Peer-to-Peer Networks: An In-depth Study

Anandha Gopalan
Department of Computer Science
University of Pittsburgh, PA 15260
axgopala@cs.pitt.edu

Abstract

Peer-to-Peer computing has caught on in the last few years, thanks in part to the emergence of various file sharing programs. Although, peer-to-peer networking is not a new concept, interest has been rekindled in this field. This has lead to a plethora of ideas for solving the various issues with peer-to-peer networks and also to try and find out how to effectively utilize this infrastructure.

This survey investigates the field of peer-to-peer networking and summarizes the key concepts and ideas. An overview is provided for many of the new technologies and projects in this area. This paper will be helpful in making people understand this growing technology. This paper is targeted to both the novice reader who is not very familiar with the topic (this will help him/her understand the basic concepts and get familiar with peer-to-peer networking) and also for the reader who is well versed in this topic (the different models are described in sufficient detail, thus helping the reader grasp the breadth of ideas).

Keywords: Peer-to-Peer, P2P, routing, networks, keys, hashing, distributed, centralized, location.

1 Introduction

The advent of file sharing programs like Napster [18], KaZaa [13] and Morpheus [17] has resulted in catapulting peer-to-peer networking into the news. The peer-to-peer technology is nothing new, servers (e.g: News) cooperate in peer-to-peer manner to exchange information. This technology has received a boost due to the improvement in several factors, namely: high bandwidth, inexpensive computing power, low memory cost and high storage capacity. This in turn leads to a host of new opportunities for distributed computing, file sharing, information sharing and discovery.

Peer-to-Peer networking has no formal definition, but the general consensus is: "Peer-to-Peer networks and systems are a class of distributed nodes without any centralized control, where each node performs its task individually and every node in the system has the same functionality".

Peer-to-Peer systems have a lot of advantages provided by the distributed architecture. Since nodes share the resources, an application that was constrained due to the lack of resources can now be executed by using the resources offered by another *peer*. An example of this is the SETI@home project [28] (this is covered in more detail in section 3.4). Maintenance costs are lower due to the fact that nodes can be easily replicated using the existing infrastructure. Information retrieval can be made more effective by having dedicated P2P groups for different topics, for e.g: A doctor researching about the SARS disease can become a part of a SARS P2P network (one wherein people have similar likes), thus gaining access to information faster. Also, this information is more specific and is easier to browse through. *Hot spots* on the web can be alleviated, by having *peers* serve the data instead of having the client retrieve the data from the same server every time.

The advantages in P2P networks however comes at a price. The biggest concerns facing the P2P research community are: security, user anonymity, efficient data location, robustness and routing in the overlay network. Nodes have to interact with one another and share each others resources to achieve the needed goal in a P2P network which raises some security concerns for users. User anonymity is also important, since we do not want users in the network to know about each others' identities, this could lead to differential service. Consider the case when a merchant in Minnesota is distributing a product to customers in New York, noticing this another merchant in New York can start advertising his/her products at a lower price and can *claim* to deliver them faster as *topologically* he/she is closer. A P2P network must be robust so as to avoid network crashes, it should also be robust against DoS attacks and malicious nodes. Data location and routing are key issues in P2P networks, since data is what we would ultimately need (be it music files, data files, stock quotes, movie show times or application data).

This survey paper is designed to provide an insight into this exciting technology. The rest of the paper is organized as follows. Section 2 gives a brief overview about P2P networks and the various types of P2P networks. Section 3 lists some of the P2P systems along with a detailed explanation of each. Section 4 concludes the paper.

2 Overview

This section provides an overview into P2P networking. The different models of P2P networking are studied in detail. Before we delve into the details of different P2P models, we need to define a few terms:

Definition 2.1 Client: *The* client *is a computer system (or a process running on that system) that requests a service of another computer system (or process running on that system). This service is requested through some kind of a network protocol. The client waits for the response before it continues execution.*

Definition 2.2 Server: *The* server *is a computer system that provides some service for other computers.* Clients *request this service from the server and the server responds using some kind of networking protocol.*

Definition 2.3 Servent: *The* servent *is a computer system that is capable of functioning both as a* server *as well as a* client*. Servent is derived from* SERver *and* cliENT*.*

Definition 2.4 Node: *In the context of a P2P network, any computer system is referred to as a* node*. This* node *may act as a* client*,* server *or a* servent*.*

In this paper, will will use the term node for any generic (client, server or servent) P2P computer system.

The different frameworks of peer-to-peer networks that are studied are: centralized framework, decentralized framework, controlled-decentralized framework and the distributed hash table.

2.1 Centralized Frameworks

This framework follows the traditional client-server model of communication, with all clients requesting a service from a central server. A user who wishes to join the P2P network connects to the central server where he/she either uploads information to the server or downloads information from the server or both. Once this request/reply has completed, the client can request for some service from the server and the server responds with a list of nodes that can perform that service. The client can then connect to one of the peers *directly* and request for the service. Figure 1 shows this kind of a network.

The advantage of such a scheme is that the point of control is with the central server, which means that this server can police the requests coming through. This method is very scalable and robust. Just by adding some extra servers, we can scale the system as well as make it robust by adding data redundancy. Searching in this framework is very fast as the central server has the list of files currently available on the network.

The disadvantage of this scheme is that the central server becomes a single point of failure. If the central server is absent, then this network fails completely.

An example of a centralized framework is Napster [18].

2.2 Decentralized Frameworks

This framework does not have a central server, instead each node acts as a *servent*. To join this P2P network, each user has to initially connect to at least one node that already belongs to the P2P network. Once this connection has been established, then the state of the network is updated by flooding the routing messages through the network. An example of a decentralized network is shown in Figure 2.

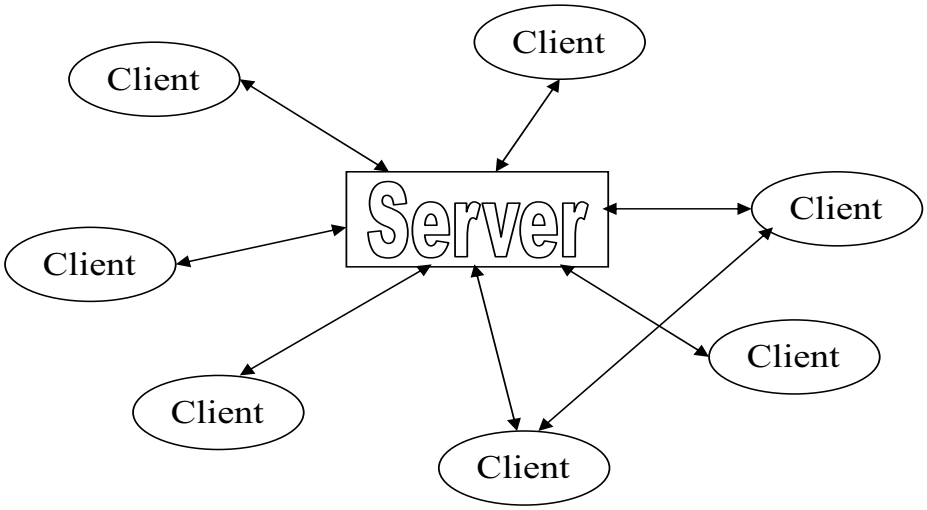

Figure 1: Illustration of Centralized Framework

The advantage of this scheme is there is no single point of failure; even if a node or many nodes crash, the network will still be up.

The disadvantage of this framework is that searches usually take a longer time than that of a centralized network as the search query has to traverse through the network. Once a result is found, the node can directly request the file from the *peer*. There is no control over the files that are shared on this network as there is no entity that can monitor this.

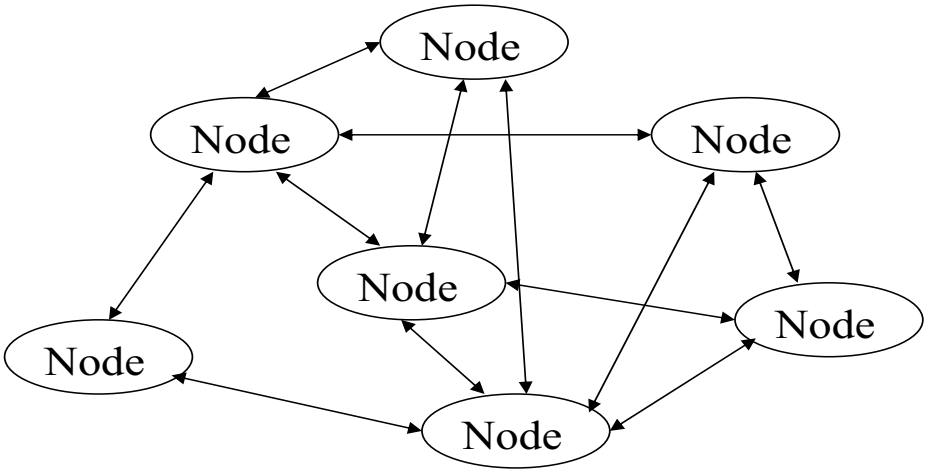

Figure 2: Illustration of Decentralized Framework

Examples of decentralized frameworks are: Gnutella [11] and Freenet [6].

2.3 Controlled Decentralized Frameworks

This scheme combines the salient features of both centralized as well as decentralized frameworks. The disadvantage in the centralized scheme was that the central server was a central point of failure, while in the decentralized framework the disadvantage was the time taken to search. This scheme uses the

knowledge about the nodes in the system, any node with sufficient computing and bandwidth capabilities is given the status of a *supernode* in the system. The *supernode* acts as a central server, but it is not the only supernode in the framework. There are other supernodes in the framework too. Anytime an user wants to join the network, he/she searches for a supernode in the framework and connects to it. Searches are directed to the supernodes, if a supernode does not have the result to a query, it is forwarded to the next supernode. This way flooding messages through the network is avoided. Once a result is received, the user fetches the file directly from the *peer*. An illustration of this scheme is shown in Figure 3.

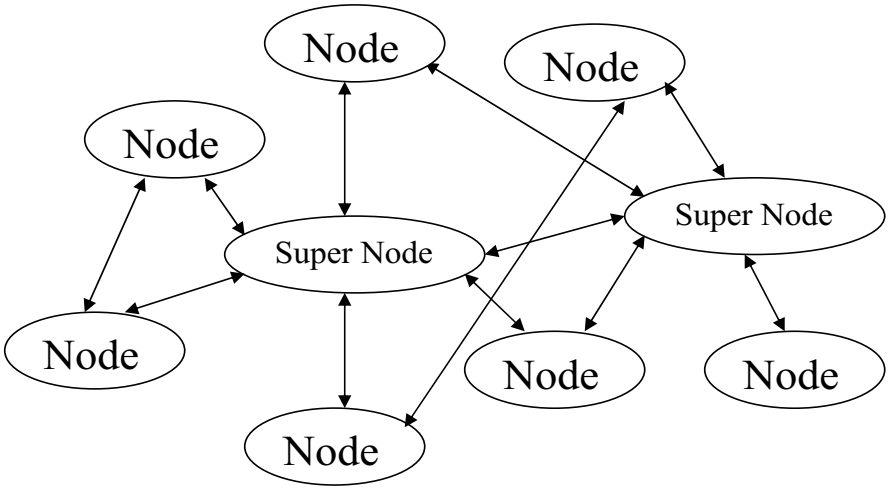

Figure 3: Illustration of Controlled Decentralized Framework

The advantage of this scheme is that it is very scalable, since messages are mainly routed only between the supernodes. It is also very robust, as even when a supernode fails, the network will be up and running, and after a certain amount of time another node (with sufficient resources) will be chosen as the super node.

The disadvantage with this scheme is that it still takes a fair bit of time for searching. Also, nodes can decline to be supernodes if they want to making the choice of selecting a supernode non-trivial if a supernode fails.

The FastTrack protocol is an example of this scheme and Morpheus [17] and KaZaa [13] are implementation based on this protocol.

2.4 Distributed Hash Tables (DHTs)

While the centralized framework scheme is very scalable, it suffers from the fact that the server is a single point of failure. The distributed framework schemes are not very scalable due to the high increase in network traffic that is caused by search queries. A Distributed Hash Table data structure has been suggested by researchers to fix the problem of scalability. In this scheme, just as in a real hash table, files are associated with keys (an example of associating a key to a file is shown in Figure 4). These keys are spread around the different nodes in the system with each node being responsible for storing certain range of keys (based on the algorithm used).

Every DHT supports one basic operation *lookup (key)*. Given a key, this function returns the identity of the location of the key (most probably the IP address of the node that holds the key). The problem of scalability now reduces to a problem of effectively placing and retrieving the keys in this context. Hence, routing messages efficiently between the node that issues the query *lookup (key)* and the actual node that holds the key assumes paramount importance. The scalability and performance of a DHT is thus directly related to the routing algorithm employed.

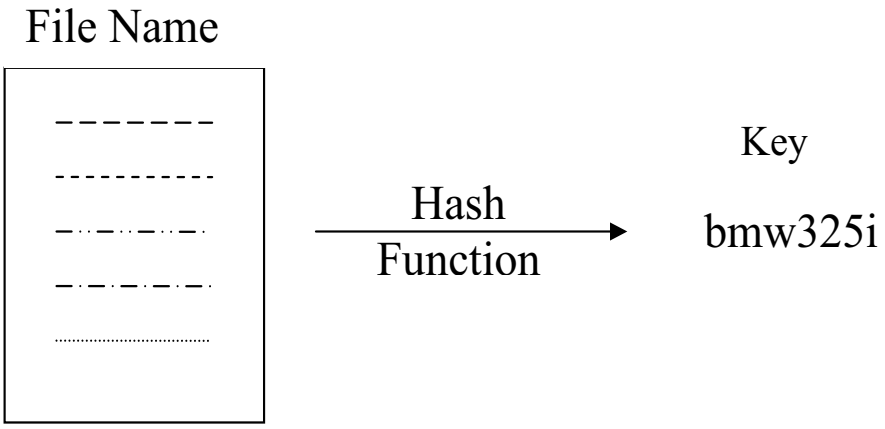

Figure 4: Example of obtaining key from a filename in a DHT

Examples of P2P systems that use a DHT are: Chord [29], Tapestry [31], Pastry [25] and Content Addressable Networks (CAN [21]). Some of these systems have lead to the development of large scale P2P networks based on the underlying DHTs, like large scale distributed systems and chat services.

3 P2P Systems in practice

This section gives an in-depth study on some contemporary P2P systems available. The chosen set of systems covers the entire range of frameworks discussed in section 2. Each system is analyzed based on following important criteria: Bootstrapping, node behavior (node joining, node leaving, nodes failing, malicious nodes in the system) and searching for required objects in the system.

3.1 Napster

3.1.1 Overview

Napster [18] was born in January 1999, when Shawn Fanning, a freshman at Northeastern University wrote an application that would allow the students in his dormitory to share music amongst them. In May 1999, Napster Inc. was formed and at its peak had as many as 21 million users. Napster was ultimately sued by the record industry in America for copyright infringement. Napster is still being used today, but only a fraction of the users use it anymore.

Napster is based on the centralized framework architecture. To access Napster, an user connects directly to the Napster server. He/she then issues a query (the name of a song) and the server returns a list of clients that currently have that song. The user then connects to the preferred client and retrieves the song directly from the client. Once the list is returned, the interaction with the server is not necessary. If the client wants another file, he/she has to issue another query to the main server. A sample interaction between the Napster server and client is shown in Figure 5. The important thing to notice out here is that the file never resides on the Napster server, nor does the server actually see the file. The file is directly transferred from another client to the user.

3.1.2 Bootstrapping

Achieving bootstrapping in Napster is easy as there is only one central server. The IP address of the server is available as part of the Napster package and hence the client can easily connect to the server

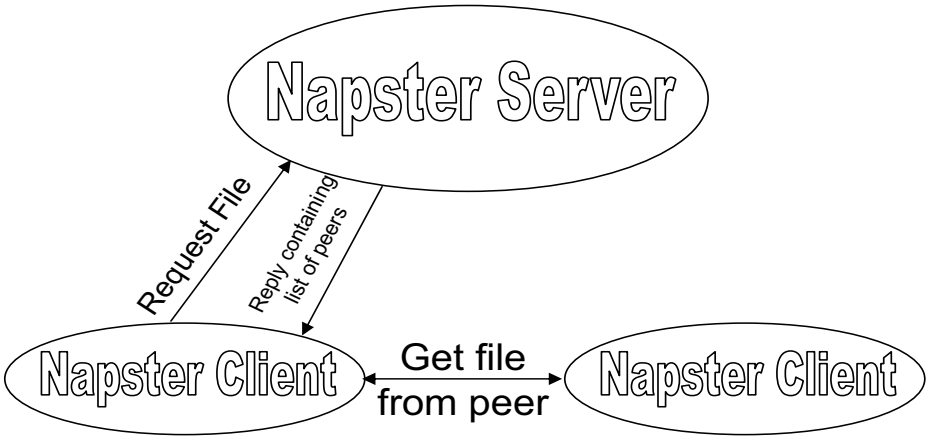

Figure 5: Illustration of the process of getting a file in Napster

directly.

3.1.3 Node Behavior

Once a client connects to the server the first time, Napster keeps track of this node till the node either dies (in which case, it is detected by a timeout) or the node sends an explicit *logoff* message.

Malicious nodes can cause a problem with Napster as the server is a single point of failure. DoS attacks can occur with Napster.

3.1.4 Searching

When a client connects to the Napster server, it uploads a list off all the music files that it has in its database. The Napster server immediately updates its database with this information. The music files are indexed on the Napster server for fast lookup to client queries.

3.1.5 Salient Features

Napster was partly responsible for a rekindle of interest in peer-to-peer networking today. It describes an architecture which is robust, fault tolerant and inherently scalable.

3.2 Gnutella

3.2.1 Overview

The Gnutella protocol was initially developed by Nullsoft [19] (the company responsible for the WinAmp MP3 player [30]). This protocol was based on a completely decentralized framework with no central server. The project was abandoned after only a month, but not before this had been downloaded. Over the next few months this protocol was reverse engineered and soon enough Gnutella clients started appearing, the most popular ones being LimeWire [15] and BearShare [4].

3.2.2 Bootstrapping

Before a node can join a Gnutella network [11], it must be able to connect to at least one node that is part of the network. Discovering this node is not easy, hence the implementations provides a list of hosts

along with the application so that the application can try and connect to one of the hosts mentioned in that list. Once the node manages to connect to one of those nodes, it becomes part of the protocol.

3.2.3 Node Behavior

Every time a node joins into the network, the node that it connects to sends out a broadcast message to all its neighbor nodes about the existence of this node. This broadcast continues on till the packets' TTL (Time-To-Live) becomes zero. Usually, the packet starts off with a TTL of 7.

Nodes in the Gnutella network do not explicitly need to logoff from the network, their disappearance is noticed once a neighboring node does not receive a response to its *ping* message. If a node is alive, it usually replies to a *ping* message with a *pong* message of it own. Since, route discovery and maintenance is done using broadcasting, the failure of nodes does not affect this protocol adversely, as it can always find routes to route around the failed node.

Malicious nodes can create a problem in the Gnutella network. These nodes can create false search queries that just flood the network and achieve no real purpose, thus consuming valuable network resources.

3.2.4 Searching

When a node needs to search for a particular file, it sends out a search query to all its neighbors, who in turn send out the query to all their neighbors. This continues on till the TTL of the query packet becomes 0. The packet starts off with a TTL of 7. Since, this broadcasting is slow, results usually take a while. Once the list of nodes is returned, the client can choose a node at its discretion and retrieve the file directly from that node.

This searching algorithm is clearly not scalable as it can lead to a lot of traffic on the network. A study showed that for a simple query string like "grateful dead live" with a TTL of 7 and 8 neighbors per client, the amount of traffic generated was close to 90 MB [24].

3.2.5 Salient Features

The appealing thing about the Gnutella protocol was that any type of file could be shared, unlike Napster which only allowed the sharing of music files. Also, the routing algorithm is very localized, each node looks at the TTL on the packet and then routes the query to all its neighboring nodes. This offers a small degree of anonymity as the node receiving a packet does not know the node that originated the query. This does not work if a node receives a packet with the TTL set to 7, since the node that received this packet would be the first node after the node where the packet originated.

3.3 FastTrack

3.3.1 Overview

This design follows the controlled decentralized framework. Both KaZaa [13] and Morpheus [17] are implementations that follow this framework. This architecture combine the salient features of Napster and the salient features of Gnutella. These file sharing software are very popular and in great use today.

3.3.2 Bootstrapping

When a node decides to connect to this network, it first has to connect to a supernode of this network. A supernode is a node that have better resources in terms of bandwidth and processing power. Supernodes can connect to other supernodes and can also accept connection from nodes trying to join the network. As with Gnutella, a list of supernodes is provided along with the implementation. The user can also download this list and then can try and connect to the supernode. The supernodes in this protocol act similar to the central server in Napster.

Once the user has successfully connected to the supernode, he/she becomes a part of the network. The supernode updates this information to the other nodes that are connected to it. It also passes this information to other supernodes. This causes considerably less number of routing messages than Gnutella, since only the supernode is responsible for sending routing messages.

3.3.3 Node Behavior

Every time a node joins into the network, the supernode takes care of sending routing updates to all the other nodes that are connected to it. The node must also be authenticated by a central KaZaa server. The list of supernodes can also be retrieved after this authentication (this depends on the implementation).

Nodes do not need to explicitly logoff in this network, node failures are detected when a node does not respond to the routing messages that are sent periodically. If a normal node fails, then it does not affect the protocol, since supernodes are the ones responsible for maintaining the routing information. If a supernode fails, then all the other nodes wait for a time out before electing another node as a supernode. Nodes have the option of not accepting to be supernodes. In case, after a time interval a supernode has not been chosen, then these nodes try and connect to other supernodes so as to stay as part of the network.

Malicious nodes can insert false query messages into the network, but since the amount of messages passed in this case if not very much, this does not adversely affect the protocol. A malicious node can take on the responsibilities of a supernode and then fail, thus causing a disruption to the network. This is not much of a concern as the node would not have been functioning as a supernode for too long. The reason for this is that the resources of a supernode are used quite extensively, so to cause real havoc in the network, the supernode must have been established for a while, which a malicious node would not do for fear of having its resources utilized.

3.3.4 Searching

When a node searches for a particular file, this query is forwarded to the supernode to which it is connected. The supernode maintains a list of the files that are contained on all the nodes that are connected to it (while joining the network, nodes are required to upload the list of files in their database). The query is also forwarded to other supernodes, which in turn forward it. This stops once the TTL becomes zero. Once the results are received, the node chooses the client to connect to so as to retrieve the file.

3.3.5 Salient Features

Using supernodes greatly reduces the network traffic, which was a problem with Gnutella. FastTrack also addresses the problem of incomplete file downloads and slow downloads. The technology called *SmartStream* addresses the issue of incomplete file downloads, it attempts to locate another peer sharing the same file and tries to resume the download from where it had stopped. *FastStream* addresses the issue of slow downloads. When a user requests a file, there may be several peers that contain that file, when the user initiates a download of the file, this technology attempts to try downloading the file from different sources so as to relive one peer from serving the download completely.

3.4 SETI@home

3.4.1 Overview

SETI@home [28] is part of the Search of Extraterrestrial Intelligence project at the University of California, Berkeley. This is based on the traditional client-server architecture. Once the client interface for SETI@home is installed (the client is in the form of a screen saver), the clients contacts the server and the server responds by sending the data that needs to be computed upon. The data at the server is collected from the Arecibo radio telescope in Peuto Rico.

Once the client finishes the computation on the data, it sends it back to the server, the server then verifies that the data returned by the client is not fake. This takes care of any malicious clients in the network.

3.4.2 Salient Features

This implementation gives us an insight into the power of distributed peer-to-peer computing. There are so many computing systems in the world that lie idle wasting CPU and memory cycles. If we could harness this resource, then that would help in spreading the load. Also, this would reduce the cost immensely in terms of building specialized high speed machines, for example SETI@home cost around $500,000 and its computing power is close to 15 teraflops [28], while ANSI white, one of the most powerful supercomputers in the world produces 12 teraflops and it's cost is close to $110 Million [1].

3.5 Freenet

3.5.1 Overview

Freenet [6] is a P2P system based on early work by Ian Clark at the University of Edinburgh. The main goal of Freenet is to provide for anonymity for users in P2P networks. This implies that when a file is stored or modified or requested, it must not be possible to determine the user who issued the request for that action to be taken. Freenet uses the completely decentralized framework. The anonymity of users is only provided for Freenet transactions, this anonymity is not provided for general network usage.

The architecture of Freenet is a peer-to-peer network of nodes. The basic operations are to store and retrieve data files that are named by location independent keys. Each node in the network contains its own data and also includes a routing table that has the information about the other nodes and the keys that they hold. This cooperation between nodes enables nodes to utilize unused disk space on other nodes.

3.5.2 Bootstrapping

To use the resources provided by Freenet, users must first connect to the network. This is a simple process of discovering the address of one or more nodes that are already part of the network and then start sending messages to them. The new node announces its presence by sending an announcement message that consists of the message and a hash of a random seed generated by the node. The key for the new node must be generated so as to retain anonymity. The node that received the message to join from the new node generates a random seed and XOR's the random seed with the hash that it just received and then hashes the result to create a commitment. This new hash is then forwarded to another node (chosen randomly). This process continues until the *hops-to-live* count of the announcement becomes zero. The node that holds the announcement now generates a seed and all the other nodes (the path along with the announcement has traversed) reveal their seeds. The key is then created by XORing all the seeds.

3.5.3 Node Behavior

As mentioned earlier in section 3.5.2, new nodes are added to the network when they discover an existing node that is already part of the network. Node failure is detected when a node does not reply to its routing update messages. Freenet routes around such problems, as nodes contain as part of their routing tables, more than one node key, so if a node fails, then the next node is tried and so on till all nodes have been tried. If all nodes fail, then we have a routing failure.

Freenet protects the identity of user from malicious nodes as user identity is hidden. The biggest cause of concern from malicious clients is the DoS attack. This can be performed by the node introducing junk files into the network so as to fill up the network storage space. This can be avoided by schemes such as Hash Cash [5]. In this scheme, nodes have to perform a lengthy hash function as a sort of a *payment* to introduce files into the network. This helps in slowing down an attack.

3.5.4 Searching

Every file in the Freenet network is recognized by a key associated with it, this key is obtained by applying a hash function. The current hash function used in Freenet is 160-bit SHA-1 [2]. Three types of keys are used (for a more detailed explanation refer to [6]): *keyword-signed key* (KSK) - derived from a short descriptive string describing the file, *signed-subspace key* (SSK), derived from the personal namespace and *content-hash key* (CHK), derived by directly hashing the contents of the file. The weakest form of encryption is KSK which is vulnerable to dictionary attacks. Using a combination of SSK and CHK, nodes can perform insertion and updation of files in the network.

When a user requests a file, he/she must calculate the binary key associated with the file. This request is then sent to a Freenet node along with a *hop-to-live* value. The node on receipt of this checks its data store to ascertain as to whether it has the file or not, if the file is not available, it then forwards this query to its neighboring nodes (the node that is chosen has its key closest to the key requested). Once the data is found, it is passed back to the originating node (but along the return path, the file is cached at each node). This helps in localizing the request and also alleviates *hot spots*.

3.5.5 Salient Features

Freenet is a completely distributed peer-to-peer networking system that provides user anonymity and strong security against malicious clients. This project is being developed as an OpenSource implementation [10]. This system is very scalable. The support for caching files greatly enhances the effectiveness of the protocol in reducing the latency and also network traffic.

3.6 Chord

3.6.1 Overview

The Chord [29] project was developed at MIT. It provides an efficient, scalable, distributed lookup service. There is just one operation in Chord: given a key, it maps that key to a node in the network. The main thrust of this protocol is to map keys to nodes efficiently so that key lookup is easy. Keys are assigned to nodes using consistent hashing (this has a very important property that with a high probability, all nodes receive the same number of keys).

Chord assigns each node a key which is a m-bit identifier using a base hash function such as SHA-1. All arithmetic in the identifier space are done *modulo* 2^m. Keys are assigned to nodes using consistent hashing by the following way: Key k is assigned to the first node whose identifier is equal to or follows k in the identifier space. This node is called the *successor* of k and is denoted by $successor(k)$.

Each node maintains the information about its successor and another table known as the *finger table*. The finger table is a routing table of m entries, where m is the number of bits in the node identifier. The i^{th} entry of node n is the successor of $n + 2^i$, where $1 \leq i \leq m$. A finger table entry contains information about the node identifier as well as the IP address of that node (this is used for routing purposes).

3.6.2 Bootstrapping

There is no special bootstrapping mechanism in the Chord protocol, a new node joins the network by finding out the address of another Chord node and by connecting to it.

3.6.3 Node Behavior

Consistent hashing makes sure that when a node enters or leaves a system, there is only a minimal disruption. When a node joins the system, certain keys that were previously assigned to n's successor now get assigned to n, for example: if node 10 is the successor of node 5 and contains the keys 6, 7 and 9 and node 7 joins the group, in this case node 7 will take key 7 from node 10, node 5 will remain unchanged.

When a node joins the system, it sends out a join message to any known Chord node. Once the key settings are stabilized between these two nodes, the *stabilize()* function is invoked. This function runs on all the nodes periodically and is responsible for updating each node's successor pointer and its finger table.

Node failure is handled by introducing robustness into the protocol. The problem with just maintaining the successor nodes and the finger table is that if all the nodes in the finger table fail, then the algorithm leads to incorrect results. Hence, instead of just one successor node, each node maintains a list of r successor nodes. In case the immediate successor does not reply, the node queries the next node in the list. For the protocol to completely fail now, all r successor nodes must fail simultaneously, which is a rare occurance. This robustness scheme is tied closely to the value of r, the larger the value of r, the more robust the system. Also, we must take care not to increase r too large, otherwise the size of the list become unweildingly large.

Malicious nodes could send in bogus information about the Chord ring into the network and hence present the nodes with an inconsistent view of the network. To check for global consistency, a node i could ask another node j to do a lookup on itself. If node i is not returned as the answer, then this indicates an inconsistent state in the network. There is actually no remedy as yet for malicious nodes, other than detecting the fact that the network is in-consistent.

3.6.4 Searching

The efficiency of locating keys in Chord is tied closely with the efficiency of the protocol for placing keys on the network. When a node issues a *lookup(key)*, this query is sent to the successor node and it percolates through the network (using the successor pointers of the other nodes) until it reaches the right node. This is clearly not an efficient method, since the number of messages is linear in the number of nodes.

The user of *finger tables* avoids this problem. The node looks through its finger table to find that node whose *id* immediately precedes the *key*. The query is then sent to this node and using the same algorithm, it percolates through the network until it reaches the node that has the key. The number of *hops* on an average with this scheme is O(*log N*).

3.6.5 Salient Features

Chord uses consistent hashing to assign keys to nodes, previous work on consistent hashing took the assumption that each node knew about every other node in the network making such an architecture non-scalable. Chord improves upon this by only requiring a node to remember O(*log N*) other nodes. Chord has been used in quite a few other research projects like the Chord File System (CFS), which uses Chord to locate storage blocks [8]. Also, a DNS system has been built on Chord [7].

3.7 Pastry

3.7.1 Overview

Pastry [25] was developed as a project at Microsoft Research. The emphasis of this protocol is to be able to support a variety of peer-to-peer applications. The main operation in Pastry is: given a key and a message, the algorithm routes the message effectively to the node whose *nodeId* is numerically closest to the value of the key. The *nodeId*s are assigned on a circular space ranging from $0 - (2^{128} - 1)$.

Routing is achieved by prefix matching. At each routing step, a node forwards the message to the next node that shares at least 1 more digit in its prefix with the key; if no such node is found, then the node forwards the message to the node that is numerically closer to the key (provided that the other node also shares the same number of digits in its prefix with the key as the current node) than the current node. This routing needs the help of a routing table. Each Pastry node has a routing table associated with itself. The routing table is split into $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries each. Each row contains a list of IP addresses of nodes that share a particular prefix with the current node, for e.g: row

n of the routing table contains all those nodes whose first n digits share the same prefix as the current node, but the $(n+1)^{th}$ digit is different.

Apart from the routing table, two other important data structures are also maintained. The *neighborhood set M*, which contains the *nodeIds* and IP addresses of the $|M|$ nodes that are closest to the current node. Closeness between nodes is defined by the proximity metric. The choice of the which proximity metric to use depends on the application. The idea is that the application can provide the functionality to allow the Pastry node to judge the distance to another node (since a node with a smaller distance is better). The other data structure is the *leaf set L*, which is the set of nodes with half the nodes being numerically closest smaller nodeIds, while the other half of the nodes are numerically closest larger nodeIds. The *neighborhood set* is used in maintaining locality properties in the system, while the leaf set is used in routing.

3.7.2 Bootstrapping

A node must connect to an existing Pastry node to join the network. Once a new node needs to join the network, it initializes its state tables and sends a *join* request with the key set to the nodeId of the node that is about to join the network, for e.g: if nodeId of the new node is N, then the special *join* message will have its key as N. Pastry will route this message to the node which is numerically closest to N, let us say this node is M and the route traversed is through nodes X, Y and Z. Once these nodes receive the *join* message, they send their state tables to N, which receives the tables and initializes its state tables.

3.7.3 Node Behavior

When a new node arrives in the network, the sequence of events that take place is given in section 3.7.2. Any node that does not respond to messages sent by its neighbors is considered dead. The pastry routing scheme is deterministic and is hence vulnerable to failed or malicious nodes along the route that accept messages but do not correctly forward them [25]. Repeating the same query again would be futile, as the route taken would be the same.

Since nodes in a Pastry network are self-organizing, routing failures can cause the network to be partitioned into smaller, multiple Pastry networks. Even upon the resumption of the links, this may still be the case, since a Pastry network is completely dependent upon the messages that are passed in its network. Using IP multicast, the Pastry network can perform an expanding ring multicast, thereby discovering the other isolated Pastry networks and integrating them back again.

3.7.4 Searching

As mentioned earlier in section 3.7.1, routing is done based on prefix matching. When a node has a message and a key to send, it first searches its *leaf set* to find out whether the node to send the key to is in that set or not. If the node is part of the leaf set, then the message is sent to that node directly. If the node is not in the leaf set, the routing table is considered and the message is sent to the node that shares at least 1 more digit in its prefix with the key. If this is not true, then the node sends the message to the node with the same prefix as the current node, but one that is numerically closer to the key.

3.7.5 Salient Features

Pastry is a self-organizing network of peer-to-peer nodes that use a distributed hash table framework as part of its operation. Using the proximity metric, we can map routing in the overlay network to closely mirror actual IP networking in terms of distances (maybe use the RTT to measure the distance). This can be very beneficial, consider the following example: A node in Pittsburgh wants to send a message to a node in Chicago, if we are not careful about routing in the overlay network and have a node in Berlin as part of our leaf set, then the node in Chicago receives our message after its gets routed through

the node in Berlin. PAST, a large scale storage utility [9, 26] is an implementation based on Pastry. Another implementation based on Pastry is SCRIBE [27], which is a scalable publish/subscribe system.

3.8 Tapestry

3.8.1 Overview

Tapestry [31] was developed at the University of California, Berkeley. It uses a variant of the Plaxton algorithm [20] to route. The Plaxton algorithm uses a data structure called the Plaxton Mesh to be able to route between nodes. This architecture makes the assumption that all nodes are stationary. This is clearly not the case with P2P networks, where nodes join and leave most of the time. Tapestry uses the salient features of the Plaxton algorithm, but takes into account the dynamic nature of P2P networks.

Each node in the network is responsible for maintaining its routing table, which is called a *neighbor map*. The neighbor map is a data structure that is split into different routing levels. Each level contains entries that point to current nodes that have the same suffix as that level. Also, these nodes are closest in network distance to the current node. In addition to the neighbor map, each node also contains backpointers to those nodes which contain this node in their neighbor map.

3.8.2 Bootstrapping

A new node wishing to join the Tapestry network must first connect to a node that is already a part of the network. The neighbor map of the new node is then populated by routing to the new node. This helps in populating the neighbor map at each level as neighbor maps along each hop can be copied and optimized [31]. Once this is completed, a message is sent into the network about the new node that has just joined it and the nodes update their neighbor maps accordingly. This scheme is very similar to the scheme used by Pastry [25].

3.8.3 Node Behavior

A new node can be inserted into the network as shown in section 3.8.2. Node deletions are quite easy, a node can inform its neighbors and they can in turn update their neighbor maps. Node failures can be detected using TCP timeouts. Since, routing is deterministic (as in the case of Pastry), malicious nodes can cause problems in this network.

Tapestry can route around faulty or nodes that have crashed by using backpointers and having two backup neighbors in addition to the closest/primary neighbor. In case routing through the primary neighbor fails, Tapestry can then use one of the secondary neighbors and try routing through them. This increases the robustness of the system.

3.8.4 Searching

Tapestry uses Plaxton's algorithm to accomplish its routing. It is important to note that every destination node is the *root node* of its own tree, which is a unique spanning tree across all nodes. To publish an object O, a node N (which has the object O) sends a routing message to the *root node* of the embedded tree for object O. At each hop along the way, information about the object (O) and the node (N) are stored in the form of a tuple: <Object-ID(O), Node-ID(N)>. This tuple indicates that Object-ID (O) is stored at Node-ID (N).

While searching for an object in the Tapestry framework, messages are routed towards the root of that object (this can be found out from the tuple). At each step of the way, it is checked to see if there is another node that contains this object. If there is another node, then the message is re-directed to that node, else it is forwarded one more step closer to the root node. This is inherently helpful for caching.

3.8.5 Salient Features

Tapestry is a robust, self-organizing, fault-tolerant network that follows the principle of distributed hash tables. The network is inherently scalable. Like Pastry, Tapestry also uses routing in the Overlay network that closely mirrors routing in the IP network, thus ensuring no expensive routing between nodes. By placing replicas of nodes along the path from the node to the server with the object, we can hope to take advantage of caching techniques. Also, the node that retrieves the data has the discretion as to which version of data it wants to retrieve, hence we can build an in built versioning system into the network. OceanStore [14] is a wide-area distributed storage system that uses Tapestry as its underlying peer-to-peer mechanism. Also, packet level simulators for Tapestry have been developed in C and a Java implementation for large-scale deployment is on the way. Another implementation on top of Tapestry is Bayeux [32], which is an application-level multicast protocol.

3.9 Content Addressable Networks (CAN)

3.9.1 Overview

The Content Addressable Networks (CAN) work is currently being done at AT&T Center for Internet Research at ICSI (ACIRI). It also uses the distributed hash table framework. CAN provides a scalable indexing mechanism for peer-to-peer networks. The operations performed on CAN (given a key) are: insert, delete and lookup. The key space in CAN is divided into a d-dimensional Cartesian coordinate space. Each node has a part of the hash table, termed as a *zone*. Each node also holds information about adjacent nodes (this information is important for object location). In case of a request for a particular key, this request is routed to the node whose zone contains that key.

The d-dimensional coordinate space is used to store (key,value) pairs. The distribution of (key,value) pairs is done as follows: The key is mapped onto a point (P) in the Cartesian space using a hash function and the (key,value) pair is stored at the node that owns that zone which contains the point P. To retrieve the (key,value) pair, the node performs the same hash function to locate the point P. If P lies in the same zone as the node, then it retrieves the information from itself, otherwise, this message must be routed to the zone which contains the point P. This implies that the efficiency of the protocol is closely tied to the efficiency of the routing algorithm.

3.9.2 Bootstrapping

The bootstrap node to which the new node connects to must have some idea about the nodes that belong in the CAN network. The new node must be allocated its own space in the coordinate system. This process takes the following three steps:

- The new node must find a node that is already a part of the network.
- Using the routing algorithm the new node must find a node whose zone is to be split.
- The zone is split in half (half is retained by the original node and the other half is given to the new node). This information must be updated in the routing table of the neighboring nodes.

3.9.3 Node Behavior

The scenario of a new node joining has been covered in section 3.9.2. Once a new node has joined the network, it uses the information contained in the previous node to update itself about the neighboring nodes. The node that previously occupied this zone must also update its routing tables. Once this is completed, this information must be passed onto the neighboring nodes.

In the event of nodes disconnecting, the space that was occupied is taken over by other nodes in the system. If this zone can be merged with an existing zone, then this is accomplished. If such a zone cannot be formed, then the node in the neighborhood with the smallest zone handles both zones. The (key,value) pairs that the old node contained are transferred to the new node that is going to take care of

this zone. Nodes keep updating each other through periodic timers, if there is an indication that a node is dead (the node does not send periodic update messages), then a take over mechanism is initiated.

3.9.4 Searching

When a node needs a particular object from the CAN, it computes the key associated with the object. This key is then mapped to a point P in the Cartesian space. If this point lies within the same zone as the node it self, then the query is serviced immediately. If not, then the query is routed through the CAN to the zone which contains the point P. The node which is in charge of that zone returns the request value. Routing in a CAN follows a straight line path through the Cartesian coordinate space. The message is sent out through the neighboring nodes in the CAN (each node has a list of its neighboring nodes). A node is a neighbor of another node in the CAN if their coordinate spans overlap along $d-1$ dimensions and abut along one dimension.

3.9.5 Salient Features

CAN provides a scalable, efficient distributed hash table. While Chord and Pastry have flat key spaces, CAN has a d-dimensional key space, which increases the number of keys that can be stored using this system. There are a number of improvements suggested to CAN which can really enhance this protocol. Some of the suggestions are: multi-dimensioned coordinate spaces - this would reduce the path routing length, multiple coordinate spaces - also called realities, this would help in replicating the hash table, which would help in fault-tolerance, better routing metric - to better reflect the underlying IP routing, multiple hash functions - here a (key,value) is hashed using k different hash functions, and hence there are k points in space where this (key,value) pair could exist. This would help in reducing routing latency, because we can simultaneously route to all k nodes in parallel.

4 Conclusion and Future Work

In this paper we journeyed through the world of P2P networking. We took a look at the different kinds of P2P networks that can be deployed. We also looked at real-life P2P systems that have been deployed and studied their characteristics.

Peer-to-Peer networking has certainly bloomed in the last few years and this interest is only going to continue in the years to come. There is a lot of business potential in this technology (clearly ascertained by the popularity of file sharing programs like Napster, KaZaa and LimeWire). P2P is being used by many businesses today for some efficient solutions, for e.g: Cloudmark Inc. [12] uses P2P technology in its software $SpamNet^{TM}$. This software fights spam, when a spam message is submitted to Cloudmark SpamNet, the system generates a secure fingerprint or a digital signature of each message. This fingerprint is then shared with other SpamNet users to identify the same spam message in their email. SpamNet is actually an extension (using P2P) of an original software called Vipul's Razor [22], which was developed by Vipul Ved Prakash, a software architect.

With increasing deployment, there will be a lot of issues to be overcome before this technology really becomes more mature. Better algorithms would be needed to handle the problems of scalability, anonymity and data location. The ever increasing interest in this subject would only help improve and push this technology in years to come.

References

[1] Accelerated Strategic Computing Initiative (ASCI). `http://www.llnl.gov/asci/`.

[2] American National Standards Institute. American National Standard X9.30.2-1997: Public Key Cryptography for the Financial Services Industry - Part 2: The Secure Hash Algorithm (SHA-1), 1997.

[3] Software & Information Industry Association. Stretching the fabric of the Net: Examining the present and potential of peer-to-peer technologies. Whitepaper, 2001.

[4] Bearshare. http://www.bearshare.com/.

[5] Hash Cash. http://www.cypherspace.org/~adam/hashcash/, 2000.

[6] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[7] R. Cox, A. Muthitacharoen, and R. Morris. Serving dns using chord. In *First International Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002.

[8] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[9] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, pages 75–80, Schloss Elmau, Germany, May 2001.

[10] FreenetProject.org. http://freenetproject.org/cgi-bin/twiki/view/Main/WebHome.

[11] The Gnutella home page. http://gnutella.wego.com/.

[12] Cloudmark Inc. http://www.cloudmark.com/.

[13] Kazaa. http://www.kazaa.com/.

[14] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[15] Limewire. http://www.limewire.com/.

[16] Dejan S. Milojicic, Vana Kalogeraki, Kiran Nataraja Rajan Lukose, Jim Pruyne, Bruno Richard, and Zhichen Xu Sami Rollins. Peer-to-Peer computing. Technical Report HPL-2002-57, HP Laboratories Palo Alto, March 2002.

[17] Morpheus. http://www.musiccity.com/.

[18] Napster. http://www.napster.com/.

[19] NullSoft. http://www.nullsoft.com/pinkumbrella.phtml/.

[20] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[21] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[22] Vipul's Razor. http://razor.sourceforge.net/.

[23] Roberto Rinaldi and Marcel Waldvogel. Routing and data location in overlay peer-to-peer networks. Research Report RZ-3433, IBM, July 2002.

[24] Jordan Ritter. Why GNUtella Can't Scale. No, Really. http://www.darkridge.com/~jpr5/doc/gnutella.html.

[25] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[26] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[27] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[28] SETI@home. http://setiathome.ssl.berkeley.edu/.

[29] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. *ACM Sigcomm*, 2001.

[30] WinAmp. http://www.winamp.com/.

[31] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[32] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination, 2001.