Topics in Database Systems: Data Management in Peer-to-Peer Systems

Survey of Peer-to-Peer Systems

Vicki Tziovara

Abstract

Peer-to-peer (P2P) networking has generated tremendous interest worldwide among Internet users as well as computer professionals. P2P systems like Kazaa and Napster rank amongst the most popular software applications ever. P2P systems have become popular Internet applications since the arrival of Napster and other P2P architectures used for music file sharing. Several businesses and Web sites have promoted P2P technology as the future of Internet networking. In recent years, peer-to-peer systems have emerged as an interesting architecture for implementing distributed file systems. In a P2P network, end users share resources which are able to exchange. Information is distributed among the member nodes instead of being concentrated at a single server. A successful P2P file-sharing network must have some desirable properties, such as ease of use, ability to expand over large numbers of peers, produce results despite node failures and find infrequent items. This survey of P2P systems presents different system architectures used for file-sharing and query routing protocols. We discuss more complex queries, such as range queries over one or more attributes and replication techniques. Finally, we refer to algorithms for updating replicas.

1. Introduction

Peer-to-Peer systems are distributed systems in which all nodes are equivalent in functionality and perform similar tasks. A peer-to-peer network usually consists of a large number of equal peer-nodes. Each node acts both as a client and as a server with respect to the other nodes of the network. Every peer stores local content and makes (some of) it available to other peers. The nodes of the network are connected in order to share resources such as files (content like audio, video, data or anything in digital format), computing power, data storage, network bandwidth, etc. Another feature of peer-to-peer systems is the autonomy of the participating peers. The nodes are able to function independently and establish their own communication rules, since there is no central coordination. Peers organize themselves into some network topology and are capable of preserving connectivity when nodes join or leave the network. The peers are also able to maintain some acceptable performance when some failure occurs.

The connected peers construct a set of logical connections with their neighbours, which is called the overlay network. This overlay network is not necessarily the same as the physical network, but is usually constructed on top of the underlying physical network. Overlay networks can be structured or unstructured, i.e., there is control over network topology and placement of data, or not.

P2P computing raises some interesting research problems. The most interesting problem is the *Look-Up Problem [2]:* Given a data item X stored at some dynamic sets of nodes in the system, to be able to successfully locate it.

The most popular search methods used in P2P systems are: blind and informed search[6]. In *blind search*, the query is propagated without knowledge of content location, while *informed search* utilizes information about object locations to forward the query to nodes likely to produce answers.

2. Structured P2P systems

Structured P2P systems are based on the principle that the data is placed at precisely specified locations. Usually there is a mapping between the file identifier and the location where the file is placed. In this category, the overlay network assigns keys to data items by converting for example the file name to a numeric value, called *key.* Then the key is mapped to a particular node and stored.
When a peer tries to locate a file, the keys are used and the node that possesses the item is found. This technique leads to efficient discovery of data items. Chord and CAN can be classified as structured systems.

2.1 Distributed Hash Table - Based Systems (DHTs)

These systems use a *hash function* to map the data item to a numeric key. Each data item is stored at a particular peer. A hash table is a data structure that maps keys onto values and serves as a building block in the implementation of software systems.

2.1.1 THE CHORD PROTOCOL

Chord [3] is a distributed lookup protocol used in dynamic P2P systems with frequent node arrivals and departures. This protocol uses a variant of *consistent hashing* to assign keys to Chord nodes.
First, Chord uses a base hash function such as SHA-1[1] to assign to each node and key a unique *m-bit identifier*. This process is described in detail as follows:

- Each node is assigned a node-identifier by hashing the node's IP address.
- Each key is assigned a key-identifier by hashing the key.

The identifier length *m* must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

The Chord nodes are arranged in an imaginable *identifier circle* modulo 2^m. Figure 1 shows an identifier circle with m=3. The circle consists of 3 nodes: 0, 1 and 3.
The assignment of keys to certain Chord nodes is described as follows: Key *k* is assigned to the first node whose identifier is equal to or follows (the identifier of) *k* in the identifier space. This node is called the *successor node* of key *k*, denoted by *successor(k).* If identifiers are represented as a circle of numbers from 0 to 2^m - 1, then *successor(k)* is the first node clockwise from *k.* As a result, *each key is stored at its successor node.* For the example presented in Figure 1, the successor of identifier 1 is node 1, so key 1 must be located at node 1. Similarly, key 2 is assigned to node 3, and key 6 to node 0.

Consistent hashing tends to balance load, since all nodes receive roughly the same number of keys. Furthermore, the disruption caused by nodes joining or leaving the network is minimal due to consistent hashing. When a node *joins* the network, some keys previously assigned to n's successor should now become assigned to n. When a node *leaves* the network, all the keys assigned to n are reassigned to n's successor.

Each node needs to be aware of its successor node on the circle. Using this information, queries for a given identifier can be passed around the circle through successor pointers,

until they find a node that succeeds the identifier. The query maps to this node. Let N be the total number of nodes. To ensure that it will not be necessary to traverse all N nodes to find the node the query maps to, Chord maintains additional routing information. Each node maintains a *finger table,* which is a distributed routing table with at most *m* entries. Entry *i* in the finger table of *n* points to the successor of node $n+2^i$, e.g., in Figure 1, the 1st entry of 1's finger table points to the successor of $1+2^0=2$, which is 3. A lookup for key *k* is performed at node *n* by searching its finger table for node *j* whose identifier most immediately precedes *k*. The successor of *j* is the node that stores the item with key *k*.

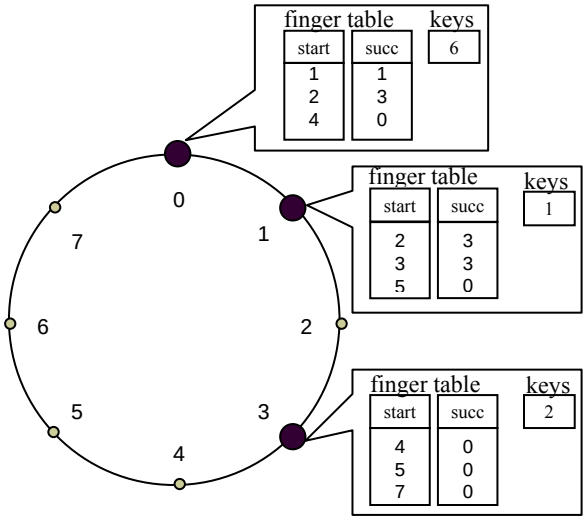

Figure 1: A Chord ring with three nodes 0, 1 and 3.

Each Chord node maintains a finger table, which contains routing information about only a few other nodes. A Chord node requires information about only O(logN) other nodes to work efficiently. In addition, Chord resolves all lookups using O(logN) messages to other nodes.

2.1.2 CAN

CAN (Content-Addressable Network) [4] is a Distributed Hash Table- based protocol, which resembles the functionality of a hash table (efficiently maps keys onto values). The basic operations CAN performs are: Insertion, Lookup and Deletion of (key, value) pairs in the hash table. Each CAN node stores a part of the entire hash table, called a *zone.* Moreover, each node stores information about a few "adjacent" zones in the table.

CAN structure is based on a virtual d-dimensional Cartesian coordinate space on a *d-torus*. This means we use the assumption that the coordinate space wraps. This virtual space is used to store pairs of keys and values. At any point in time, the coordinate space is partitioned into distinct zones, such that every node owns its different zone within the entire space. The zones cover the entire space and do not overlap. Figure 2 shows a 2-d space between 5 nodes and the values each covers.

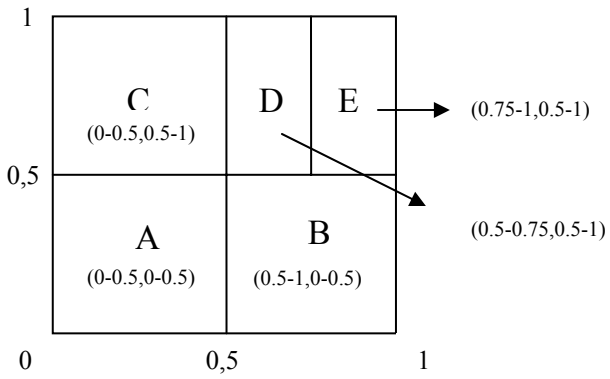

Figure 2: Example CAN

Every (key, value) pair is stored as follows:

- A uniform hash function is used to map the key to a point P in the coordinate space.
- The point P lies within the zone owned by node N.
- The (key, value) pair is stored at node N.

To retrieve an entry corresponding to a particular key, the same hash function is applied to map the key to point P in space and then retrieve the value from the point P.
CAN nodes are able to organize themselves in an overlay network that represents the coordinate space described above. Each node stores information about a small number of nodes whose zones are adjacent to its own zone (*immediate neighbours*). This information involves the IP addresses and the coordinate zones of the immediate neighbours.

Routing in CAN is performed at node n by sending a message towards the destination to the neighbour with coordinates closest to the destination coordinates.
When a node joins the network, a part of the coordinate space must be assigned to it. The new node chooses randomly a node already in the network and uses the routing mechanism to find a node whose zone can be split. This zone is split in half and one half of it is assigned to the new node. Finally, the node's neighbours must be notified for the new node.
When a node leaves, it hands over its zone and the corresponding (key, value) pairs to one of its neighbours.

Each node keeps information about O(d) neighbours in the network and the average routing path length is $O(n^{1/d})$.

Some design improvements leading to better CAN performance are:

1. Use of Multiple "realities" – multiple, independent coordinate spaces.
2. Multi-dimensional coordinate spaces – increase the number d of dimensions.
3. Better CAN routing metrics – improve the routing metric by taking into consideration the underlying IP topology. For example, routing based on RTT (Round-Trip-Time) to each of the node's neighbours.
4. Overloading coordinate zones – multiple nodes share the same zone.

5. Topologically-sensitive construction of the overlay network – assume the existence of a set of machines on the Internet (landmarks) and compute RTT-time to each landmark. The underlying idea behind this construction is that topologically close nodes will be placed at the same zone of the coordinate space.
6. Multiple hash functions – use k different hash functions to map the key onto k different points in space.
7. More Uniform partitioning – when a node joins the network, find the zone with the largest volume and split it in half.
8. Caching and Replication techniques.

2.2 Comparison of Structured P2P Systems

In order to compare the structured systems presented above (CAN and Chord), we can mention the cost of maintaining the routing information by each node, assuming the number of nodes in the network is n. The difference is that a Chord node requires information about O(logn) other nodes to work efficiently, while each CAN node keeps information about O(d) neighbours.
In terms of performance, Chord resolves all lookups using O(logn) messages to other nodes while the average routing path length in CAN is $O(n^{1/d})$. Table 1 presents some characteristics of Chord, CAN and flooding and compares their routing performance, as well as the routing table size for each. Then, we explore their ability to scale.

Protocol / *Parameters*	**CHORD**	**CAN**	**Flooding**
Type of overlay network	Ring Topology	d-dimensional Cartesian space	Random topology
Characteristics	Hash look-up routing		Neighbour forwarding
Routing Table size	O(logn)	$O(n^{1/d})$	Number of connected neighbours
Routing path length	O(logn)	O(d)	Variable
Upper limit on total number of visited nodes	Dimension of ring	Dimension of space	Unbound
Scalable	Yes	Yes	No

Table 1: Comparison of Chord, CAN and flooding

3. Unstructured P2P systems

Unstructured P2P systems refer to systems where there is no control over network topology or data placement. Data is distributed randomly over the peers. Napster, Gnutella and KaZaa are systems which can be classified as unstructured. Decentralized and unstructured Peer-to-Peer networks such as Gnutella are attractive for certain applications because they require no centralized directories.

3.1 Flooding - Gnutella

Gnutella is an unstructured P2P system which relies on flooding. It is based on the decentralized model, where there is no central organization. The nodes self - organize into an overlay network. To locate content, a peer sends a query to its neighbours on the network. Then, the neighbours forward the query to all of their neighbours until the query has traveled a certain distance. This distance is computed using the TTL value (Time-To-Live), which is the maximum number of hops allowed for this query. The TTL value decreases by one at every hop made towards the destination, until the file is found or TTL becomes zero. Then the lookup ends. If the value of TTL is not large enough, the search for the file may be unsuccessful. If the file is found, the peer that has the file sends back the answer to the initial node using the same path.
The main features of Gnutella are simplicity and robustness. Although Gnutella manages to discover content efficiently at most times, this approach does not scale, since the overhead produced by forwarding the query to a large number of peers is enormous. Another drawback of Gnutella is the increased traffic observed at the network caused by flooding. Furthermore, this protocol encounters many problems when nodes do not stay on line for a long time to answer queries.
An improvement of the original flooding approach is proposed by Lv et al.[6], who suggest replacement of flooding by multiple parallel *random walks*. In Random walks, the query initiator sends query messages to k random neighbours and in turn, each neighbour passes the request to its own random neighbour. A walker terminates if it locates the object or if it fails. Failure is detected using some TTL counter or by *checking* every few steps, if the item has been located.

3.2 Centralized-Search Systems - Napster

Centralized-search systems use specialized nodes that maintain an index of the documents available in the P2P system. Napster is a centralized unstructured Peer-to-Peer system, which became very popular because of its support for music file-sharing. It is based on the existence of a central peer which plays a more significant role to the functionality of the network than other peers. This peer acts as a "supernode" and usually maintains a central index of the files shared by local peers. When a new node signs-in the network, registers itself with the central node (server). Then, it publishes its files and the central index is updated to include the new files. Every peer in the system knows the identity of the central server, while the server keeps information about all nodes and objects in the system. Whenever a peer wishes to locate a file, it sends the request to the central node which holds the index. The central index matches the request with one of the peers that store the file and returns its IP address. The requesting node passes the request to the returned peer and downloads the object directly from it.

The Napster model proved to be efficient at lookups, because only one message is necessary for locating context. On the other hand, Napster is based on a central superpeer. This means that if any problem occurs to the central node, the whole system fails. In addition, such centralized systems encounter security problems (hacker attacks).

Another disadvantage is its limited ability to scale, i.e., expand over a large number of peers. Furthermore, this model presents great difficulty at keeping the central index updated.

3.3 Distributed-Search Systems - Routing Indices

A distributed-search system maintains indices at each node, which are used to locate objects and improve look-ups. These indices need to be small. Examples of such indices are Routing Indices (RIs) proposed by Crespo and Garcia-Molina [5]. A RI is a data structure that contains information about the files stored at nearby nodes.
Files are categorized into topics and queries request documents on particular topics. Each node has a Routing Index. A RI points at the direction to locate an object, rather than its actual location. Figure 3 shows 10 connected nodes. Node E stores the document with content "x" but the RI of A points to node B instead of pointing directly to E. This reduces the size of the RI, which becomes proportional to the number of neighbours, rather than the number of documents.

There are three different types of RIs: Compound RIs (CRIs), Hop-Count RIs (HRIs) and Exponentially Aggregated RIs (ERIs).
A CRI contains information about the total number of documents maintained by nearby nodes, as well as the number of documents on each topic. Given the query, CRI entries are used to estimate each neighbour's "goodness" and rank neighbours according to this metric. By "goodness" we refer to a neighbour's suitability to receive the query and produce results. This way, the neighbour which is more likely to produce results is selected.
An HRI contains aggregated routing indices for each hop, up to a maximum number of hops, called the "*horizon*" of the RI. The metric to select a neighbour is computed as the ratio between the number of documents reachable through that neighbour and the number of messages needed to get those documents.
Lastly, an ERI stores the result of applying the regular-tree cost formula to an HRI. This is done to combat the fact that the HRI does not contain information beyond a certain hop-count threshold.
Experiments show that the use of RIs reduces the number of messages to answer a query by half when compared to not using RI.

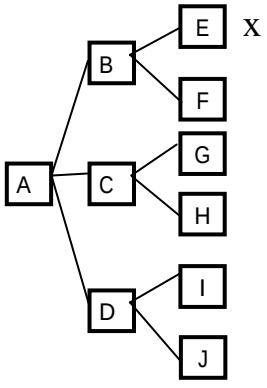

Figure 3: Example of RIs

3.4 Comparison of Unstructured P2P Systems

The main problem that appears in Unstructured P2P systems is their difficulty in locating context without distributing the query in a large number of peers. This is the reason unstructured P2P systems lack the ability to scale.

4. Loosely-Structured P2P Systems

These peer-to-peer systems form a *loose structure* on top of the underlying overlay network. The overlay network is usually based on unstructured models (i.e. Gnutella) or DHT-based protocols. When the loose structure fails to locate content, peers resort to using the underlying network.
We will now refer to an important principle, called the *Interest-based Locality Principle*, which posits that if a peer has a piece of content one is interested in, it is very likely that it will have other items that one is interested in as well.

Interest-based Shortcuts [8]

This protocol is based on the Interest-based Locality Principle stated above. It is based on creating shortcuts that point from one peer to another, provided those peers share similar interests. By the term "interests", we refer to a group of files a peer is interested in. In other words, if two peers are interested in the same files, they share "similar interests". The main goal of introducing the use of shortcuts is to improve the performance of the overlay network and therefore increase scalability, by reducing the time to locate documents and the amount of load in the system.
The description of the *Interest-based Shortcuts* protocol that follows, assumes the existence of an overlay Gnutella network and the creation of additional links (shortcuts) on top of the overlay that serve as performance-enhancement hints.

When a peer joins the network, it occupies a special amount of storage to implement shortcuts. In order to find out other peers' interests, it resorts to flooding the network. Flooding returns a set of peers having the content. One of these peers is selected at random and added to the peer's "shortcut list". Any following queries are answered through the shortcut list. If content is not found through the list, the overlay of Gnutella is used. The process is repeated for adding new shortcuts.
Another variation exists where instead of selecting only a random peer to add to the shortcut list, more shortcuts are added to the list at once, e.g. adding 5 shortcuts at a time.

The query is forwarded to peers through shortcuts. If none of the shortcuts possess the item, the query is flooded to the entire system. The shortcut list is ranked according to the utility of shortcuts. Useful shortcuts are placed at the top of the list. A peer asks all shortcuts on its list starting from the top, until content is traced.

There are many advantages of using shortcuts. First of all, shortcuts can be easily integrated into any underlying content locating network, such as Gnutella, DHTs, hybrid centralized – decentralized architectures (such as KaZaa), etc. Furthermore, shortcuts are only used to improve the performance of the overlay network, so in case shortcuts fail, any document can be accessed via the underlying network.

5. Content Clustering P2P Systems

These systems partition the nodes into clusters according to data they hold. Each group contains semantically similar data. This way, a logical network is created that contains groups of nodes with similar data, called a *semantic overlay*. The distance between two nodes is proportional to their dissimilarity.

5.1 Associative Search

Associative search systems presented by Cohen et al. [9] divide nodes in groups based on content they possess. Partitioning of nodes is performed based on guide-rules. Each *guide-rule* is a set of peers that satisfy some predicate. Peers containing semantically similar data belong to the same guide-rule. Guide-rules dictate the overlay structure in the sense that peers belonging to the same guide rule are grouped together forming different sets of peers. Each peer maintains a small sequence of guide-rules it belongs to. Furthermore, it keeps track of other rules and peers that belong to them.
When a node searches for content, it forwards the query only to neighbours that belong to the same rule. The initial node has the flexibility to choose which guide-rules to use for a given query. The search process within a guide-rule is essentially blind.

There are two different schemes to decide on each peer's probability of being probed during the search process. URAND(Uniform Random Search) offers all peers the same likelihood of being probed, while in PRAND(Proportional Random Search), peers are probed proportionally to their index size.
A simple search strategy is RAPIER, which involves a two-step process. First, the requesting peer selects a random guide rule from its index and initiates search at peers that belong to this rule. Another greedy strategy is GAS, which finds at each step the guide-rule that maximizes the likelihood of resolving the query and forwards the query to it. RAPIER is better than PRAND when peers have fewer topics.

A node joining the network must find out the list of guide-rules it belongs to and peers belonging to the same rules. Next, it has to inform the other peers for its arrival. This process is performed as follows: For each item it possesses, the peer performs a search to the network, either by flooding or with the help of another peer. This way, it discovers useful information in order to create its own index. The peer informs peers of the same rule for the items it possesses, to update their own indices.

5.2 Semantic Overlay Networks (SONs) [10]

SONs are P2P systems where nodes with similar content are clustered together. Groups of peers are built according to their "topics".

Queries are processed by identifying which group (or groups) of nodes are more appropriate to answer it. Then the query is forwarded only to the members of this group.

The SON is an overlay network based on the idea of hierarchical classification. A *classification hierarchy* is a tree consisted of concepts. Each document and query is classified into one or more nodes of the tree. In order for the protocol to work efficiently, the classifier mechanism must produce results fast enough and make as few mistakes as possible.
There are two different strategies which can be followed in order for the classifier to work and assign a query to a particular SON. A *conservative* strategy will place a node in the SON for concept c if it contains at least one document in concept c. A *non conservative* strategy will place a node in the SON for concept c if it has a significant number of documents in c.

When a node issues a query, the classifier is used to assign the query to the appropriate SONs and forwards the query to them. Nodes within these SONs find matches for the query by using some mechanism such as Gnutella flooding or super- peers.

When a node wishes to join the system, it uses flooding to learn the hierarchies. Then, the node runs a document classifier based on the known hierarchy. The next step is for the node to join the appropriate SONs. If the conservative strategy is used, the node joins each SON it has at least one document that matches in concept. If a non conservative strategy is used (e.g. Layered SONs), the node joins each SON it has a satisfying number of matching documents. Finally, the node must find the nodes that belong to those SONs. This is done either by flooding the network or by using a central directory.

5.3 Self - Organizing Semantic Overlay Networks [11]

This approach extends classical Information Retrieval algorithms to the Peer-to-Peer environment. Two well-known algorithms, VSM and LSI apply to P2P systems. They represent document and queries as vectors in the Cartesian space and measure the similarity between queries and documents. For each document a vector is created and used for placing the document to some node. Some nodes of the overlay form a pSearch Engine. A peer that desires to issue a query or publish document indices connects to any Engine node.

5.3.1 VSM (Vector Space Model)

VSM creates a vector for each document and each query. The vector elements represent the importance of a term in a document or a query. Each vector element is computed by the product of the *term frequency * inverse document frequency,* where *term frequency* is

the number of occurrences of the term in the document and *inverse document frequency* is the frequency of the term in other documents.
The similarity of the query vector and each document vector is computed as the inner product of the two vectors. Finally, the document with larger value in similarity is selected. VSM suffers from noise and synonyms in documents.

5.3.2 LSI (Latent Semantic Indexing)

LSI instead of term vectors uses statistical indices for content location. It is based on SVD (Singular Value Decomposition), which is applied to the high-dimensional term vectors VSM produces to convert them to lower-dimensional *semantic vectors*. The elements of a semantic vector represent the importance of an abstract concept in the document or query. This way, LSI does not suffer from noise and synonyms in documents. Furthermore, LSI can be applied to CAN.

6. Range Queries

A *range query* is a query that asks for all objects whose value over a certain attribute falls into a desired range, e.g., age between 30 and 40 years old. A *multi – attribute range query* is a combination of sub-queries on each attribute, e.g., name = 'A' & age < 25.

6.1 Mercury

Mercury[12] is a protocol that supports multi-attribute range queries. Its structure is based on the creation of several groups of nodes. The nodes of the network are partitioned into groups of nodes called *attribute hubs*. Every physical node can belong to several hubs. Each hub is responsible for one of the attributes in the application schema. The hub nodes are organized into a circular overlay and each node is responsible for a contiguous range of values for the particular attribute, e.g., node 1 is responsible for the range [20,40) of age values. Each hub node has a pointer to its successor and predecessor nodes in the circle. It also maintains a link to each of the other hubs. As the number of hubs is expected to be rather low, the cost of maintaining additional links is considered insignificant. Finally, a node maintains k long-distance links for efficient inter-hub routing.
Every data item is represented in Mercury as a list of the form (type, attribute, value), e.g., int x=100. The queries are conjunctions of predicates of the form (type, attribute, operator, value),e.g., int x>100. A disjunction of predicates can be resolved by multiple distinct queries.
To ensure that queries will find all relevant data, an incoming data record is inserted into all hubs responsible for one of the query attributes. Each node inside a hub is responsible for a range over the attribute and stores data items with attribute values which belong to this range. When the data item is sent to a hub, the item is stored at the node with the appropriate range.
Figure 4 shows two hubs H_x and H_y, responsible for the attributes x and y respectively. The nodes are denoted as a, b, c, d, e, f, g and each range follows the node's name. A record int x=100,int y=200 is inserted into both hubs, at nodes b and f (because $x \in$ [80,160) and $y \in$ [105,210)).

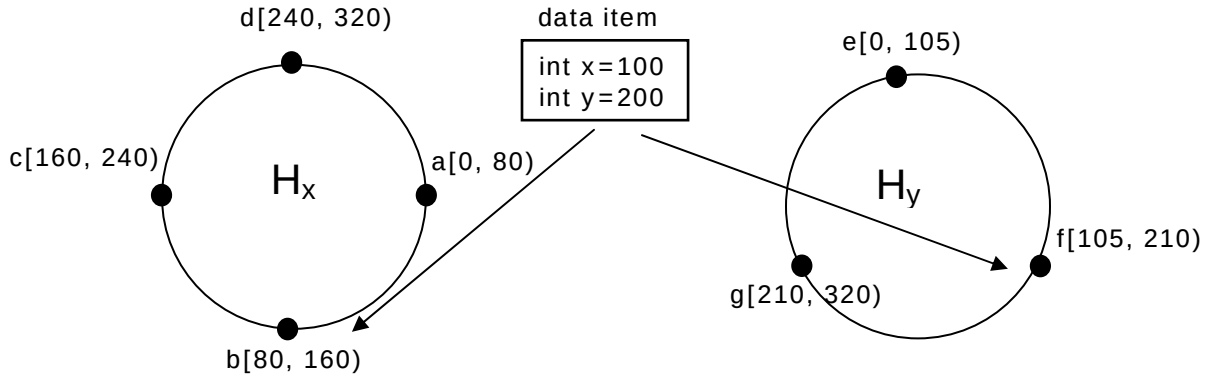

Figure 4: Example insertion of data records

Routing of queries is performed in a similar manner. The difference is that the query needs to be entered at exactly one hub. One of the hubs responsible for one of the query's attributes is selected randomly and receives the query. As mentioned above, each hub node has a pointer to its successor and predecessor node in the circle. Thus, the node having the query uses those pointers to route to the first value appearing in the range and then the contiguity of range values is used to spread the query along the circle, to gather all necessary results.

Figure 5 shows the example of the query $60 \leq x \leq 110$ and $y>120$. The query enters H_x at node d and is routed and processed at nodes a and b through successor pointers. The arrows show the route of the query.

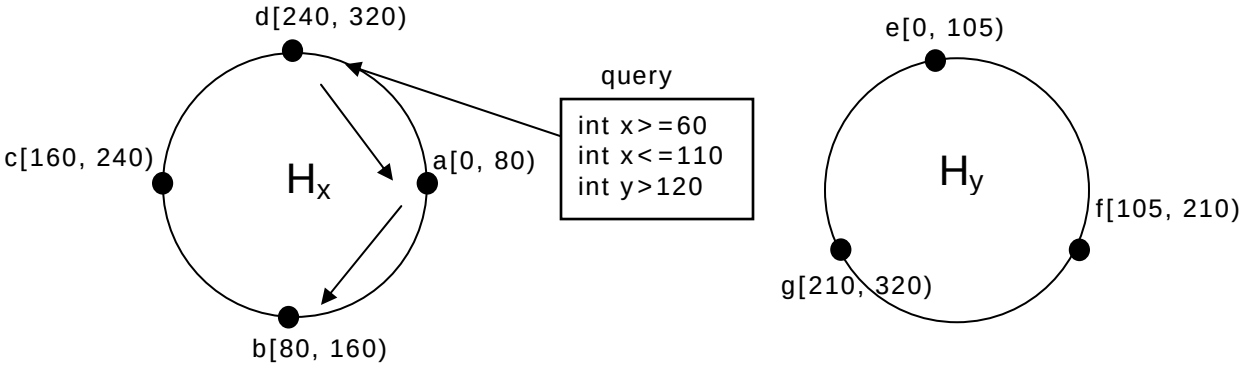

Figure 5: Example routing of a query

When a node joins the network, it queries an existing node to gather information about the hubs and a number of representatives for each hub. Then, the node randomly chooses one hub to join and installs itself as a predecessor of a member m of that hub. It takes over half of m's range and copies its routing state and links. Then, it sets-up its own pointers and initiates random-walks to the other hubs to obtain new neighbours distinct from its successor's. This way the node becomes a member of the hub.

When a node decides to leave the network, all successor, predecessor, long-distance and inter-hub links must be repaired. This repair exploits appropriate techniques, such as periodic reconstruction of long-distance links using recent estimates of node counts, querying the successor and predecessor for their links, etc.

6.2 Caching Range Queries

Current P2P systems are mainly used for object sharing and provide answers to queries based on object names. This means that P2P systems cannot offer the means to answer more complex queries, such as range queries. In order to overcome this problem, a new approach is presented by O.D. Sahin et al.[13], which supports the design of a general purpose P2P system for data sharing. We assume the existence of a relational database whose schema is globally known to all peers. The approach is based on the idea of using peers to store the answers of previous queries. These answers are usually range partitions of the database and are later used to respond to incoming queries. It can be obvious that caching results helps to reduce the retrieval delay in future queries.
The system model that is presented here is based on the structure of CAN and uses a 2-d virtual coordinate space. The virtual space is partitioned into rectangles, called *zones* and zones cover the entire space and do not overlap. Each zone is assigned to a peer that participates in the partitioning, called an *active* node. All the other peers that do not take place in the partitioning are referred to as *passive* nodes. Each passive node registers with one of the active peers. Each active node maintains a routing table with the IP addresses and the coordinate zones of its neighbours (similarly to CAN). A range query <a, b> is mapped at point (a, b) in the virtual space. This point is called the *target point* of the query range and is used to select the location to store the answer of the query. The target point belongs to a zone, called the *target zone.* The node-owner of this zone is called the *target node*. The answer of the range query is stored at the target node of this query range. When a node receives the answer of a range query, it caches the result and informs the target node (the target node creates a pointer to this node). Another alternative is for the target node to cache the result itself. In both cases, the target node has full access to the result.
Figure 6 shows that the range query <60,110> is mapped at point (60, 110) in space, which corresponds to zone 6. The answer of this query may be stored at the owner of zone 6 or the node will store a pointer to the peer that caches the tuples in that range.

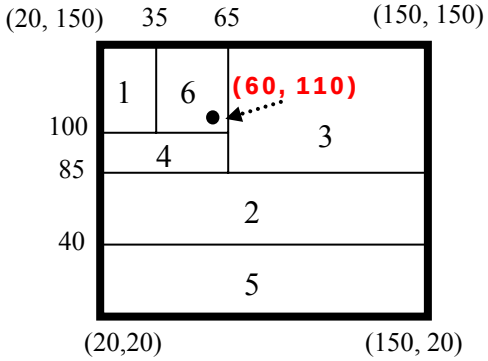

Figure 6: Example caching of query results

Initially the entire coordinate space is considered as a single zone and is assigned to the overall database which is the only active node. As new nodes arrive, the existing zones split into rectangles and new zones are assigned to passive nodes, which become active. The current owner of the zone decides to partition its zone in two rectangles. For this reason, it finds a passive node through its own list of passive nodes or by asking its neighbours. Then, the owner node splits its zone in two parts that share equal load, i.e., parts that are responsible for equal numbers of points and assigns one half of its zone to the passive node. A decision for partitioning is taken either because the current owner has to answer too many queries or because the current owner has to route too many messages. The new active node is always assigned a zone on the right or below the current owner.

Query routing is performed like CAN. The first place to look for cached results is the target zone of the range. When a query is issued, it passes through neighbouring zones until it reaches its target zone. Each node on the route forwards the query to its neighbour with coordinates closest to the target point in the virtual space.
Figure 7 illustrates how the range query <60,110> is routed in the virtual space. The query is issued at node 5 and then routed to node 6.

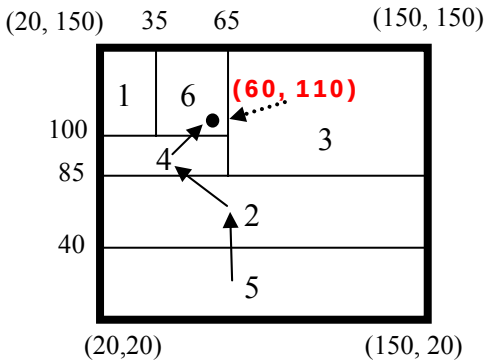

Figure 7: Example routing of a query

Research analysis has shown that the average routing path length is O(√n), where n is the number of nodes in the system.
When the query reaches the target zone, the cached results in this zone are checked for matches. If any result is found, it is forwarded to the querying node. If there is no such result, the target node forwards the query to adjacent zones in the top left space area comparing to its own zone. These zones are likely to store results, since they store answers to range queries that are hyper sets of the answers of this query. For example, let <a, b> be the range query and point (a, b) its mapping. Every point that lies in the top left area of this point, has coordinates (a-offset, b+offset), where offset is a positive value. *Acceptable region* is the square area defined by these offsets and the target point.
There are two different approaches to forwarding queries from the target node to adjacent zones. The first one supports forwarding to all top left neighbours, while the other forwards the query only to the neighbour having largest overlap area with the acceptable region.

Figure 8 presents the shaded acceptable region of the target point (a, b) and the zones B, C and D that intersect with the acceptable region and may contain results. The first approach forwards the query to zones B, C and D, while the second approach forwards the query to zone D.

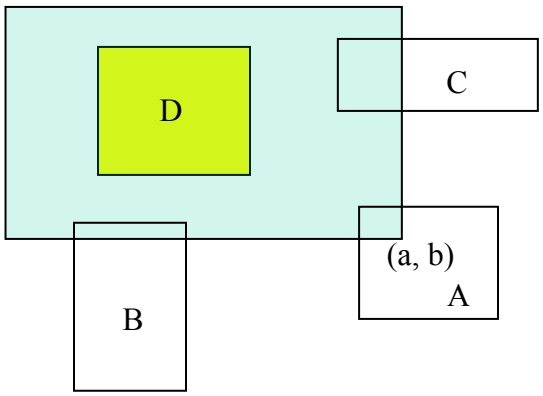

Figure 8: Acceptable region and candidate areas for forwarding the query to acquire results

The techniques presented above can be used for exact-match queries. For example the query x=20 can be answered by querying the system for the range <20, 20>. Some improvements of CAN can also be applied, such as multiple realities, overloaded zones, etc. Finally, this technique generalizes to answer multi-attribute range queries.

6.3 SCRAP

SCRAP (Space-filling Curves with Range Partitioning) [14] is an approach proposed by P. Ganesan et al. [14] to support *multi-dimensional range queries*, i.e. conjunctive queries containing range predicates on two or more attributes of a relation. SCRAP involves two consecutive stages. First, it maps the data into one dimension using a space-filling curve. Then, it partitions the data in ranges across a dynamic set of nodes. The two steps can become clear with the following example: say we have 2-d data that consists of 4-bit attributes, A and B. A two-dimensional data point <a, b> can be reduced to an 8-bit value, by interleaving the bits in a and b. Thus, the point <0100,0101> becomes <00110001>. This mapping is bijective. In the second step, the data are range-partitioned across the available nodes.
To route a multi-dimensional query, we convert it to a set of 1-d range queries and send each of them to nodes whose ranges intersect the query range. Routing of queries to the appropriate nodes is performed using *skip graphs*, i.e., circular lists of nodes with skip pointers to enable fast routing in O(logn) hops.

6.4 MURK

MURK (Multi-dimensional Rectangulation with kd-trees) [14] is a different approach to support multi-dimensional queries. MURK is based on the following idea: partition the data space into rectangles and assign each node to manage one rectangle. Partitioning is

performed based on kd-trees, in which each leaf corresponds to one rectangle. The space is split cyclically according to the dimensions and the new parts that are produced have equal load.
Initially, we assume there is only one node in the system that manages the entire 2-d space. When a second node joins, the space is divided along the first dimension in two parts of equal load. This corresponds to splitting the root of the kd-tree in two leaves. When new nodes arrive, the partitions are again split in halves over the other dimensions. This corresponds to splitting a leaf of the kd-tree. We note that the dimensions are used cyclically in splitting. When a node leaves, a sibling takes over its space. If there is no sibling, it finds a low-level leaf in the sibling sub-tree.

The partition based on kd-tree is similar as that in CAN. Both hash data into a multi-dimensional space and try to keep load balancing. The major difference is that a new node splits the existing data space equally in CAN, rather than splitting load equally.

Routing in MURK is performed similar to CAN. It involves creating links between neighbouring-nodes in order to be able to send a multidimensional query to all the relevant nodes. The query is forwarded from a node to the neighbour that reduces the Manhattan distance by the largest amount. An example of routing a query is shown in figure 9.

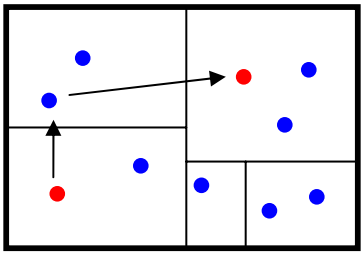

Figure 9: Routing in MURK

An optimization for faster routing is to create more links (called skip pointers) for each node. There are two methods to choose the skip pointers. In *Random,* each node maintains pointers to a number of random nodes. In *Space-filling skip graph*, nodes are ordered according to their distance and a skip-graph is built over this linear ordering.

7. Mutant Query Plans & Distributed Catalogs

P2P systems offer limited support for complex queries and functions are restricted by the application schema. Bottlenecks frequently appear, due to the absence of mechanisms that combine or manipulate content. These lead to the request of a richer query model that can provide answer to more general queries. For example, we wish to enable content publishers to export information concerning their data using XML and users to be able to query them using the features of the items. Each item has some associated information with it, such as: item name, quantity, price, description, condition, etc. We assume that this information was exported by the item's publisher.

A *mutant query plan* (MQP) [15] is a query plan graph, encoded in XML, consisting of regular query operators, verbatim XML data, references to resource locations (URLs) and references to abstract resource names (URNs). We use mutant query plans to represent a distributed query at the different stages of its processing: from typical query plans with query operators and resource references to query results, containing data. Each MQP must be fully evaluated and then the result must be sent to a network IP address, called the *target*. An MQP is initiated at the client as a regular operator tree and then passes through servers, gathering partial results, until it is fully evaluated in XML data and sent back to the client. A server can decide to mutate the MQP in the following ways: it can apply *resolution* or *reduction* to the tree. *Resolution* can be applied to map a URN to one or more URLs or a URL to its corresponding data. *Reduction involves* the evaluation of a sub-graph plan that has only data in the leaves and the substitution of the results in place of the sub-graph. If the plan is fully evaluated, it is sent to the target. Otherwise, its processing is continued and the plan is forwarded to another server.
Figure 10 shows this process. An MQP arrives at a server encoded in XML, which parses it into an in-memory graph. The server selects which URNs to resolve and the optimizer finds the sub-plans it can evaluate locally and estimates their cost. The policy manager determines the sub-plans it will evaluate. The sub-plans are executed by the query engine component. Finally, the server substitutes the evaluated sub-plans with XML data and sends the MPQ to another server.

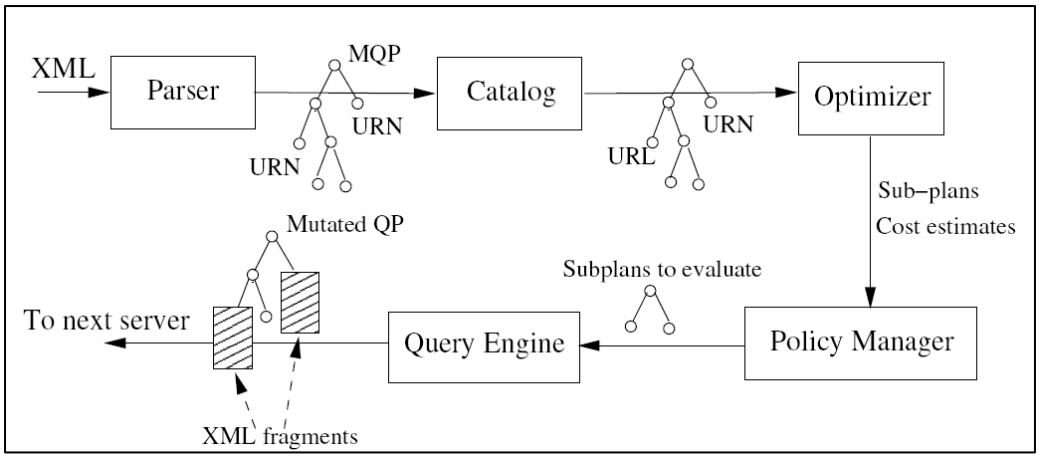

Figure 10: Mutant Query Processing

Figure 11 illustrates an example of a MQP. The operators are selection (σ) and join (⋈) and the target is an IP address. The ellipse shows a sub-graph where resolution can be applied.

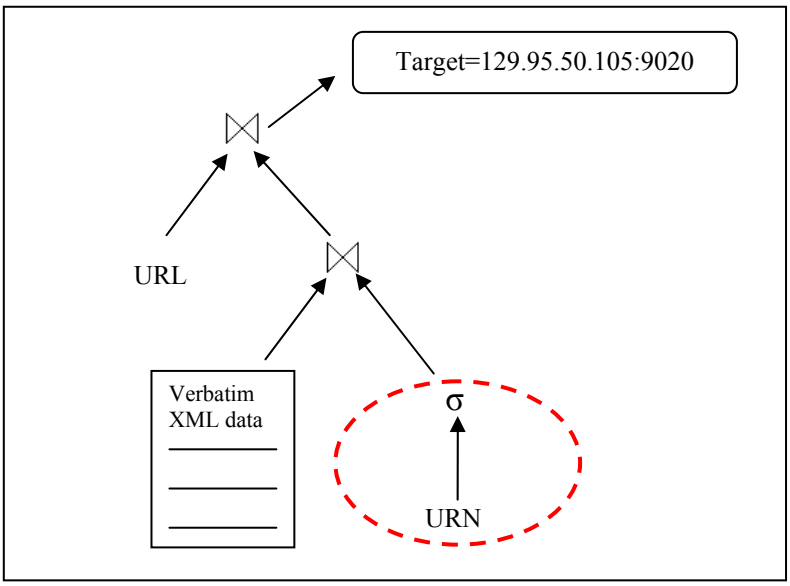

Figure 11: Mutant Query Plan

MQPs offer a viable solution for large, heterogeneous, and autonomous distributed systems. A benefit of using MQPs is that they allow each server to optimize the parts of a query it will run, independently.

Each peer maintains a distributed catalog in order to efficiently route queries to peers with relevant data. This catalog contains mappings from URNs to URLs or from URNs to servers and is consulted during resolution.
Peers define different levels of categories to classify items. We call the set of categorization hierarchies relevant to an application domain a *multi-hierarchic namespace.* Example of hierarchy is location, i.e. USA/Portland. Each item belongs to just one category called its *most specific category* and to all of its parents, e.g., USA/Portland category also belongs in the USA category. Data providers use hierarchies to describe data they serve and consumers use them to form queries. Peers perform one or more roles. *Base servers* maintain data within a category. *Index servers* maintain indices on other servers with relevant data. *Meta-index servers* maintain indices on servers with related data and have more general information than index servers.

8. Replication Techniques

P2P systems use replication techniques to place copies of items at a number of nodes. New replicas of objects reduce the number of hops to answer future queries, since a node can now find some copies closer than before and has more choices for the next search step.
There are many different replication techniques in Unstructured P2P systems [6], such as *Uniform replication,* which places copies of all objects at the same number of nodes. In *Proportional replication,* an object is replicated proportionally to its query probability, while in *Square-root replication* the object is replicated proportionally to the square-root of its query probability.

Another well-known technique is *Owner replication:* after a successful search, the item found is replicated at the requesting node. On the contrary, *Path replication* places copies of the found item at every node along the query path.

We now refer to replication methods applicable to Structured systems. Replication of data in *Chord* is based on the technique of maintaining "successor-lists". Each node stores a list of its *r* nearest successors. An application using Chord can store replicas of the items assigned to a certain key at *k* successors of the key. This way, Chord can inform the higher layer software when successor nodes come and go, in order to propagate new replicas.

There are many replication methods applicable in *CAN*. Most of them are based on CAN's variations. A few are the following:

I. A replication technique in CAN exploits the use of *multiple realities*. Every node is assigned a zone in each reality and holds many independent neighbour sets, one for each reality.
II. To improve data availability, *multiple hash functions* can be used to map a key to several points in the coordinate space. This means that replicas of the pair (key, value) can be found at many different nodes in CAN.
III. A replication strategy can also be based on the technique of *overloading zones*, where multiple nodes share a zone. Consequently, the hash table may be replicated at nodes sharing the same zone, leading to higher availability.
IV. Another idea is the *hot-spot replication*. This strategy involves the following: in case a node finds itself as being overloaded by many requests for a certain data key, it creates a replica of this key to any of its neighbours.
V. Finally, *cashing* can be used so that each node maintains in its cache the keys it recently accessed. Before answering a query, the node searches its cache for the key and if it finds it there, there is no need to forward the query any further.

Replication techniques applied at Content Clustering systems are owner replication and path replication. An alternative method is to use statistics maintained by peers, e.g., about the number of times an item is requested.

In a P2P system having replicas for objects stored at different peers, a new problem arises, concerning the need to keep all replicas that replicate the same data informed of changes. Our goal is to retain the system consistency and provide guarantees for successful results in future queries. To achieve this goal, Datta et al.[16] propose a two-phase updating algorithm. The first phase, called the *push phase*, is initiated by the originator of the update. This peer pushes the new version of the object to a set of peers that have an older version of it, along with a list of peers that have already received the update. Each peer receiving the updated objects, checks if its version is newer than its own and accepts or rejects the object accordingly. Then, it pushes the update to a random set of nodes that have not been updated yet. The second phase is the *pull phase*, where peers coming online or peers that received no update for some time or peers not sure to have the latest update, contact online peers and ask for newly updated items.

9. Comparison of Search in Different Peer-to-Peer Architectures

Peer-to-Peer systems differ in the way they perform content location. In other words, some use informed search, others blind search etc. accordingly to their construction. The following observations are a comparison of different types of search used in P2P systems:

a) **Centralized (Napster) :** existence of central index
b) **Decentralized:** supernodes caching the index of others
   - **Distributed Hash Tables (DHTs – e.g. CAN, Chord):** Informed Search
   - **Unstructured (e.g. Gnutella) :** Blind Search using flooding, etc.

Another study of the performance of different systems using different types of queries has shown that DHT-based systems are good for *point queries* where the search key is known exactly. This means that DHTs cannot efficiently support *partial-match* queries, where shared items have attributes describing their type and properties (e.g. title, composer, performer). We also compare their ability to locate infrequent items, which is called *scope*. Table 2 summarizes the above considerations.

	Partial - match queries	Point Queries	Scope
Centralized - Napster	✓	✓	✓
Decentralized - DHTs	×	✓	✓
Decentralized - Gnutella	✓	✓	×

Table 2: Comparison of support for different types of queries

10. Conclusions

Peer-to-Peer systems play a significant role in distributed file sharing and are very popular among Internet users. The above study has shown that there are many mechanisms for searching inside a P2P system, each with their own advantages and disadvantages. Some of them are: simplicity, robustness, scalability, traffic, etc. There are even some cases where different techniques combine in order to improve the performance of the system, e.g., use of shortcuts over an underlying flooding network. The question as to which mechanism to use in a particular system cannot be easily answered, since one has to take into consideration the particular needs of the current application and the specific needs of the users. It is most certain that the use of P2P systems will increase even more rapidly in the near future.

References

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

[2] Hari Balakrishnan, M. Frans Kaashoek, David R. Karger, Robert Morris, Ion Stoica, Looking up Data in P2P Systems. CACM 46(2): 43-48 (2003)

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001, San Diego, CA, August 2001, pp. 149-160.

[4] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A Scalable Content-Addressable Network.* In Proc. ACM SIGCOMM 2001, San Diego, CA, August 2001

[5] A. Crespo and H. Garcia-Molina Routing Indexes Proc. of the International Conference on Distributes Systems, (ICDCS) 2002

[6] Qin Lv, Pei Cao, Edith Cohen, Kai Li, Scott Shenker Search and Replication in Unstructured Peer-to-peer Networks, International Conference on Supercomputing: 84-95, 2002

[7] Dimitrios Tsoumakos, Nick Roussopoulos: A Comparison of Peer-to-Peer Search Methods. WebDB 2003: 61-66

[8] K. Sripanidkulchai, B. Maggs, and H. Zhang, Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems. Infocom 03

[9] Edith Cohen, Amos Fiat, Haim Kaplan Associative Search in Peer to Peer Networks: Harnessing Latent Semantics Infocom 03

[10] Arturo Crespo and Hector Garcia-Molina Semantic Overlay Networks for P2P Systems

[11] C. Tang, Z. Xu and S. Dwarkadas, Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks, SIGCOMM 03

[12] Ashwin R. Bharambe, Mukesh Agrawal, Srinivasan Seshan: Mercury: Supporting Scalable Multi-Attribute Range Queries. SIGCOMM 2004: 353-366

[13] Ozgur D. Sahin, Abhishek Gupta, Divyakant Agrawal, Amr El Abbadi: A Peer-to-Peer Framework for Caching Range Queries. ICDE 2004: 165-176

[14] Prasanna Ganesan, Beverly Yang, Hector Garcia-Molina: One Torus to Rule Them All: Multidimensional Queries in P2P Systems. WebDB 2004: 19-24

[15] Vassilis Papadimos, David Maier, Kristin Tufte: Distributed Query Processing and Catalogs for Peer-to-Peer Systems. CIDR 2003

[16] A. Datta, M. Hauswirth, K. Aberer Updates in Highly Unreliable, Replicated Peer-to-Peer Systems, The 23rd International Conference on Distributed Computing Systems, May 19-22, 2003, Providence, Rhode Island, USA.