A Survey of Peer-to-Peer Systems

Kostas Stefanidis
Department of Computer Science, University of Ioannina, Greece
kstef@cs.uoi.gr

Abstract

Peer-to-Peer systems have become, in a short period of time, one of the fastest growing and most popular applications. The decentralized and distributed nature of p2p systems leads to living aside the client-server model. In p2p model each node takes both the roles of client and server. As a client, it can query and download its wanted data files from other nodes and as a server, it can provide data files to other nodes. Two main objectives in p2p systems are data location and search for interesting data. In order to present these topics, we survey various structured and unstructured p2p systems. We also study content-based p2p systems that are systems with clusters of nodes, according to the content of their data files. We point replication strategies and techniques and we show how range queries are performed.

1 Introduction

Peer-to-Peer applications are distributed systems, without any centralized control or hierarchical organization. All nodes, which are called peers, have equivalent functionality. Each pair of nodes can communicate each other directly or via other nodes, through the routing protocol.

There are several reasons that make this kind of systems attractive. Firstly, the barriers to growing are low and so, p2p can scale well. They usually do not require any special administrative arrangements and in that way, they can support systems with an increasing number of nodes and data elements. They are robust to faults and when a failure occurs it is easy to recover, because of the data replication and the multiple paths leading to data. Furthermore, p2p systems have good performance, due to the fact that there is a balancing in load because of their decentralized character (there are no nodes that have to serve all requests from all the other nodes). In addition, the availability seems to be high, making the p2p systems very useful.

In p2p systems, there are two approaches to execute a searching operation: the centralized and the decentralized approach. In centralized systems, such as Napster, a basic problem is that in case of a failure there is a network failure, as well. That happens when the node that stores all the information about the network fails. Also, the system has a poor performance when the number of nodes is increased. The decentralized systems, such as Gnutella, Chord, CAN, try to overcome these disadvantages. They have no central directory server and each of them uses different techniques of storing and querying data.

Furthermore, there is an additional distinction of p2p systems between structured and unstructured ones. In a structured topology, data files are placed at specified locations and not at random nodes. This tightly controlled structure enables the system to satisfy queries in an efficient manner. Examples of structured systems are CAN and Chord. In contrast, in unstructured p2p systems, like Gnutella, there is no precise control over the network topology and data files' location. The files' location is not based on any kind of knowledge of the topology. Usually, to find a file the method that is followed is flooding. A node queries its neighbors, its neighbors their own neighbors and so on, for a specific number of steps. Also, an unstructured system seems to not scale well; when many nodes join the system many messages are produced in the network.

Paper outline. In this paper, we underline the features of structured and unstructured networks, in order to survey p2p systems. The rest of the paper is organized as follows. At first, in section 2 we present Chord and Can, which are examples of structured systems. Then, in section 3, we investigate unstructured systems. We present the Gnutella system and a variety of improvements. Furthermore, we discuss the use of routing indices. Section 4 focuses on content-based p2p systems and section 5 presents replication strategies and replication techniques. In this section, we also show how updates are spread. Section 5 is referred to how range queries are performed and section 6 states an additional technique that provides a way to use a full-featured query language. Finally, section 7 concludes the paper with a summary of our study.

2 Structured P2P Systems

There are many different p2p systems, each one with various advantages and disadvantages. They differ both in their object query mechanism and in their logical topology. In order to understand p2p systems we classified them into structured and unstructured. In this section, the structured systems Chord and CAN, are presented. In such systems, data files are placed not at random nodes, but at specified locations.

2.1 Chord

Chord [18] is a distributed lookup protocol. In this protocol all nodes are uniformly distributed in a ring, which is called *identifier circle* or *Chord ring*. Data files are also uniformly distributed over the network. In particular, Chord maps a key, i.e. an m-bits identifier, to the nodes and to the data files using *consistent hashing*. The method of consistent hashing uses the SHA1 algorithm (Secure Hash Algorithm - [4]). Nodes are ordered in the ring according to the modulo of the key with the number 2^m. A data file with key k is stored on the first

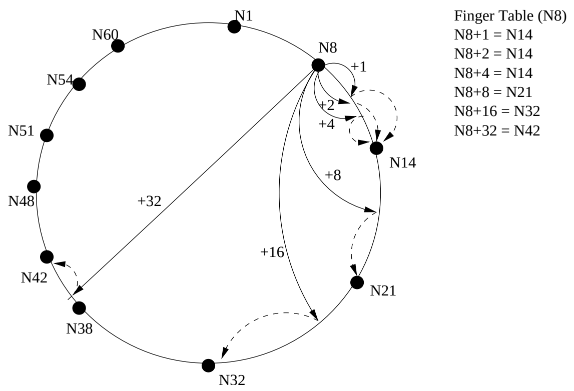

Figure 1: The Chord Ring.

node whose identifier is equal or follows k in the identifier space. This node is named *successor* node.

Each node needs only to be aware of its successor node on the circle (every node is linked to the next one with a successor pointer). Queries for a given identifier can be passed around the circle via the successor pointers until they first encounter a node that succeeds the identifier; this is the node the query maps to. To accelerate this process, each node maintains a routing table with information for only *O(logN)* other nodes. In this way, the queries are executed more efficiently. In particular, each node knows all the other nodes carrying the nearest, largest key among all the keys that are at a distance of an increasing power of 2. This knowledge is stored in a table with m entries, called *finger table*. A finger table entry includes the Chord identifier, the IP address and the port number of the relevant node. The first entry is its immediate successor on the circle. The i^{th} entry of the finger table for a node k has the identifier of the first next node from the node $(k + 2^{(i-1)})mod(m)$, where $1 \leq i \leq m$. When we look up for a key at node k, we first check if the key is located between k and its *successor*. If this is true, *k's successor* is the node that has the key that we are looking up. Otherwise, k searches its finger table to find the node with the first largest key from the key we are looking up. The procedure continues until the node that stores the key is found. In a system with N nodes, when a node executes a lookup operation, *O(logN)* messages are transmitted to other nodes. An example of a Chord ring, where is represented the finger table of the $N8$ node, is shown in Fig. 1.

Nodes can join and leave at any time. In both cases, it is necessary to move a small amount of keys to different nodes. These nodes are now responsible for the keys. More specifically, when a new node joins the network some of the keys that are assigned to the successor of the new node, must be assigned to the new node. Similarly, when a node leaves the network, all its keys are assigned to its successor. These are the only keys that must be moved, to ensure the system's consistency.

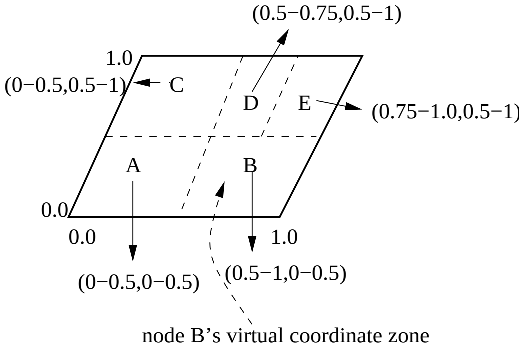

Figure 2: Example $2-d$ space with 5 nodes.

2.2 CAN

CAN (Content-Addressable Network) is a distributed hash-based infrastructure that provides fast lookup functionality. Each node stores a chunk, which is called zone of the hash table. Also, a node maintains information about its neighbors in the network. Requests for keys are routed to those nodes that their zones have the corresponding keys.

CAN [15] is designed around a virtual d-dimensional cartesian coordinate space on a *d-torus*. This coordinate space is partitioned among all the nodes of the system. In that way, each node has its own zone. For example, Fig. 2 shows a 2-dimensional $[0,1] \times [0,1]$ coordinate space partitioned between 5 nodes. The virtual space is used to store *(key,value)* pairs. A key k is assigned to a point p using a hash function. Then, the corresponding *(key,value)* pair is stored to the node whose zone includes point p. To retrieve this key and the corresponding data, a node executes the same hash function to find point p. For the routing operation each node maintains the IP addresses of the nodes of its neighborhood. It also, has information regarding the zones of the network. In a d-dimensional space, two nodes are neighbors if their coordinates are overlapped along $d-1$ dimensions and are adjoin along 1 dimension. A node, using the set of its neighbors, routes a message towards its destination by forwarding it to the neighbor, which is closest to the destination. Each node maintains information about $2d$ neighbors. The average routing path length is $(d/4)(n^{1/d})$ hops.

As referred above the coordinate space is partitioned among all the nodes of the system. When a new node joins the system, an existing node splits its zone and the new chunk is assigned to the new node. This operation is performed in three steps. First, the new node discovers an existing node. Then, it finds the node whose zone must be split and lastly the neighbors of the split zone are notified that routing can include the new node. Furthermore, CAN supports nodes leaving. In this case, i.e. when a node leaves the network, its zone is assigned to one of the existing nodes.

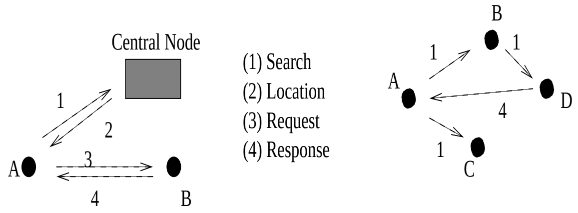

Figure 3: The Napster (left) and the Gnutella (right) model.

3 Unstructured P2P Systems

In this type of networks, the logical p2p topology is often random or is based on a nodes' hierarchy, i.e. there exists super-nodes. A query is executed hop-by-hop through the network until a success, a failure or a timeout occurs. In this section, we first present Gnutella, which is an unstructured p2p system, and then a variety of techniques for improving the performance of unstructured systems.

3.1 Gnutella

Gnutella [2] is one of the earliest decentralized p2p file sharing systems. To join the system, a new node first must connect to one of several known nodes that are already in the system. The new node notifies the Gnutella node for its existence with a join message. Then, this node notifies its neighbors that a new node has connected to the system; they notify their neighbors for the new node, and so on. This process continues until the TTL (Time-To-Live) becomes 0 (TTL has an initial value which is decreased by 1 in every hop). To find a file, a node broadcasts messages to its neighbors. The most typical method is *flooding*. A node sends a query to its neighbors on the network. In turn, its neighbors forward the query to all of their neighbors until the query has traveled for a certain radius, according to the TTL.

Gnutella is completely decentralized, and so there is no single point of failure, such as in Napster. In *Napster* ([3]) a node stores all the information about the network and in the case this node fails, there is a network's failure. The above systems are shown in Fig. 3. On the other hand, using flooding, many messages are transmitted between the nodes, causing large loads on the network.

Improvements. In order to overcome this disadvantage several techniques are proposed. In [13] two mechanisms are presented: *expanding ring* and *random walks*. In the first one, a node starts the flooding method with a small TTL. In case that the search is not successful, the node increases the TTL and starts another flooding. The process is stopped when the data file is found. This mechanism achieves best results when it is possible that flooding will satisfy a query with a small number of hops. In a different case, the mechanism produces even bigger loads than the standard flooding mechanism. In the random walks mechanism, the requesting node sends k query messages and each query message takes its own random walk, i.e. forwards the query to only one randomly chosen neighbor at each step. This mechanism effects some kind of local load balancing, because each node, which is chosen to be forwarded a query, is selected randomly. Also, the number of messages transmitted in the network is reduced. The most important drawback of this mechanism is its highly changeable performance. The above remarks are referred in [20].

3.2 Routing Indices

In Gnutella, queries are propagated from node to node until the data files are found, without any use of indices. This approach has as a disadvantage the cost of flooding the network when a query is executed. The use of *routing indices*, which are proposed in [8], allows nodes to forward queries to these nodes that possibly have the answers. So, if a node cannot answer a query, it uses its local routing index in order to forward the query to a selected set of neighbors, instead of forwarding the query to all its neighbors (flooding) or to a randomly selected number of neighbors. In following, we present the compound, the hop-count and the exponential routing indices.

Compound Routing Indices. In general, we use routing indices because they provide to a node a set of its neighbors that are appropriate to send a query to. Each node has a local index for finding local data files quickly, when a query is executed. Data files are divided into categories, according to their topic. Also, the node has a *compound routing index* containing the number of files along each path and the number of files on each topic of interest. For a node the number of its available paths is equal to the number of its neighbors. For finding how 'good' is a node, in order to propagate a query to it, the number of data files that may be found in its path is measured. In case that a new connection between two nodes is established, nodes must inform their routing indices and so the indices are updated, including the new data files that can be accessed. In a similar way, the routing indices are updated when a node leaves the network.

Hop-Count Routing Indices. The main drawback of using compound routing indices is the fact that they take into account only the number of data files in a path and not the distance cost to reach them. This leads to an alternative data structure: the *hop-count routing indices*. This kind of indices store aggregated routing indices for each hop up to a maximum number of hops. This number is named *horizon* of the routing index. For instance, when we have a hop-count routing index with an horizon of 2 hops, we store the number of data files that can be accessed with 1 hop and also the number of data files that can be accessed with 2 hops. Comparing this approach with the previous one, we note that here, there is no information about the number of data files that are stored to the nodes beyond the horizon. In addition, we observe that a path

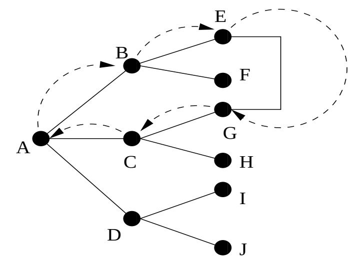

Figure 4: Cycles and Routing Indices.

which seems to be better for answering a query with k hops, it might be not the appropriate one when we use a $(k+1)$ hops distance. For example, when we compare 2 paths, A and B, to find the best one, A can give us 20 results (data files) for a specific query and B 10 with k hops, but on the other hand A might give us 25 results for the same query and B 35 with $(k+1)$ hops.

Exponentially Aggregated Routing Indices. The previous approach takes into account the number of hops, although the storage amount increases. An alternative index structure that is introduced, is the *exponential aggregated routing index*, which overcomes this problem but it looses in accuracy. In particular, each entry of the index has a value computed as:

$$\Sigma_{j=1...th}(goodness(N[j],T])/F^{j-1}),$$

where th is the height and F the fanout of an assumed tree that represent the network topology, *goodness()* is the compound routing index estimator, *N[j]* is the summary of the local index of neighbor j of N and T is the topic of interest of the entry.

In any of the above cases, the process of creating and updating indices is more complicated, when there are cycles in the network topology. An example, where there is a cycle, is shown in Fig. 4. The authors of [8] provide the following solutions:

- *No-op solution*, where there is no modifications to the algorithms. This approach can be applied to hop-count and to exponential routing indices.
- *Cycle avoidance solution*, where the creation of cycles is not permitted.
- *Cycle detection and recovery*, where cycles are detected and the effects that they create are canceled.

In Table 1, we summarize basic features of p2p systems that are presented above.

4 Content-Based P2P Systems

P2p systems that are presented above have low cost for sharing information, privacy and autonomy. However, query processing some times is not very efficient and does not scale well. These drawbacks arise because many p2p systems

Table 1: A comparison of various p2p systems

P2P Systems	Napster	Gnutella	Chord	CAN
Degree of Centralized	*centralized*	*decentralized*	*decentralized*	*decentralized*
Network Topology	-	*unstructured*	*structured*	*structured*
Parameters	*none*	*none*	*N-number of nodes*	*N-number of nodes and d-number of dimensions*
Routing State	*constant*	*constant*	$logN$	$2d$
Query Path Length	$O(1)$	$< TTL$	$O(logN)$	$O(dN^{1/d})$
Fault Tolerance	*poor*	*poor*	*random*	*random*

create a random graph that represents the network topology, where queries are propagated from node to node in a blind manner. Furthermore, there are systems where data files are placed not at random nodes but at specified locations using hash functions. Such systems have good performance for point queries, but they are not efficient for text or range queries. In this section, we present different approaches that aim to improve query performance. We first discuss the interest-based shortcuts protocol, and then we present various works that includes clusters of nodes, according to the contents of their data files and their interests.

4.1 Interest-Based Shortcuts

In [17], authors propose a content location solution. In this approach, nodes loosely organize themselves into an interest-based structure on top of the existing Gnutella network. The principle of *interest-based locality* states that if a node has a particular data file that one is interested in, it is possible that it will have other files that one is interested in as well. This principle is used to create *shortcuts* from a node to another. These nodes share many similar interests. In addition, shortcuts not only provide a loose structure on top of Gnutella, but also are compatible with many other mechanisms, such as DHTs and hybrid centralized-decentralized architectures. In that way, at first a query is sent to nodes through shortcuts, avoiding flooding or other methods, and only if shortcuts fail, the query is flooded to the entire system. A topology with shortcuts is shown in Fig. 5. The shortcuts are represented with the bold lines.

When shortcuts are used in a system, there must be a way to select which shortcut should be used. Authors in [17] propose a *ranking* of shortcuts based on their utility. So, the useful shortcuts are posed on the top of the list. A node, starts from the top of the list, checks all shortcuts one after the other, until the searching data file is found. Each node continuously updates its ranking list, based on the performance of shortcuts. This allows nodes to adapt to dynamic changes. The ranking of shortcuts is created by using metrics, such as probability of providing content, latency of the path to the shortcut, available bandwidth of the path, amount of content at the shortcut, load at the shortcut or a combination of the above.

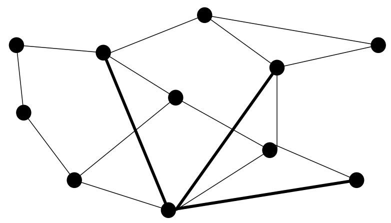

Figure 5: Paths with *shortcuts*.

4.2 Clustering Techniques

An alternative kind of decentralized p2p systems is based on the creation of nodes' clusters. In each cluster, all the nodes have similar interests. These systems retain the desirable properties of unstructured systems and support partial-match queries. In the following subsections, we present 3 systems that are organized according to the content of data files stored at each node. Firstly, we discuss the *associative overlays* ([6]), and then the *semantic overlay networks* (SONs - [9]). At last, we present how *semantic vectors* ([19]) can be used, to express the similarities between queries and data files.

4.2.1 Associative Overlays

A guided search is proposed in [6]. This kind of search is neither a blind search used by unstructured systems nor a routed search used by structured systems. Here, queries are propagated to nodes that have relevant data files. These nodes are semantically similar and belong to the same cluster. All nodes that belong to a cluster satisfy a predicate. This set of nodes is called *guide rule*. The guide rules define the network topology. In Fig. 6, we can see a pictorial example of the sets of nodes associated with two overlapping guide rules. Each node maintains a small list of other nodes that belong to the same guide rule. A search process in a guide rule is performed like the blind search in unstructured systems. Furthermore, a node has the capability to select a guide rule, among those that belongs to, in order to execute a search.

A guide rule is a set of nodes whose index satisfies a predicate. A *possession rule* is a particular kind of a guide rule. For instance, a possession rule may check for the presence of a certain entry in the index. In this way, a node participates in a rule if it has a specific item.

Two search algorithms are proposed: the *Rapier* (Random Possession Rule) and the *Gas* (Greedy Guide Rule) algorithm. In the first one, a possession rule is selected at random and then a blind search is performed. In the latter, we do not select randomly a guide rule, but we choose this guide rule that probably leads to an efficient search.

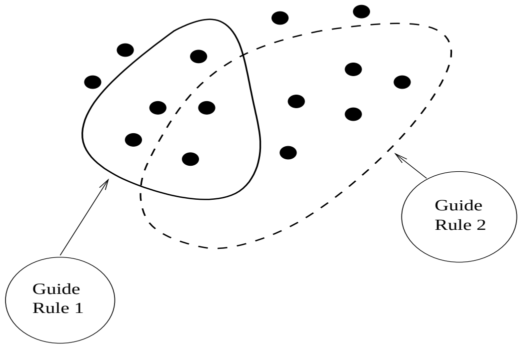

Figure 6: Two overlapping guide rules.

4.2.2 SONs

In a similar way, *Semantic Overlay Networks* (SONs - [9]) consist of clusters of nodes. Each cluster includes nodes that are semantically related. Two nodes are semantically related when the content of their data files are similar. All connections are between nodes that belong to the same SON, without the need that in a SON all nodes are connected to each other. Furthermore, a node might belong to more than one SON. Queries are processed first by finding the appropriate SON to answer it. Then, the query is propagated to this SON and finally, is performed a blind search in the specific SON. This process reduces the time to answer a query.

Moreover, in [9] is introduced a different implementation of SONs, which is called *Layered SONs*. This approach improves further the query performance. It uses a *zipfian data distribution*, while each node can decide which SONs to join according to the number of the related data files that are stored at the node.

4.2.3 Semantic Vectors

Another way to exploit the similarities of content of the nodes' data files is to place them (or their indices) to specific nodes. For each data file a vector is created, according to its content. This vector is used to place the data file. Also, each query has a vector. The similarity between a file's vector and a query's vector, leads the query to an appropriate node, i.e. a node that may have the result of the query. This approach is presented in clearness in [19].

In Fig. 7, it is shown a tree form that represents the above content based p2p systems.

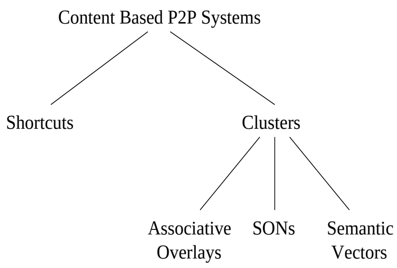

Figure 7: Content Based P2P Systems.

5 Replication

One way to improve the performance of a system is to replicate data files on several nodes, before a query is resolved. In this section, we survey various replication techniques that are applied to p2p systems.

Initially, specific replication strategies that indicates how relicas are distributed across the nodes are proposed in [7]. Uniform, proportional and square-root replication are examples of the above strategies. In *uniform replication* strategy all data files are replicated at the same number of nodes, even though some data files are more frequently requested than others. Using this technique, the required maximum search cost is minimized. An alternative strategy is *proportional replication*. Here, the number of replicas for a specific data file is proportional to the query probability of the data file. So, if nodes store only the data files that are requested for, the replication distribution is almost proportional to the query distribution. Although queries for popular data files are satisfied efficiently because there are many replicas for the requested data files across the network, queries for unpopular data files require higher search cost. Between uniform and proportional replication is *square-root* replication. In this strategy, the replicas of a specific data file is proportional to the square root of its query probability. Square-root replication provides a balance for searching popular and unpopular data files.

5.1 Discussion on Structured Systems

Additionally, structured p2p systems use specific methods to improve their performance and to increase their availability. When data files are replicated, the load of the system is balanced and usually there are copies nearby the requestor. Also, the availability is higher since we can use replicas in the case of failures and nodes departures. On the other hand, the amount of storage increases and we must maintain the consistency of the replicated data. In this section, we introduce such methods that are applied to Chord and CAN [18, 15].

Chord. A strategy for *metadata replication* that is used in Chord is based on *successor lists*. With a successor list a node maintains information about the

r next nearest successors on the ring. This list guarantees the correctness of a search.

CAN. In CAN a replication technique is based on *realities*, which are independent coordinate spaces. Each node is assigned to a zone in each reality. Thus, if CAN has r realities, a node is assigned to r zones, one for each reality. Replicas of the hash table are stored in each reality. In this way, when there are multiple realities, a pointer for a specific data file is stored at more than one different nodes.

In order to improve data availability, we can also use k different *hash functions* to map a key onto k points in the coordinate space, and so replicas of the *(key,value)* pair are placed to k different nodes in the system. In this case, the *(key,value)* pair is not available only when all k replicas are not available at the same time.

Furthermore, replication is also used in the *overloading coordinate zones* technique. According to this technique, multiple nodes may share a zone. So, replicas of the hash table are placed to all nodes that have been assigned to the same zone, ensuring higher availability.

In general, when a node conceives that receives many requests for a specific data key, it may replicate this data key at each of its neighbors. A node that holds a replica can be used to satisfy related requests, reducing the load of the node that holds the 'original' data.

A particular kind of replication is *caching*. A node can maintain a cache of data keys that are recently accessed. So, it first checks its own cache in order to find the requested data key. Only if the data key is not found, the request is forwarded to other nodes.

5.2 Discussion on Unstructured Systems

Moreover in [7, 13], authors present replication techniques for unstructured p2p systems. The first one is called *owner replication*. When a search is successful, the desirable data file is replicated to the node that requests for it. This technique is used in Gnutella. Alternatively, in *path replication*, when a search is successful, the desirable data file is replicated to all nodes along the query path, i.e. the path from the node that asks for the data file to the node that provides it. This technique is used in Freenet ([1]) and in specific circumstances may decrease the system's performance. In a different approach, the idea of *random walks* is used (random walks are illustrated in section 3.1). So, in *random replication* we count the number of nodes on a query path, say p, and we select randomly p of the nodes that the walks visited to replicate the data file. This technique seems to be harder to be implemented.

5.3 Spreading Updates

When data files are replicated at many nodes, consistency must be maintained among the nodes. In general, we can separate replication into eager and lazy methods. *Eager replication* keeps all replicas synchronized at all nodes, by updating all replicas in a single transaction. In reverse, *lazy replication* propagates asynchronously replicas' updates to other nodes after replicating transaction commits. Most times, p2p systems use lazy replication because of its lower cost.

Furthermore, in [11], several strategies for spreading updates are proposed. The following strategies are typical examples of epidemic algorithms.

- *Direct mail*. When an update occurs, it is immediately mailed from its originating node, i.e. the node where the update occurs, to all other nodes. The main advantage of this strategy is that updates are propagated very quickly.

- *Anti-entropy*. Periodically, every node selects randomly another node and resolves any differences between them, by exchanging content. There are three ways to execute anti-entropy, called push, pull and push-pull. In *push* method, when an update occurs, the originating node propagates an update message to all nodes that hold replicas of the updated data file. In *pull* method, all nodes that hold replicas, ask the 'primary' node for updates. The last method, named *push-pull*, is a combination of the others. The anti-entropy strategy is reliable, but quite slow.

- *Rumor mongering*. When a node receives a new update, it periodically selects randomly another node and checks if this node has seen the update, in order to send it to it. A node stops to send the update to other nodes, when many other nodes have seen it.

Moreover in [10], is proposed an update strategy, which is based on a *hybrid push/pull rumor spreading algorithm*. Nodes are many times offline. When these nodes are connected again, they must be informed about the updates that they have missed. This update scheme has two phases: the push and the pull one. The node where the update occurred, initiates the *push phase*. The node propagates the new update to a subset of nodes that hold a corresponding replica. They propagate, in turn, the update to another subset of nodes that they have not been updated yet, and so on. This process is similar to flooding method with constrains, because it is executed for a specific number of steps. Furthermore, it avoids many duplicate messages, while propagating the rumor. On the other hand, the *pull phase* is initiated either by a node that has been offline and then gets connected and needs to update its replicas or by a node that does not receive updates for some time or by a node that receives a pull request and is not sure that it has the freshest replica. The above hybrid spreading algorithm provides probabilistic guarantees for acceptable results for queries and results no strict consistency.

In Table 2, we integrate replication strategies and algorithms that are presented above.

Table 2: Replication Methods

General Replication Strategies	**Replication in Structured Systems**	**Replication in Unstructured Systems**	**Spreading Updates**
$uniform$	*successor lists (Chord)*	*owner replication*	*direct mail*
$proportional$	*multiple realities (CAN)*	*path replication*	$anti-entropy$
$square-root$	*multiple hash functions (CAN)*	*random replication*	*rumor mongering*
	overloading coordinate zones (CAN)		*hybrid push/pull rumor spreading*

6 Approaches for Range Queries

Most p2p systems support only simple lookup queries. For example, these queries include only the *select* operator. However, general p2p applications require a richer query model. So, in this section, we present several systems that are capable to support *range queries*.

6.1 Mercury

At first in [5], it is proposed a scalable routing protocol for supporting multi-attribute range queries, called *Mercury*. Mercury creates a *routing hub* for each attribute of the database schema. Each hub is a logical set of nodes, and so a physical node can participate in multiple logical hubs. Each new data file, according to its attributes, is sent to all the corresponding hubs. In reverse, each query that is referred to a set of attributes is propagated to only one of the corresponding hubs.

A hub in Mercury organizes its nodes into a Chord-like ring. The main difference with Chord is that in Mercury is not used random hash functions for placing data files but each node in the hub is responsible for a range of values for the specific attribute. So, when a query is posed in a hub, it is routed to the node that is responsible for the first value of the range and via *successor pointers* that each node maintains, the query is spread along the ring, until it arrives to the node that is responsible for the last value of the range. Also, a node maintains pointers to its *predecessor* to keep consistent the ring with lower cost when a node's join or leave occurs. In addition to the successor and predecessor pointers, a node can maintain a set of *k long-distance pointers* (as in Chord) to reduce the routing cost in the hub ring. Furthermore, a node maintains a third kind of pointers, called *cross-hub pointers*. These pointers are used to connect one hub to another.

In Fig 8, we see a pictorial example of hubs in Mercury. We suppose that there are two hubs, x and y, each for an attribute. The attributes take values from 0 to 320 and each node is responsible for a partition of the range, as it shown in the figure. A data file that has values $100 \leq x \leq 120$ and $200 \leq y \leq 220$ is stored to both hubs, i.e. to nodes b and g, respectively. In reverse, a query

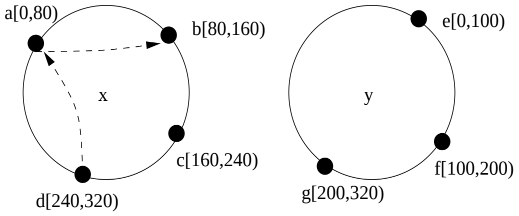

Figure 8: Routing data files and queries in Mercury.

for $50 \leq x \leq 150$ and $150 \leq y \leq 250$ is routed to only one of the hubs, e.g. to x. So, if the query is posed to node d, it is routed via pointers to nodes a and b, because these nodes are responsible for the result.

In order to reduce the storage amount of the replicated data files (there is a replica for all data files in each relative hub), nodes can hold pointers to other nodes that hold the particular data file. In this case, it is needed an additional step to catch the node that is responsible for the result of a query.

6.2 A Framework for Caching Range Queries

An alternative method to evaluate range queries is proposed in [16]. This method is based on the multidimensional CAN system. Nodes cache the results of range queries and use them to answer future range queries.

More specific, the system maintains a global database schema that is known from all the nodes. Nodes store range partitions of the data files and cooperate with each other to answer queries, instead of asking direct the database. This system is based on CAN and uses a 2d virtual space; two dimensions for each CAN dimension. The virtual space is partitioned among the nodes. A node is responsible for a part of the virtual space, accordingly to the range of the data files that stores. This part is called zone. In this approach, not all the nodes are responsible for a zone.

A data file is referred to a range. This range is assigned to a point in the virtual space. This specific point belongs to a zone, and so the data file is stored to the node that is responsible for that zone. Similarly, a query for a particular range is assigned to a point in the virtual space. The query is routed to the node that is responsible for the zone that includes this point. The routing is executed such in CAN. We first propagate the query to the neighbor that is closest to the point, and so on. For the above reason, each node maintains a routing table with the IP addresses of its neighbors. When a node gets the result for its query, it caches the result to use it in future, for itself or for another node. A main drawback of this approach is that it can not turn to account the half CAN space, because a point demonstrates a range.

In following, we present how the running example of the previous section acts in this approach. As before, there is a virtual space for each attribute. Fig. 9 shows a partitioning of the virtual space for attribute x. As it shown, four

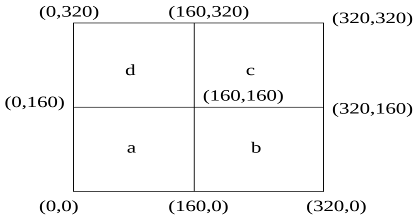

Figure 9: Partitioning of the virtual space for x.

nodes are connected in the network. So, the data file, which is described above, is assigned to node a and if the query is posed for instance to node d, it is routed to node a, because this node is responsible for the corresponding range.

6.3 SCRAP and MURK Approaches

In [12], two different approaches for answering multi-dimensional range queries are proposed. These approaches are called SCRAP (*Space-Filling Curves with Range Partitioning*) and MURK (*Multi-Dimensional Rectangulation with KD-Trees*). Both of them are divided into two components. With a partitioning strategy, data files according to their content, are distributed to a set of nodes, and with a routing strategy, queries are passed to nodes that can serve them. In following, there is a description of the above approaches.

SCRAP. In order to partition the multi-dimensional data, at first data is mapped into a single dimension using a *space-filling curve*. Examples of such curves are the *Hilbert curve* and *z-ordering*. Then, data is partitioned across all the available nodes, in a way that ensures that each node is responsible for a continuous range of values. As far as routing concerns, the multi-dimensional range query is divided into a set of range queries and each one is propagated to a node with a relative range, using for example a *skip graph*. The result of the initial query is the aggregation of subqueries' results.

MURK. This approach uses KD trees to model the storage of data files. In each leaf node is assigned a rectangle that is managed by a node. When a node joins the system a leaf's rectangle is split into two parts with equal load. On the other hand, when a node leaves the system, its rectangle is merged with the rectangle of the sibling node in the KD tree. This approach presents many similarities with CAN. However, if there is a need for a rectangle's splitting, the two new parts have equal load, instead of equal space. The queries are routed such in CAN. A node maintains pointers to connect with its neighbors and forwards the queries to them. Furthermore, there are additional pointers to few other nodes, called *skip pointers*, in order to accelerate the routing operation.

Table 3: Approaches for Range Queries

Approaches	**Data Files' Location**	**Routing Queries**
Mercury	*Data files are stored to all the corresponding hubs*	*Queries are routed to only one of the corresponding hubs*
Cache Range Queries	*Data files according to their range are stored to specific zones*	*Queries are propagated to the neighbor that is closest to the result*
SCRAP	*Multi-dimensional data files are mapped into a single dimension and they are partitioned across nodes*	*Multi-dimensional range queries are divided into a set of range queries and each one is routed to a node*
MURK	*KD leaf nodes store data files with relative ranges to node's rectangle*	*Queries routed to relative nodes via neighboring pointers*

Table 3 presents in few words the above approaches that are capable to support range queries.

7 Distributed Query Processing

Apart from the above systems, in [14] authors propose a kind of a query plan, which is called *Mutant Query Plan* or simply MQP. This approach provides a method that allows to use a full-featured query language. An MQP is an algebraic query plan graph. In this plan, data is encoded in verbatim XML. The plan has information about the locations and the abstract names of the resources (URLs and URNs). Furthermore, the plan knows the network address (IP) to send the fully evaluated MQP, i.e. the result of the query, to.

In general, the process of the plan is described in the following steps.

- The MQP starts at the node that posed the query.
- Then, it propagated from node to node, while partial results are selected.
- When the whole MQP is evaluated, the result is returned to the initial node.

Figure 10 represents the above process in details. When a MQP arrives at a node, the node *parses* the plan and resolves the corresponding with the query URNs. Locally, at each node, there is a *catalog.* This catalog maintains information about the mappings from URNs to URLs. Using this catalog, all URNs it is possible, are replaced with the corresponding URLs. Then, the *optimizer* finds parts of the plan that can be evaluated locally, optimizes them and estimates their costs. A *policy manager* selects the parts of the plan that will be executed locally by the *query engine* (the data of the partial result is encoded in verbatim XML), and finally the new plan, which includes the partial results from previous executions, is propagated to the next node to continue the execution of the query.

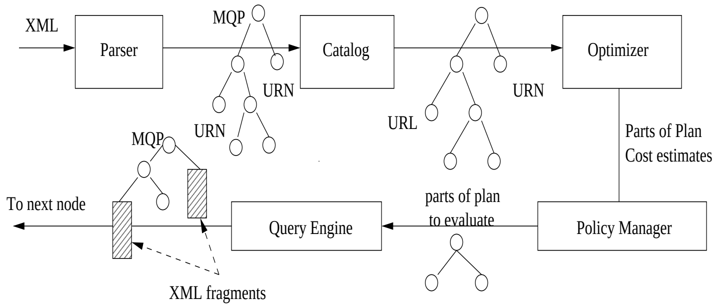

Figure 10: Mutant Query Processing.

For instance, we assume that we are looking for CDs in Portland with less than $10. In this example, the client that pose the query has a list of favorite songs. Also, there is a list of CD titles and we use an online track-listing service to connect the resources. In Fig 11, we can see the mutant query plan for this query. This plan includes several operators (*select, join, display*), two URNs and a piece of XML.

As referred above, this model of p2p systems maintains distributed catalogs, that are catalogs with mappings from URNs to URLs. In order to categorize data files, the nodes that provides data files (*data providers*) use multi-hierarchical namespaces to describe the category of files, as concerns its content, that they serve. In a similar way, the nodes that form queries (*data consumers*) use these namespaces to express their queries. In this approach, there are many different kind of roles that a node can play. So, a *base server* stores data files within a category, an *index server* stores information about base and index servers that have the same category of data files with its own and a *meta-index server* is an index server which stores additional information about the hierarchy level of a data file. Furthermore, a *category server* is responsible to answer queries about the levels of the hierarchies. Each node can play more than one of the above different roles.

8 Conclusion

In conclusion, this paper presents an overview of p2p systems, underlining the features of them. The system that is best suited depends on the application and its required functionalities. Several of these schemes are implemented in applications, such as sharing of music files, multi-player games, replication of electronic yellow pages or address books, the provisioning of location services, and the distribution of workloads of mirrored websites. In particular, here we discuss certain structured and unstructured p2p systems. We also study content-based systems, that are systems with cluster nodes according to the contents of their data files and their interests. Then, we investigate how replication techniques are applied to p2p systems and several algorithms for spreading

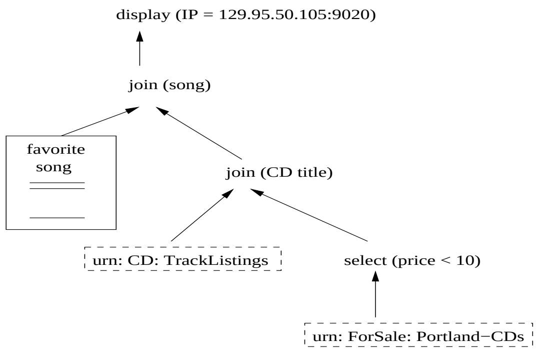

Figure 11: A Mutant Query Plan.

updates. Finally, the paper concludes with a discussion on range queries.

References

[1] Freenet. http://freenetproject.org.

[2] Gnutella. http://www.gnutella.com.

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

[4] Public Key Cryptography for the Financial Services Industry - part2: The Secure Hash Algorithm (SHA-1). American National Standards Institute, Technical Report, 1997.

[5] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. *In Proceedings of the SIGCOMM'04 Conference*, 2004.

[6] E. Cohen, A. Fiat, and H. Kaplan. Associative Search in Peer to Peer Networks: Harnessing Latent Semantics. *In Proceedings of the IEEE INFOCOM'03 Conference*, 2003.

[7] E. Cohen and S. Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. *In Proceedings of the ACM SIGCOMM'02 Conference*, 2002.

[8] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. *In Proceedings of the Distributed Computing Systems Conference*, 2002.

[9] A. Crespo and H. Garcia-Molina. Semantic Overlay Networks for P2P Systems. Stanford University, Technical Report, http://www-db.stanford.edu/peers, 2003.

[10] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. *In Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[11] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. *PODC*, 1987.

[12] P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule them All: Multi-dimensional Queries in P2P Systems. *In the International Workshop on the Web and Databases*, 2004.

[13] Q. Lv, P.Cao, E.Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. *In Proceedings of the ACM ICS'02 Conference*, 2002.

[14] V. Papadimos, D. Maier, and K. Tufte. Distributed Query Processing and Catalogs for Peer-to-Peer Systems. *In Proceedings of the CIDR'03 Conference*, 2003.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. *In Proceedings of the ACM SIGCOMM'01 Conference*, 2001.

[16] O. D. Sahin, A Gupta, D. Agrawal, and A. El Abbadi. A Peer-to-Peer Framework for Caching Range Queries. *In Proceedings of the ICDE'04 Conference*, 2004.

[17] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems. *In Proceedings of the IEEE INFOCOM'03 Conference*, 2003.

[18] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *In Proceedings of the ACM SIGCOMM'01 Conference*, 2001.

[19] C. Tang, Z. Vu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. *In Proceedings of the ACM SIGCOMM'03 Conference*, 2003.

[20] D. Tsoumakos and N. Roussopoulos. A Comparison of Peer-to-Peer Methods. *In Proceedings of the Web and Databases Conference*, 2003.