theorems is based on Chernoff bounds. The proofs are explained in detail in [10]. Here, I just state the important theorems without their proofs.

Theorem IV.1. *Let $\mathcal{V} = \{V_1, V_2, \ldots, V_k\}$ be a set of views of the set of buckets $\mathcal{B}$ such that: $\mid \bigcup_{j=1}^{k} V_j \mid = T$ ie. T is the set of all the servers that appear active in atleast one view. Let $\forall\ 1 \leq j \leq k, \mid V_j \mid \geq T/t$. Let N be a confidence parameter. If each bucket is replicated and mapped m times then:*

- *Monotonicity: The family UC_{random} is monotone.*
- *Spread: For any item $i \in \mathcal{I}$, $spread_f(\mathcal{V}, i) = O(t\ log(Nk))$ with probability greater than 1 - 1/N over the choice of $f \in UC_{random}$.*
- *Load: For any bucket $b \in B$, $load_f(\mathcal{V}, b) = O\left(\left(\frac{|\mathcal{I}|}{T} + 1\right) t\ log(Nmk)\right)$ with probability greater than 1 - 1/N over the choice of $f \in UC_{random}$.*
- *Balance: For any fixed view V and item i, the probability tha i is mapped to bucket b in view V is $O\left(\frac{1}{|V|}\left(\frac{log(N|V|)}{m} + 1\right)\right) + \frac{1}{N}$.*

Note that in the above theorem, we have assumed that bucket copies and items are mapped to the unit circle using fully independent hash functions. However, in reality, having fully independent hash functions is impractical. We always talk of a k-way independent hash function. In the next theorem, we show that we can work even with limited independence.

Theorem IV.2. *If bucket copies and items are mapped to the unit circle using a $\Omega(t\ log(NTk))$-way independent hash function, and item points and bucket points are mapped independently, Theorem 1 holds.*

Another big problem with the hash function in its present form is that we assume that we have infinite precision. This would require infinite bits to represent a point on the circle. In reality, we have to restrict the precision of a point on the circle to reduce the number of bits needed to represent it. The following theorem shows that we require only a small number of bits to represent a point on the circle for the hash function to work as desired.

Theorem IV.3. *With probability at least 1 - 1/N', the clockwise ordering of n random points in the unit circle is determined by the $2\ log(N'n)$ most significant bits of the points.*

Theorem 2 and 3 above attest the claim that this hash function is practical and can be deployed in the real world. Combining Theorem 2 and 3, we get Theorem 4.

Theorem IV.4. *If the mappings of items and bucket points are $\Omega(t\ log(NTk))$-way independent (N is the confidence parameter of Theorem 1), items are mapped independent of bucket points, and $c = O(log(N'(m \mid \mathcal{B} \mid + \mid \mathcal{I} \mid)))$ bits of precision are used, then Theorem 1 holds with probability at least 1 - 1/N'*

V. Distributed Hash Tables (DHT's)

Let us consider a p2p file sharing system. Here, every client shares some files and is interested in downloading some files from other peers. The central problem here is to be able to retrieve a file that the client wants. There are various ways to do this.

One way is to have a centralized index, like the one maintained by Napster [6]. Here, we have a central Napster server that maintains a list of all the currently shared files. Whenever a node joins the network, it informs the central server about the files shared by it, and the server incorporates this information in its index. Now when a node wants to search for a file, it sends a search request to the central server. The server performs a lookup on its index and returns the results (in the form of IP addresses of machines having that file) to the requesting node. The node now contacts the node containing this file and downloads the file from it. This solution has several problems. Because of a central server, there is a single point of failure. The central server is highly loaded since it not only has to store a large amount of information, it also has to perform numerous queries on it. There is also a legal problem here. Since there is a single node that maintains the index of files, and performs searches on it (and hence knows about the available content), it can be easily sued for promoting illegal sharing (as happened in the case of Napster [7]).

Another solution is to use limited flooding, as in Gnutella [11]. In this approach, when a node joins the system, it comes to know about a few other nodes through a simple mechanism. Now whenever it wants to retrieve a file, it sends a request to its neighbours. If a neighbour has the file, it informs the requesting node and file transfer can take place. Also, at the same time, the host node can either cancel the request that it sent to other nodes, or it can refuse the other nodes when they return positive responses. If the neighbour does not have the file, it broadcasts the request to all its neighbours (other than the one from whom it got the request - to avoid cycles). Each request can have a counter that contains a small number (7 in the case of Gnutella) that is decremented each time a request in forwarded, thus ensuring that every request does not reach every other node and clogs the network. Although this approach is completely decentralized and does not have a single point of failure, there are still many problems. Even though we are doing limited flooding, the number of messages exchanged for a single query is huge and there is huge waste of network resources. Also, even though a file might be available on the network, a node might not be able to retrieve it.

In order to avoid the problems of the previous approaches, we need a system that given a key (such as a file name), can quickly tell which node stores that key. A hash table that distributes keys to nodes can do the trick. Such a hash table is known as a Distributed Hash table (DHT). DHT's can be

used to store data as well as to do routing. Thus, a DHT offers a decentralized, scalable and fault tolerant solution to the problem of locating the node that stores a key in a dynamic, distributed environment. A DHT based approach can also be used to provide anonymity in the system.

VI. CHORD

The most important problem faced by a node in a p2p system based on DHT's, is to locate the node that stores a particular key, in an efficient way. This is the problem addressed by Chord - a distributed lookup protocol [2]. All that Chord does is, given a key, it maps the key onto a node. It must be noted that Chord does not provide any authentication, replication, caching etc. and hence the application that runs over Chord must itself provide the features that it needs (eg. data replication to ensure availability even when some nodes abruptly leave the system).

Chord does this lookup in an efficient and scalable manner (as is indicated by results from theoretical analysis, simulation and experiments). It uses a consistent hash function (a variant of the function described in Section 5) to assign keys to nodes. This ensures that all nodes are responsible for roughly the same number of keys and whenever a node joins or leaves the system, only a very small number of keys are transferred between nodes. Chord is a completely decentralized system. No node in Chord is more important than any other node. The protocol is designed in a way that ensures that unless there is a major failure in the underlying network, the node that is responsible for a key can be found out by any other node in the system (although efficient querying might not be possible).

We now describe how the protocol works. The protocol uses the SHA-1 hash function to assign an m-bit identifier to every key and node. m is chosen to be large enough so that there is a very low probability of two nodes or two keys hashing to the same m-bit identifier. A node's identifier is obtained by hashing the its IP address, and a key identifier is obtained by simply hashing it. It must be noted that in consistent hashing, we speak about a k-way independent hash function that maps keys and nodes onto the identifier space, while here we use SHA-1. This makes the Chord protocol deterministic. However, the use of SHA-1 can be justified because producing a set of keys that collide under SHA-1 is equivalent to breaking or decrypting SHA-1. However, since it is believed that SHA-1 is difficult to break, the hash function can be considered to be equal to (or better than) a k-way independent hash function and hence all the bounds derived in Section 5 still old. Moreover, the type of hash function that we choose to use does not affect the Chord protocol (as long as it is equivalent to a k-way independent hash function).

A. Allocation of keys to nodes

Key k (with identifier i_k) is assigned to the first node (n) whose identifier (i_n) is equal to or follows the identifier of k ie. i_n is the smallest node identifier such that $i_k \leq i_n$ (module 2^m. This node is called the $successor$ node of key k, denoted by $successor(k)$. If identifiers are represented on a circle by having 2^m ordered points on the circle corresponding to the numbers from 0 to $2^m - 1$, then $successor(k)$ is the first node clockwise from k. When a new node n joins the network, in order to maintain this structure, certain keys previously assigned to n's successor now become assigned to n. When node n leaves the network, all the keys that were previously assigned to it are reassigned to its successor. Since we use a consistent hash function, whenever a node joins or leaves the system, only $O(K/N)$ keys need to be transferred from a node to its successor or vice versa (where K is the total number of keys and N is the number of nodes currently active). Also, since in Chord we map each node to a single point on the identifier circle, we can no longer ensure that perfect load balancing takes place with a high probability. Instead, we can just say that on an average, a node will be responsible for at most $O(logN) \cdot K/N$ keys.

B. Key Lookup

Naive Key Lookup: Every node maintains its successor information correctly and keeps on passing the query to its successor until the destination node is reached. This takes $O(N)$ time. However, this lookup protocol only needs that there is just one entry in a node's routing table that is correct. Hence this technique can be used when the network is changing rapidly (ie. many nodes are joining and leaving the system within a short time span) and the routing tables are quite inconsistent.

Normal (fast) Lookup: Each node n maintains a routing table with m (length of key/node identifiers) entries, called the *finger table*. The j^{th} entry in the table at node n contains the identity of the first node s, that succeeds n by atleast 2^{j-1} on the identifier circle ie. s = $successor(n + 2^{j-1})$ or i_s is the smallest node identifier such that $i_s \geq i_n + 2^{j-1}$ where $1 \leq j \leq m$ and all arithmetic is modulo 2^m. Node s is called the j^{th} finger of node n.

Due to the way in which the finger table is maintained, a node knows more about nodes that closely follow it on the identifier circle. This guides the way in which lookup for a key k is done. If a node n can find a node whose identifier is closer than its own (i_n) to i_k, that node will know more about the identifier circle in the region of k than n does. So n searches its finger table for the node j whose identifier most immediately precedes that of k, and asks j for the node it knows whose identifier is closest to i_k. By repeating this process, n learns about nodes with identifiers closer and closer to k. Eventually, the node that is the immediate predecessor of k will be reached and its successor will contain the key k. With high probability, the number of nodes that must be contacted to find a successor in an N-node network is $O(log\ N)$ - less than the expected $O(m)$.

Theorem VI.1. *With high probability, the number of nodes that must be contacted to find a successor in a N-node network is* $O(logN)$.

Proof: Suppose a node n wants to find the location of a key k. Let p be the immediately preceding node of k. Now, if

$n \neq p$, n will forward the query to the node in its finger table that is just preceding k. Let us suppose that p is at a distance in the range $(2^{i-1}, 2^i]$ ie. it is in the i^{th} finger interval of the node n. This implies that this interval is not empty and so there exists a node f in the i^{th} finger table entry of n. Therefore, the distance between n and f is atleast 2^{i-1} and the distance between f and p is less than 2^{i-1} (since both are in the same interval). So if we go to f from n, then we have reduced the distance by atleast half. So we will require at most m steps to reach p.

After k hops, the distance to p reduces by a factor of atleast 2^k. Therefore, after $logN$ hops, the distance between the current query node and p will be at most $2^m/N$. Now since there are only N active nodes currently, the expected number of nodes in this interval is 1 and $O(logN)$ with high probability. So, now even if we proceed by simply going to the immediate successor node, we will still reach the destination in $O(logN)$ hops. Therefore, on an average, we require only $O(logN)$ hops for a lookup.

C. Node Joins

Chord preserves two invariants to achieve correct (though slow) routing in case of node joins and exits:

- Each node's successor is correctly maintained.
- For every key k, the node $successor(k)$ is responsible for it.

In addition to the above, for lookups to be fast, the finger tables need to be correct as well.

It is assumed that a new node n, due to some external means, knows a node n' that is already part of the Chord network. Also, to simplify the process of node join and exit, we also maintain a predecessor pointer per node (pointing to the immediately preceding node on the identifier circle). Upon joining, the following tasks must be performed:

- **Initialize the predecessor and finger table of the new node:** The node n asks n' to compute the successor of each of its finger table entries (ie. of $n + 2^{i-1}$). This would require $O(m\ logN)$ steps. As an optimization, before asking the successor for the i^{th} entry, n checks whether the $i-1^{th}$ finger is also the correct i^{th} finger. This step reduces the number of finger entries to be looked up to $O(logN)$ and hence the total number of steps to $O(log^2N)$.
  As an alternate way of reducing the complexity, the new node can ask its immediate neighbour to share its entire finger table. Most of the finger table entries of its neighbour will also work for the new node and only a few will have to be looked up. This reduces the time needed by n to build its own finger table to $O(logN)$.
- **Update fingers of existing nodes:** Node n can become the i^{th} finger of a node p if and only if:
  - p precedes n by atleast 2^{i-1} on the identifier circle
  - the current i^{th} finger of p succeeds n

  A counter-clockwise walk on the identifier circle can be used to update the finger table entries of all nodes. With high probability, the number of nodes whose finger tables need to be updated when a new node joins the network is $O(logN)$. Therefore, the total time needed for this step is $O(log^2\ N)$.
- **Transfer keys:** n only needs to contact its successor, since it can become responsible only for those keys for which its successor was responsible before n had joined the network.

The paper [2] also presents a way to handle concurrent joins. Inspite of concurrent joins, lookups are still very efficient. This is stated in the following theorem:

Theorem VI.2. *If a stable network of N nodes is joined by N more nodes with no finger entries (except the correct successor), then with high probability, lookups take $OO(log\ N)$ time.*

Node exits can be handled by maintaining a list of $O(log\ N)$ immediate successors. If the node is unable to contact its immediate successor, it tries the next successor in the list and continues till it finds an active node. In particular, the following theorem holds:

Theorem VI.3. *If for every node, we maintain a list of $O(logN)$ immediate successors, the network is initially stable, and every node now fails with 50% probability, then we can still find the successor for any key in $O(logN)$ steps (upon expectation).*

Simulation and experimental results given in the paper [2] verify that the load balance and lookup path length are as obtained by theoretical analysis. In particular, simulations show that the average path length in a N node network is about $\frac{1}{2}log\ N$. This is because, if we consider the binary representation of the distance between the source and the destination nodes, we expect half the bits to be 1 and half the bits to be 0. In order to reach the destination, we only need to follow the finger table entries corresponding to 1's in the distance. Since the representation uses $logN$ bits, we need only $\frac{1}{2}log\ N$ steps to reach the destination.

VII. CAN

In this section, we discuss another p2p routing protocol. This section can be considered to be a summary of [3] and [12]. A Content-Addressable Network (CAN) is another example of a DHT based routing protocol. It has several desirable properties, such as scalability, fault tolerance and building a completely self-organizing network.

A. Basic Design

The basic CAN design revolves around a virtual d-dimensional Cartesian coordinate space on a d-dimensional torus. A d-dimensional torus can be visualized as a d-dimensional solid hypercube with every face joined to / being identical to the opposite face. In other words, we have a d-dimensional coordinate system that wraps around every dimension. The entire coordinate space is dynamically partitioned among all the nodes that are currently present in the system.

In a stable system, every node owns a single zone and there is no overlap between zones. In the transient stage, there might be zones that are not owned by any node or a node might be responsible for more than one zones. Whenever a node joins the system, an existing zone is split. When a node leaves the system, its zone is alloted to another node.

B. Storage (Retrieval)

To store (retrieve) a (key,value) pair, the key is hashed using a uniform hash function (ie. a hash function that distributes keys uniformly in the entire coordinate space), to obtain a point P in the coordinate space. The pair is stored at (retrieved from) the node that currently owns the zone to which P belongs.

C. Routing

A CAN node maintains a small routing table that holds the IP address and the dimensions of the coordinate zone of each of its immediate neighbors in the coordinate space. Two nodes are neighbours if their coordinate spans overlap along $d-1$ dimensions and the spans are adjacent to each other along one dimension. Intuitively, routing in a CAN works by following the straight line path from the source to the destination node in the Cartesian coordinate system. One naive way to route messages is by simple greedy forwarding to the neighbour with coordinates closest to the destination coordinates. In other words, we look at the coordinates of the destination and our own coordinate zone and compute the distance to the destination. We will get a distance in each dimension. In greedy forwarding, we will route the message to that neighbour which will lead to the largest reduction in the total distance to the destination. However, note that other routing policies can also be used. We can proceed to reduce the distance along any of the d-dimensions to reach the destination. We can also first try to reduce the maximum of these distances, then the next largest distance and so on.

For a d-dimensional space that is partitioned into n equal zones, each node maintains $2d$ neighbours (ie. two neighbours per dimension, one in the forward direction and one in the backward direction). The average routing path length is $(d/4)(n^{1/d})$ since each dimension has $n^{1/d}$ nodes and the average distance between two points on a torus along one dimension is $1/4$th of the number of zones. Put another way, on an average, two nodes will be $(1/4)(n^{1/d})$ zones away along every dimension and hence the total distance between them is $(d/4)(n^{1/d})$.

Because of its geometry, a CAN is quite robust to node failures. Since many different paths exist between two points in the space, even if one or more of a node's neighbors crash and the recovery mechanism has not assigned this zone to some other node, a node can automatically route along the next best available path or forward the packet to any node that is closer to the destination than itself, and from there greedy forwarding can resume.

D. Node joins

When a new node joins, three steps need to be followed:

- **Finding an existing node:** By looking up the CAN domain name in the DNS, the node retrieves the IP of a bootstrap node that supplies to it the IP of several randomly chosen nodes that are currently present in the system. In this way, it is ensured that a new node comes to knows about atleast one existing CAN node.
- **Finding a zone:** The new node randomly selects a point P in the coordinate space and sends a JOIN request for P using an existing CAN node. The current occupant of the zone then splits its zone into half and assigns one half to the new node. The splitting is done following a certain ordering of the dimensions so that it is easy to re-merge the zone once a node leaves ie. we decide an order, say d_1, d_2, $\ldots$ in which the dimensions are going to be split. The first split is done in the d_1 dimension and it is remembered. The next split will be done in the d_2 dimension and so on. Once the split has been done, the (key, value) pairs in the new node's zone are transferred to it by the previous owner of this zone.
- **Joining the routing:** The previous occupant of the zone informs the new node about its neighbours. After this, both the new node and the previous occupant update their neighbour tables and inform their neighbors about the splitting of the zone so that they can update their routing tables. In this way, a new node affects only $O(d)$ other nodes, making the CAN system extremely scalable.

E. Node departure, recovery and CAN maintenance

Normally, before leaving, a node should hand over its zone and associated (key, value) pairs to one of its neighbours. If this zone can be merged with the zone of one of its neighbours, this is done. Else, it is handed over to the neighbour which currently has the smallest zone. There is a background zone reassignment algorithm that ensures that there is only one zone per node. So after a certain time, we will again have a single zone (ie. a rectangular box) per node.

Every node sends periodic update messages to its neighbours. Failure of a node is detected by the absence of these messages. If this happens, a takeover mechanism is initiated that ensures that a neighbouring node with a small zone is efficiently chosen.

F. Design Improvements

The bound obtained for the number of hops O($dn^{1/d}$) represents the *application level* hops in a network and not the IP-level hops. Since there is no locality based zone assignment in a CAN, adjacent nodes might be many IP hops away and hence the total latency of routing in a CAN might be many times the actual IP routing latency. To avoid this, various mechanisms are proposed to reduce the total path latency by either reducing the number of hops (path length) or by reducing the per hop latency. Some of these mechanisms are as discussed below.

1) Increasing the dimension: Increasing the dimension of the coordinate space leads to lower path length for a small increase in the size of the routing table. Since each node now

has more neighbours, it also increases the number of paths available between two points and hence the fault tolerance. In particular, making d equal to $logN$ results in a routing table of size $logN$ and a path length of $logN$ as well. This matches the best bounds of various other routing mechanisms such as Chord.

2) Multiple coordinate spaces or Realities: There can be multiple, independent coordinate spaces (or "realities") with each node in the system being assigned a different zone in each reality. This means that a single node will own many zones with each zone belonging to a particular coordinate space. The contents of the hash table can be replicated on each reality, improving data availability and fault tolerance. As before, a node will have $2d$ neighbours for every reality. Since the number of realities is more than one, a node now has a large number of neighbours, and this leads to a substantial reduction in routing path length.

Through simulations, it turns out that for the same number of neighbours, the reduction in path length achieved by increasing the dimension is much more than that obtained from increasing the number of realities. However, increasing realities offers the advantage of data replication and better fault tolerance.

3) Better routing metric: To improve latency, every CAN node can be required to measure the network level round trip time (RTT) to each of its neighbours. A message can now be forwarded to that neighbour with the maximum ratio of progress to RTT. Although this might lead to increase in the number of hops, it leads to a reduction in the total path latency. As we increase the dimension, we get more routing choices and hence the improvement obtained by using this optimization increases.

4) Overloading coordinate zones: Multiple nodes can be allowed to share the same zone subject to a maximum of MAXPEERS nodes per zone. The nodes that share a zone can keep a copy of the same information or split the information. A node needs to know all peers in its zone only and not all peers in its neighbours zone. Out of all the peers (belonging to a single zone) that can act as a neighbour node to a particular node A, it chooses that one which has the smallest RTT. Overloading leads to reduced path length (since the number of zones goes down), reduced per-hop latency (since the minimum RTT node is chosen as the neighbour), improved data availability (if information is copied) and better fault tolerance (since even if one node crashes, the zone does not become vacant).

5) Multiple hash functions: k different hash functions can be used to map a single key to k different points and the (key, value) pair can be stored at each of these points. This improves data availability. Moreover, a query can be sent to each of these points in parallel to reduce the total latency. However, multiple hash functions result in increasing the amount of storage and query traffic (if parallel queries are made) by a factor of k.

6) Locality based zone assignment: If instead of assigning a new node a random zone, we assign it a zone in such a way that its neighbours are close to it ie. they have a low latency, we can gain significant improvements in performance. A reduction in the per hop latency would lead to a lower total path latency. This optimization can be achieved by dividing the coordinate space into some segments and assigning a zone to a node depending on its location (measured using the RTT to a fixed set of routers).

7) Uniform Partitioning: If a node, upon receiving a JOIN request, instead of directly splitting its own node, compares its volume to that of its neighbours and that zone is split which has the highest volume, a much more uniform partitioning of the space is obtained (this is verified using simulation). This leads to better load balancing and reduced variance in the total path length, amount of data stored and the number of neighbours for a node.

G. Problems with CAN

Currently, there are some open problems that need to be addressed before commercial applications using a CAN can be realized. One of the major problems is to make a CAN system resistant to denial of service attacks. This is difficult, since a malicious node can act as either a client, a server or a router.

Another problem is to modify the algorithm for being able to support keyword based search.

VIII. The Impact of DHT Routing Geometry on Static Resilience and Proximity

Till now, we have seen two DHT based routing protocols - Chord and CAN. The Chord protocol has an underlying ring geometry, while the CAN protocol is based on a hypercube geometry. There are DHT based routing protocols that use still other geometries like butterfly networks, tree structures etc. Here, we take a look at how the basic routing geometry of a protocol affects its static resilience and proximity properties. The title of this section has been taken from [4] and this section is essentially a summary of that paper.

As we will soon quantify, these basic routing geometries differ in the amount of freedom (or *flexibility*) that they provide in the choice of neighbours (ie. the various ways in which the routing table of a node can be chosen) and in the choice of a routing path (ie. given a routing table, in how many ways can we make a next-hop in order to route the packet to the destination). It must be noted that inspite of differences in the flexibility that these geometries provide for neighbour and route selection, they all support a key lookup in $O(logN)$ steps, provided we have a routing table of $O(logN)$ entries (except in the case of a butterfly network, which only needs a small constant size routing table). Therefore, we must distinguish between these geometries based on other performance metrics such as static resilience and proximity (described below).

Stated more precisely, flexibility is the algorithmic freedom left once the basic routing geometry has been chosen. Also, while looking at a particular routing geometry, we must not look at the amount of freedom that we have in neighbour and route selection for a particular routing protocol based on that

geometry (eg. CAN in the case of a hypercube geometry). Instead, we must look at how the geometry itself puts bounds on the number of choices that we have. This is because, there can be many routing protocols that are based on the same underlying DHT geometry, but differ in the way they do things. In the analysis that follows, we will come across what are know as *sequential neighbours*. These are neighbour nodes to which one can route a packet and be sure that progress has been made for all possible destinations eg. the immediate successor node in the chord protocol (described in section 6) is a sequential neighbour. We now quantify the flexibility afforded by the various routing geometries mentioned above.

A. Flexibility in neighbour and route selection

In each of the following geometries, we will assume that all nodes are in a $logN$-bit identifier space ie. every node has a unique $logN$-bit identifier. Also, while considering the flexibility in route selection for a given routing table, we are mainly interested in finding the number of paths that allow us to do efficient routing ie. in $O(logN)$ steps.

1) Tree: In a tree geometry, we have a binary tree of height $O(logN)$ and all the nodes in the network can be viewed as leaves in this tree. The distance between any two nodes in a tree network is defined as the height of their smallest common subtree. Since the height of the tree is $O(logN)$, the maximum distance between two nodes is $O(logN)$. The routing table for a node consists of $O(logN)$ entries, where the i^{th} entry is for a node that is at a distance of i from the current node. Therefore, a node at a distance of i from the current node will have its first $logN - i$ bits same as that of the current node, and will differ in the i^{th} bit from the end. Routing in a tree network is achieved by successively correcting the highest order bits. So, we look the highest order bit at which the current node differs from the destination node. If this is the j^{th} but from the end, we send the packet of our j^{th} neighbour ie. to the node that differs from the current node at the j^{th} bit (from the end). In this way, one bit is corrected and we continue like this until we reach the destination node.

Now we look at the flexibility in the above design. A subtree of height i has 2^{i-1} leaf nodes. Therefore, a node has 2^{i-1} choices for selecting its i^{th} neighbour (all these nodes have their first $logN-i$ bits the same as the current node and differ on the $logN-i+1^{th}$ bit. Therefore the total number of ways in which a node can select its neighbours (ie. the number of different routing tables that it can have) is given by:

$$\prod_{i=0}^{logN-1} 2^i = 2^{(logN)(logN-1)/2} = n^{(logN-1)/2}$$

However, once we have chosen a particular routing table out of all the above options, there is a only a single node that will increase the longest prefix match to the destination. Therefore, there is no flexibility in the choice of routes.

2) Hypercube: In a hypercube geometry, once we have been given an identifier, we have no choice in the selection of neighbours. Therefore the hypercube geometry offers no flexibility in the choice of neighbours.

For a given choice of neighbours (ie. for a particular routing table), we have a number of paths that lead to the destination. We can choose to correct the bits along any dimension first. In other words, we have a choice in the order in which we correct the bits ie. the order in which we choose the dimensions. We consider the case when the hypercube is of dimension $logN$, since then we can route in $O(logN)$ hops and each node has a routing table of size $O(logN)$ (and hence we can compare this to other geometries which have similar parameters). In this case, the first node has $logN$ choices for the next hop, since it can choose to correct the distance along any dimension first. The second node has $logN-1$ choices and so on. Therefore, the total flexibility in route selection is $(logN)!$

3) Butterfly: We look at the butterfly geometry in the light of a particular routing protocol that is based on it - Viceroy [13]. From this we get the flexibility properties of such a routing geometry. For realizing a butterfly network, Viceroy imposes a global ordering on nodes and designates nodes to different levels (to correspond to levels in the butterfly network). We only need to have a small constant size routing table consisting of the immediate successor and predecessor nodes from the immediately preceding and following levels. In addition, we store successor and predecessor pointers to nodes in the same level. Using this routing table, we can achieve routing in $O(logN)$ hops. This is done in three steps. In the first step, we reach a node at level 0 (this will take $O(logN)$ hops). In the second step, we reach a node in the same level as the destination node in another $O(logN)$ hops. We now use the successor and predecessor pointers to reach the destination node in another $O(logN)$ hops. Since each step requires $O(logN)$ hops, the complete lookup algorithm also needs $O(logN)$ hops.

Because of the geometry that Viceroy is based on, there is no flexibility in either the choice of neighbours, or in the choice of routes to a destination.

4) Ring: In a DHT routing algorithm based on ring geometry (such as Chord), a node maintains information about at least one other node that is at a distance between 2^{i-1} and 2^i from it (provided such a node exists). By using this table, routing can be achieved in $O(logN)$ hops. Chord stores the first node that is at a distance greater than 2^{i-1} from it. However, routing in $O(logN)$ hops can be achieved by using any node in this range. The argument of Theorem VI.1 still goes through without any modification. Therefore, the number of ways in which we can select the i^{th} neighbour in a ring geometry is 2^{i-1} (same as in the tree geometry). And so the flexibility in neighbour selection is $n^{(logN-1)/2}$.

In a ring geometry, to reach a destination from a source node, we need $O(logN)$ hops. These can be considered as jumps. In Chord, we do greedy routing and try to take the largest possible jump at every step that will not make us overshoot the destination. However, for routing in these many hops, it must be noted that the order in which we make the jumps does not matter, as long as we take all the jumps. Since we have $O(logN)$ jumps, the number of different ways in which these can be taken is $O(logN)!$. Therefore, the choice

in route selection offered by a ring geometry is $logN!$.

In addition to the $logN!$ possible choices that do not lead to an increase in the path length, there are also other paths much longer than $O(logN)$, that can be taken to reach the destination. This can be understood in terms of Chord as follows. For two nodes that are at a distance of $O(n)$ from each other, there are approximately $O(logN)$ entries in the routing table of the source node that lead to progress in the direction of the destination node. We can choose any of these neighbours to make progress towards the destination. Some of these paths will be optimal, while some will not be optimal.

5) XOR: Kademlia [14] is yet another DHT based routing algorithm. It defines a new distance metric - the distance between two nodes is the numeric value of the XOR of their identifiers. A node has a routing table of size $O(logN)$, where the i^{th} entry is a node that is at a distance in the range $[2^{i-1}, 2^i]$. Therefore, like the tree and ring geometries, the XOR geometry offers a total of approximately $n^{(logN-1)/2}$ choices of routing tables.

In the normal case, routing takes place exactly as in the tree geometry. We calculate the distance between the source and the destination nodes and try to fix the MSB. At the next hop we recompute the distance and again fix the MSB and so on. To fix the i^{th} bit (that is the MSB), we choose the i^{th} neighbour. Since the i^{th} neighbour is at a distance in the range $[2^{i-1}, 2^i]$, in the tree geometry, it is at a distance of i from the current node. In tree geometry, routing works by reducing the distance to the destination in every step and if the distance to the destination is i at the particular step, we choose the i^{th} neighbour. Therefore, if the i^{th} neighbours in the tree and XOR geometries are the same, the path followed will be exactly the same. Therefore, there is no flexibility in route selection in the XOR geometry, just like in tree geometry. However, in case of failures, the two geometries differ. In the tree geometry, if the distance to the destination from the current node to the destination is i, and the i^{th} neighbour fails, there is no way in which we can route the packet to some other node and make progress towards the destination. However, in XOR geometry, we can route the packet to a node that results in the correction of some lower order bit(s). In this way, we will still make progress towards the destination. However, the number of hops that are needed now might be much more than $O(logN)$, since now when we correct a higher order bit, a corrected lower order bit might again become incorrect.

B. Static Resilience

One of the reasons behind DHT's (and p2p architectures in general) being seen as an excellent way to build large scale distributed systems, is that they are resilient to changes in the system ie. they can handle node failures gracefully. Resilience has three main concerns:

- **Data Replication:** In this paper, it will be assumed that there are enough copies of a data item in the system, so that a node exit does not result in data loss. The question that will be addressed is whether routing can be done in the presence of failures.
- **Routing Recovery:** To take care of failures, we need recovery algorithms that will periodically fix the routing tables of the active nodes. All the DHT based routing algorithms have (or should have) a background recovery algorithm that, depending on the protocol, fixes the routing tables. Therefore, in this paper, we will not distinguish between the various approaches used by recovery algorithms.
- **Static Resilience:** There is a time gap between the system experiencing node failures and the recovery algorithm being able to fix the routing tables of all active nodes. Static resilience is a measure of how well can the system route in this time gap. It is important for the robustness and usability of a system. Moreover, it gives an idea of how fast the recovery algorithms need to fix the routing tables.

In this report, we will concentrate on static resilience. From simulation results, we conclude that the Tree and Butterfly networks, that offer no flexibility in route selection, perform very badly in the event of node failures. When 30% of the nodes in the system fail, then almost 90% of the paths fail (ie. routing can't take place in 90% of the cases). Ring and Hypercube, that offer the greatest flexibility in route selection, perform quite well and even when 30% of the nodes in the system fail, only 7% of the paths fail. The XOR geometry, which has some routing flexibility, perform in between the previous extremes and here around 20% of the paths fail when 30% of the nodes in the system fail.

On carrying out the simulation with the addition of sequential neighbours, the results greatly improve. A simulation carried out by adding 16 sequential neighbours resulted in no path failures in any geometry, when 30% of the nodes in the system failed. However, this resilience comes at the cost of increased path length. Also, there is still a marked difference between the resilience properties of ring and hypercube geometries, as compared to other geometries. There is also a difference between the ring and hypercube geometries. This is expected, since a ring is the only geometry that can support sequential neighbours naturally. For other geometries, artificial mechanisms have to be deployed, for being able to use sequential neighbours.

Simulations also show that when there are a large number of node failures, then adding sequential neighbours leads to better resilience properties as compared to simply increasing the number of regular neighbours of a node. However, this comes at the cost of much larger path lengths. Therefore, sequential neighbours must be used when we are only interested in routing success and not concerned about total path latency.

Therefore, it can be concluded that *the static resilience of a routing geometry is largely determined by the amount of routing flexibility that it offers.*

C. Path Latency

DHT routing algorithms are designed to do efficient routing by reducing the number of hops that are needed for a lookup. However, we must note that these hops take place in the identifier space. Two nodes that are neighbours in the identifier space, might be very far off in the underlying internet topology in terms of the path latency. Therefore, for efficient routing, we now want to reduce the total end-to-end path latency. This can be done in three ways:

- Proximity Neighbour Selection (PNS): We choose the neighbours (ie. our routing table) depending on their proximity. We define *proximity* between two nodes as a measure of the IP latency between them.
- Proximity Route Selection (PRS): For a particular routing table, to send a packet to a particular destination, we want to choose that neighbour for the next-hop that leads to a lower total path latency.
- Proximity Identifier Selection (PIS): We choose node identifiers based on the location of the nodes. However, this leads to problems of load balancing and correlated failures and is hence not considered here.

We now look at PNS and PRS as ways to reduce the total path latency. In DHT geometries that offer flexibility in neighbour selection, for PNS we would like to choose that neighbour that is the closest in terms of IP latency. This is the ideal PNS algorithm. However, this is not always possible/practical, since the number of choices might be too mant. Therefore, as an approximation, we look at the first K choices in the candidate set for selecting the neighbour. Out of these K nodes, we pick that node as the neighbour which is the closest in terms of proximity. This algorithm is known as the PNS(K) algorithm.

Having a good PRS algorithm for selecting the next hop is much more difficult. There is a tradeoff in the latency and the number of hops. If we seek to reduce the total number of hops (as was being done earlier), then it might lead to a huge path latency. On the other hand, if we try to reduce the latency by routing the packet to the neighbour with the smallest latency (that leads to progress towards the destination), it might result in a large number of hops with small per hop latency, but a large total path latency. We follow some simple heuristics for PRS depending on the geometry.

In a hypercube geometry, all paths are of equal length. Therefore, from among the possible next hops, we simply choose the one with the smallest latency. In the ring geometry also, we use the fact that there exist multiple paths with the same number of hops. We do this by first computing the distance (in binary) between the source and the destination. We now look at all the bits that are 1's. We now make a candidate set consisting of all the neighbours that corresponding to these bits ie. if there is a 1 in the i^{th} position, we add the i^{th} neighbour to the candidate set. We now route the packet to the neighbour with the least latency among these candidate nodes. However, we cannot apply these kind of heuristics to the XOR geometry, since there we do not have multiple paths with the same number of hops. So the heuristic used in XOR geometry is to take a non-greedy next hop only when the difference between the latency in the hop choices (ie. between the higher latency next hop that does greedy routing and the lower latency next hop that leads to potentially more number of hops) is greater than the average hop latency in the network.

1) Results: In this section, we compare the improvement achieved due to PNS and PRS. Only the XOR and the Ring geometry can use both PNS and PRS. For these geometries, simulations show that a tremendous improvement in the latency is achieved by using any of these algorithms. However, the improvement achieved by using the ideal PNS algorithm is much more than that obtained by using the PRS algorithm. Moreover, the improvement achieved by using both PNS and PRS is only slightly more than that achieved by using just PNS.

This is in fact expected, and can be understood as follows. Suppose we have a ring geometry, and we want to reach a node that is at a distance in the range $[2^i, 2^{i+1}]$. The ideal PNS algorithm picks a neighbour from amongst 2^i choices. The PRS algorithm works on a fixed routing table and hence has only i choices for the next hop (essentially, it can route to any of the first i neighbours). Since we have many more choices in the case of the ideal PNS algorithm, its performance is expected to be much better.

Addition of sequential neighbours to the above geometries, leads only to a moderate improvement in the latency performance (that too only in case of the normal algorithm and the PRS algorithm), as compared to the case without sequential neighbours. Therefore, sequential neighbours do not affect the difference in performance between the normal algorithm, the PRS algorithm and the ideal PNS algorithm.

From the above results, it is clear that in order to reduce path latency, the geometry must have the ability to support PNS. Support for PRS is not that essential. We also observe that the improvement in performance obtained by using either PRS or PNS or both is irrespective of the underlying geometry. For example, the performance improvement obtained in both the hypercube geometry and the ring geometry by using PRS is approximately the same.

Simulations show that the proximity methods described above can reduce the end to end path latency to a very small multiple ($\sim$2) of the underlying internet latency.

IX. Conclusion

Recently, there has been a lot of academic and commercial activity in the domain of peer to peer systems. p2p systems offer desirable properties like extreme scalability, robustness and fault tolerance. However, they also have certain problems such as latencies, inconsistency etc. Earlier, research took place on unstructured p2p systems such as Napster, Gnutella etc. However, now the focus of research has shifted onto structured overlay networks. Also, people have started looking at the security issues in p2p systems.

The p2p paradigm has enabled a new class of distributed applications, and reshaped the way traditional applications

work. As we move towards a better connected world, more and more applications will make the shift to a p2p system. In short, peer to peer systems are the way of the future and are here to stay.

ACKNOWLEDGMENT

I would like to thank Prof. Dheeraj Sanghi for allowing me to undertake this reading project as my CS625 (Advanced Computer Networks) course project.

REFERENCES

[1] M. Roussopoulos, M. Baker, D. S. H. Rosenthal, T. Giuli, P. Maniatis, and J. Mogul, "2 P2P or Not 2P2P," in *IPTPS*, 2004.
[2] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications," in *SIGCOMM*, 2001.
[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *SIGCOMM*, 2001.
[4] K. Gummadi, R. Gummadi, S.Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The Impact of DHT Routing Geometry on Resilience and Proximity," in *SIGCOMM*, 2003.
[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," in *Workshop on Design Issues in Anonymity and Unobservability*, 2000.
[6] Napster. [Online]. Available: http://www.napster.com
[7] A&M Records v. Napster. [Online]. Available: http://www.gseis.ucla.edu/iclp/napster.htm
[8] KaZaa. [Online]. Available: http://www.kazaa.com
[9] D. Karger, E. Lehman, T. 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," in *STOC*, May 1997.
[10] D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," Master's Thesis, Department of EECS, MIT. 1998.
[11] Gnutella. [Online]. Available: http://www.gnutella.com
[12] S. Ratnasamy, "A Scalable Content-Addressable Network," PhD Thesis, University of California, Berkeley. October 2002.
[13] D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: A scalable and dynamic emulation of the butterfly," in *PODC*, 2002.
[14] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric," in *IPTPS 2002*, 2002.
[15] K. Aberer and M. Hauswirth, "P2P Information Systems," in *ICDE*, 2002.
[16] J. M. Hellerstein, "Architectures and Algorithms for Internet-Scale (P2P) Data Management," in *VLDB*, 2004.