GALANX: An efficient Peer-to-Peer Search Engine System

Paper by Yuan Wang, David J. DeWitt, Leonidas Galanis

Write-Up by
Steffen Metzger

Tutor: Christian Zimmer

Peer-to-Peer Information Systems
winter semester 2004/05

Saarland University
January 18, 2005

Contents

1 Introduction **3**

2 Key Idea and State of the Art in P2P Routing **4**
- 2.1 The Key Idea . . . 4
- 2.2 Wanted System Properties . . . 5
- 2.3 Simplifying Assumptions . . . 6
- 2.4 State of the Art in P2P routing . . . 7

3 A first DHT solution **8**

4 The GALANX design **11**
- 4.1 GALANX Peer Indices . . . 11
- 4.2 Building GALANX Peer Indices . . . 13
  - 4.2.1 Direct Index Construction . . . 13
  - 4.2.2 DHT Index Construction . . . 19

5 Implementation and Tests **21**
- 5.1 A Prototype Implementation . . . 21
- 5.2 Experimental Setup . . . 22
- 5.3 Results . . . 23

6 Conclusions **26**

7 References **27**

1 Introduction

Having a look at large data networks like the Web, corporate LANs or perhaps file-sharing networks, we have a lot of information providers(servers) and even more users(clients) searching for specific data. Our aim is to find the answer for search queries(all the data files which correspond to the query) fast and efficient.

The most famous search system in the Web, Google, does this using a centralized approach. It crawls the pages in the Web and maintains a central index with all the relevant information it can gather by crawling the network.

Therefore queries can be answered very fast, because the central server can directly access its index and give back the pages with corresponding content. But on the other hand it is obvious, that it's not scalable, expensive and vulnerable by its centralized design. And it is also never up-to-date and cannot cover the whole web, because crawling all pages every second and updating instantly the entries for all pages would need an immense bandwidth and processing power, not only at the central search server, but also on every server providing data.
So it seems to be impossibly to get a really reliable, up-to-date and fully covering search system with a Google-like centralized approach using a central index and several crawlers.

So, a Peer-to-Peer(P2P) solution seems very promising, because using a Peer-to-Peer network we could distribute the load among all the peers by distributing the index.

In fact there are already some related approaches in that direction (e.g. the file-sharing networks, Gnutella[6], Morpheus[16],Freenet[4] and so on), but for data-sharing P2P systems there are still many open questions about how to build such a system "right". Existing systems have a lot of weaknesses and there are still problems without a perfect solution. An overview of the most relevant problems for data-sharing P2P systems is given in the paper 'Open Problems in Data-Sharing Peer-to-Peer Systems'([2]).

GALANX now focuses on the core of how to route a query fast and efficient to the appropriate data provider. It does not worry about other issues like security and stability at all.

We will first look at the Key Idea of data-sharing P2P networks. Then we will have a short look at the weaknesses of existing routing techniques and will see a first routing solution using distributed hash tables. Improving that first solution will lead us towards the GALANX system. To achieve a fast query routing, the authors introduce special peer indices to locate the corresponding peers for a query by looking up the needed information in this index. If the network is well build, a peer is mostly able to look up all the information it needs to answer the current query in its own index. If not it has to ask one (or several) other peer(s). So we consider these GALANX peer indices in Detail and will see how they can be build in two different ways.

Finally we will see how fast GALANX can work in practise with some experiments and through that come to some conclusions.
Of course, at the end you can find some references to related work. You can find there especially references to everything marked by a number in brackets like that [xx].

2 Key Idea and State of the Art in P2P Routing

We want to use a P2P system efficiently in a ”Web” like environment, where we have several information providers with mostly specialized content offering a lot of content-related data to the public, but perhaps not be willing to give the complete content source files away. And furthermore this major information providers are stable in contrast to the information searching users.

2.1 The Key Idea

So we imply that **each such data provider is a node in our P2P network**. And **every data providing node publishes an index containing all information needed for resolving search queries about the data it shares**.

That way the information is pushed to the network and has not to be pulled out of the data files. We don’t have to crawl through all the pages a server offers, but this information is given in a compact, complete and up-to-date way by the data holding node itself.

2.2 Wanted System Properties

Developing a P2P search engine system, we want it to fulfill several properties:

1. **The system should be adaptable to complicated search queries**. It should not only be able to handle simple ID-lookups, but also more complicated kinds of search queries. In ”the Web”, the most common search, is full text search, so that should work, of course. It’s also imaginable to have some kind of property search, for example searching movie trailers by their director’s name and the name of some appearing actors. All that should be possible to implement in our network. It would be also nice if we could introduce more complex forms of information retrieval techniques. The GALANX design already is able to handle full text keyword search queries in the prototype implementation. And even more complicated searches should be easy to implement by changing the local Data Indices, the Peer Indices and the kind incoming queries are handled. P2P systems are by design in general more easy to adapt than centralized systems and GALANX is not very restrictive so much is imaginable, but has to be thought through.

2. As already said, data servers can be willing to have the full control over their source data. So **the network should be ownership respecting**, which means there is no source data moved or copied for caching or load distributing. Only the data owning server decides what data to give out to a user query. Imagine the website of a library, where you can ask if you can get a certain book. You just will get ’Yes’ or ’No’ as answer and perhaps some date when you can get it, but you will not get the information who has the book at that moment. If in that case, the nodes would copy the whole database of the library server, every node could look up such information. As a matter of securing the privacy of its customers the library node does not want that, of course.

3. At last **the peers in the network should have as much freedom as possible** to decide how much of their bandwidth and other resources they want to give the network. The system should especially not force a peer with a small bandwidth to do a lot of additional work to speed up the network, if it does not want that. But of course, every node has to answer incoming queries and return its corresponding data.

2.3 Simplifying Assumptions

For the following, we make some assumptions to keep things simple. In the next parts we will think of a simplified environment in which we want develop our efficient search engine system. But starting with a system working in that simplified environment, you could easily imagine to adapt the system to more general, more complicated environments without much trouble and without the need to make (big) changes in the system's design. We assume the following:

1. **All data objects in the network, which can be offered by data providers are simple text files**. We do not care about sound files, pictures, movies and so on. We have just text files from which we can retrieve the contained keywords simply by reading them out (and perhaps applying a stemmer and use a stopword list to make it more efficient).

2. Because we have just simple text files, we can restrict our **queries to be just simple keyword lists. If we find a file containing all the keywords in the current query, we have a hit** and send that file as a matching file back to the requesting user. We do that of course with all corresponding files, so we have no ranking or 'top k' search.

3. As in the real world's "Web", the main data servers are stable, so we also assume that **the nodes in our network, the data providers, are stable**. They don't come and go again. If they are in the network, they stay there. The user clients searching for data can come and go, we don't worry about them. (compare JXTA classification [14]).

With that assumptions a Local Data Index on an arbitrary node would look like this:

Keyword_1	$\text{Doc}_{1,1}, \text{Doc}_{1,2}, \ldots, \text{Doc}_{1,k1}$
Keyword_2	$\text{Doc}_{2,1}, \text{Doc}_{2,2}, \ldots, \text{Doc}_{2,k2}$
...	...
Keyword_n	$\text{Doc}_{n,1}, \text{Doc}_{n,2}, \ldots, \text{Doc}_{n,kn}$

Figure 2.1: The Local Data Index on an arbitrary node

We have for each keyword, a list with all Documents on that node, which contain this keyword. So whenever a query with several keywords is routed to that node, it can answer it locally for its own content by having a look in its Local Data Index. It just intersects the Document lists for all the keywords and that way finds out in which documents all the searched keywords appear. The node then can send these Documents to the user and so answer the query (with the data files it holds itself).

2.4 State of the Art in P2P routing

As already said, there exist several P2P routing approaches, which could be applied to data sharing P2P systems. In fact they are already implemented in the most common file-sharing networks. Let's have a look at their weaknesses.

- At first we have centralized approaches, where a central server is maintaining the complete data file index for the complete network. That kind of system was used for Napster[17]. In such a system lookups are fast, but because of its partly centralized design, it is vulnerable to attacks, expensive, not scalable and in fact no real P2P system. We want a real P2P system which is scalable and does not rely on one (or a few) central server(s).

- The other extreme is the flooding technique used for example by Gnutella[6]. Every node contacts for all queries the whole neighbourhood and so on. That is very unefficient, slow and bandwidth wasting. Additionally it is not reliable, because the queries have a time to life and can easily expire before having found all the data that is in the network. Of course there exist also several hybrid solutions combining this technique with the previous technique, but they have also their weaknesses.

- At last we have distributed hash tables(DHT). One example for such a technique is CHORD ([24]). We know that DHT systems are reliable, theoretically scalable and distributing the load among the nodes. They have also more or less fast lookup routines depending on the necessary relay messages to find the correct node in the network. But applying a DHT mechanism implies redistributing the data on the servers according to the DHT hash mechanism. That conflicts with the ownership respecting we want from our system. On the other hand it would be very hard to find a hash function which fits exactly the data distribution we have on our nodes, and which also fits if updates appear. So we cannot work directly with a DHT mechanism, but perhaps that is a good starting point.

3 A first DHT solution

The key idea for building a data sharing P2P system with a DHT mechanism, is the following. Just put a DHT routing layer above the real data network. So the DHT mechanism does not distribute the data itself, but the responsibility to know where the data lies. That means nodes have to know which other nodes are holding data which contains the keyword searched for. Thus, every node is responsible to know for some keywords all the nodes holding data with that keywords.
For that we need **Peer Indices, which look like the Local Data Indices. Instead of Document Lists for each keyword they store the list of all Peers having data files which contain the keyword**. So they look like that:

$\mathbf{Keyword_1}$	$\mathbf{Peer_{1,1}, Peer_{1,2}, \ldots, Peer_{1,k1}}$
$\mathbf{Keyword_2}$	$\mathbf{Peer_{2,1}, Peer_{2,2}, \ldots, Peer_{2,k2}}$
...	...
$\mathbf{Keyword_n}$	$\mathbf{Peer_{n,1}, Peer_{n,2}, \ldots, Peer_{n,kn}}$

Figure 3.1: A Peer Index

A Peer Index covering all keywords in the network, is called a Global Peer Index. With such a Global Peer Index we could obviously route every query directly to the correct nodes, because we have all the information we need directly accessible in that table. Assume that we have such **a Global Peer Index on every node**. **In terms of routing that would be perfect**, because we could route every query directly to the corresponding nodes. Obviously the update costs in such a system would be horrible, because every change had to be sent through the entire network, so every node can update its peer index.

To decrease the update cost we distribute the Global Peer Index over the whole network by applying the DHT hash function on the keywords. So every node maintains just a part of it. Whenever a query comes in, the peer processing this query can apply the hash function on the keywords and that way find out which nodes are responsible for knowing where data for that keywords can be found in the network. So it asks those peers for the corresponding peer lists and like working with the Local Data Index, it intersects the incoming Peer lists and finds out which node have data for all the keywords and so can have matching files.

In the example on the next page we have a network where every node is responsible for maintaining the Peer Index for one keyword. Now a user asks an arbitrary node, in that case P3, the Query ”Math AND Matrix”. So he searches files containing the words ’Math’ and ’Matrix’. P3 applies the hash function on both words. That gives it the nodes responsible for that words. It asks those nodes for the corresponding peer lists. When it gets them, P3 intersects the two lists and finds out that just P4 can have data files containing both keywords, because it is the only node having data for the keyword ’Math’ as well as for the keyword ’Movie’. So it routes the query to P4 and P4 answers the query by sending all its matching files back to the user.

We can easily understand that this works in a stable environment and has a lot of the properties we defined earlier. For example it is scalable, reliable because DHT mechanisms are. On the other hand it is very restrictive to the nodes, because the DHT mechanism determines deterministic which nodes are responsible for which keyword. So it could happen, that a node with a high bandwidth is responsible for a keyword which just appears once a year in a query, while another node with low bandwidth is responsible for the most popular words. Both cases should be avoided. Furthermore, through this extra lookup at the keyword responsible peer, we have a performance loss of one extra hop, which could perhaps be avoidable. That performance loss gets even worse through the deterministic distribution of the keywords, because it is very likely that content related keywords are on different nodes. That means for n keywords in a query we probably have to ask n peers for their peer lists.

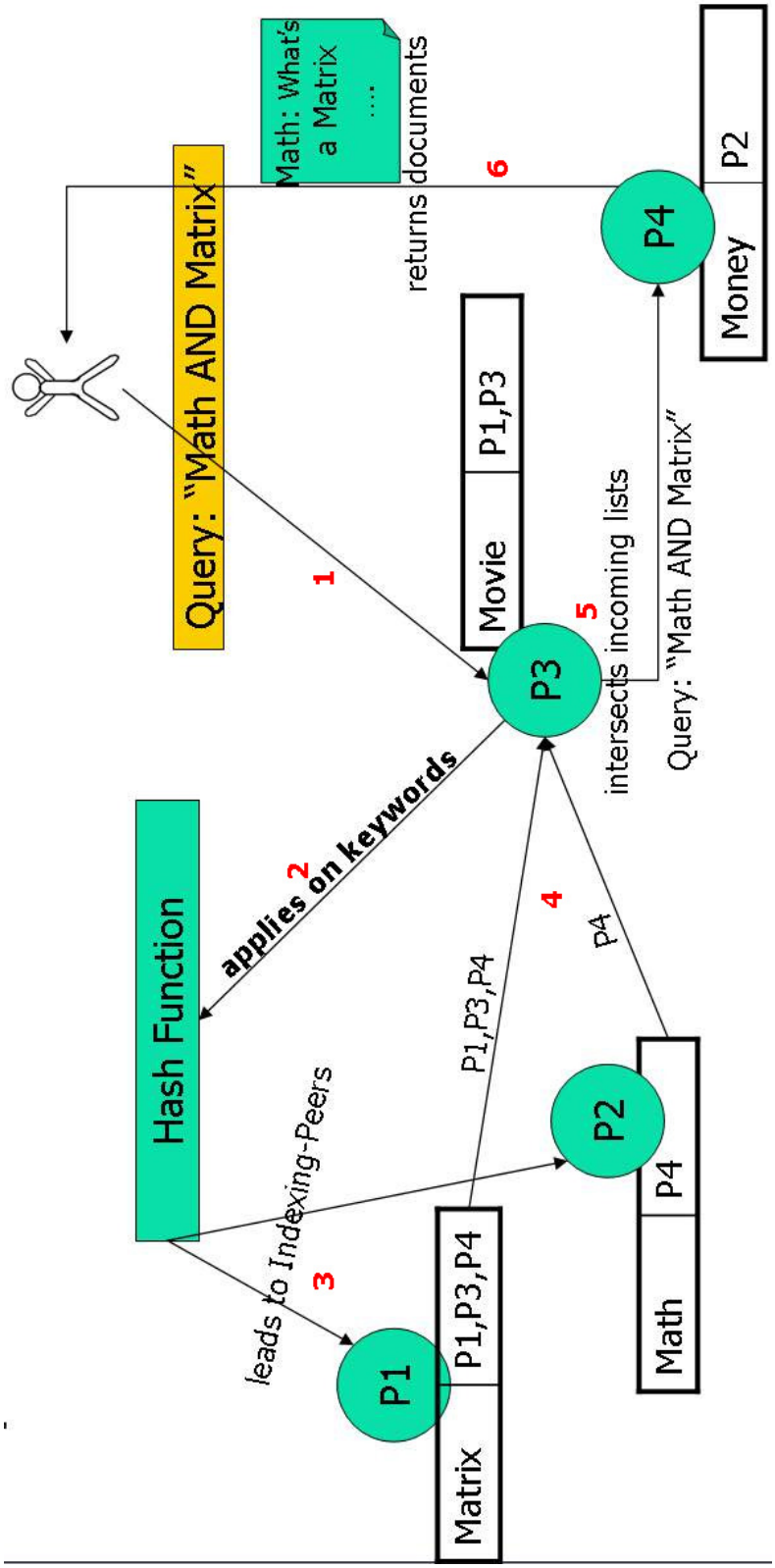

Figure 3.2: A DHT routing example

4 The GALANX design

As we saw the perfect solution for fast routing would have too high update cost, the DHT solution on the other hand is very restrictive and loses a lot performance through the extra lookup hops. So something in between could be a good solution. For realizing that the GALANX authors introduce special Peer Indices.

4.1 GALANX Peer Indices

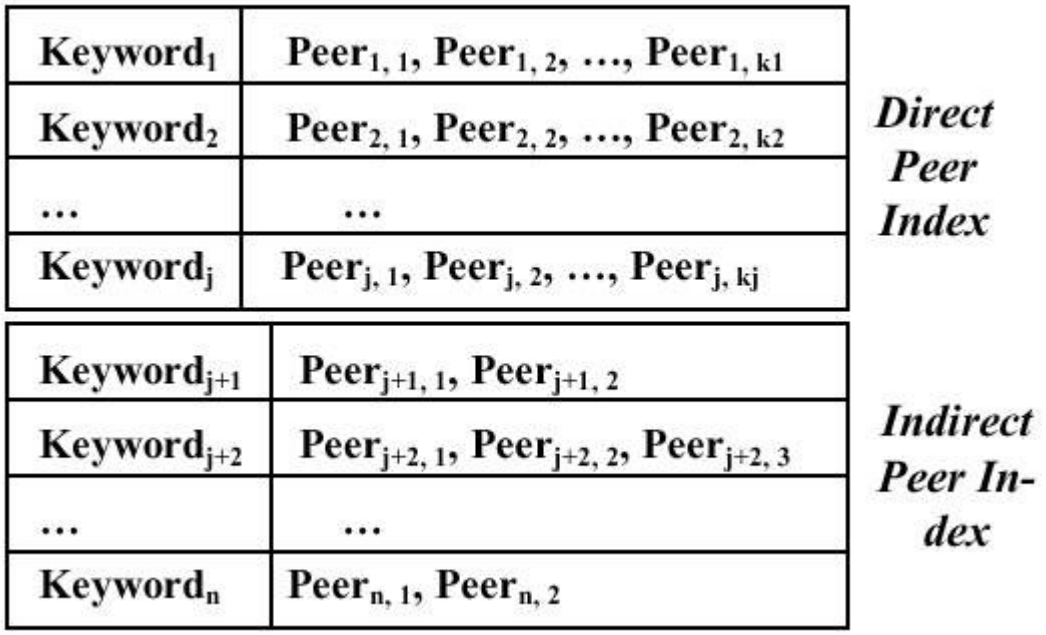

Figure 4.1: A GALANX Peer Index

This Peer Indices are divided into two parts. **The Direct Peer Index is a Peer Index as we know it this far**. So for several keywords, we have for each keyword the list of all nodes having data in which this keyword appears. The Indirect Peer Index covers the remaining keywords so that both parts together cover all keywords in the network. **In the Indirect Peer Index nodes store for each keyword some peers which have that keyword in their Direct Peer Index**. So for each keyword a node has either all information it needs, having that keyword in the Direct Peer Index or it knows a node which has all the needed information about the appearance of that keyword.

In the example (next page) we have again a user asking for data with the keywords 'Math' and 'Matrix'(1.step). P3, the query processing node, then looks up both words in its peer index(2). 'Math' is in its Direct Peer Index, so it has the list of all corresponding peers directly. Because 'Movie' is in the Indirect Peer Index, to get the peer list for 'Movie' P3 has to ask P2, which is its the indexing node for 'Movie'(3). P3 looks up the keyword and sends the peer list back to P3(4). P3 then can intersect both lists and route the query to P1 which is the only peer appearing in both lists(5). P1 then answers the query to the user(6).

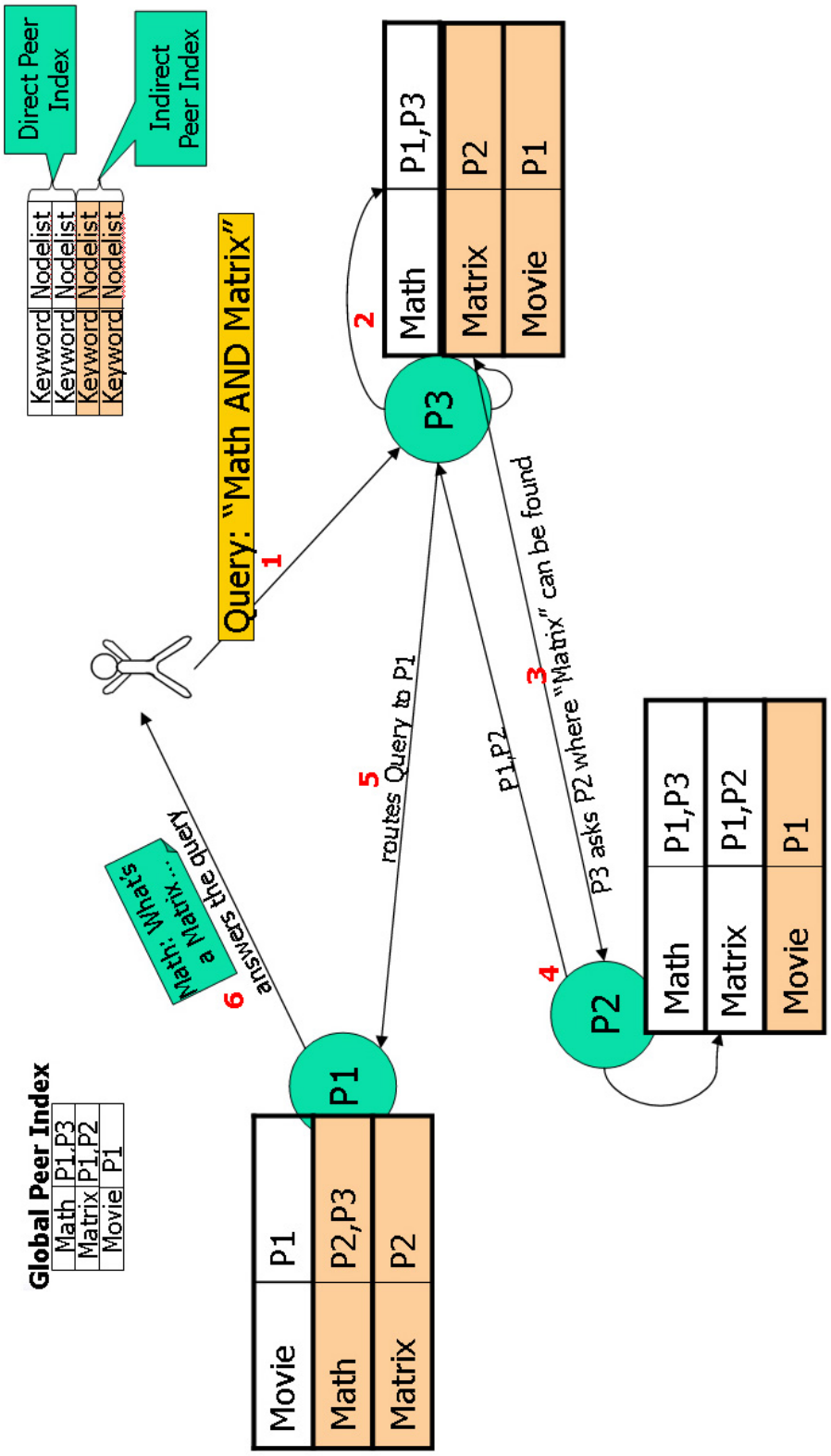

Figure 4.2: A routing example with GALANX Peer Indices

As you can see, **if a node has all the keywords of a query in the Direct Peer Index that is just perfect**, because it can route the query directly to the corresponding nodes. If that is not the case, it can probably look up the remaining keywords on one other node or at least it is very likely that it has to ask less nodes than with the DHT approach we saw. If for example P1 would be asked the same query('Math' AND 'Matrix') it just had to ask P2 for both peer lists. Obviously it would be good if the most popular keywords are in the Direct Peer Index of a node. Thus, **if a nodes strategy is good, most popular words can be found in Direct Peer Index**. But on how to ensure that a node acts wisely to achieve this, the authors do not say anything concrete.

4.2 Building GALANX Peer Indices

We saw that in a (stable) network routing with GALANX Peer Indices can work and we believe it can route fast. But how to construct these Peer Indices?
There are two different building strategies: **The Direct Index Construction**, which works incrementally without further changes to the design and the **DHT Index Construction**, which uses a DHT mechanism to give the network some concrete base on which the system then evolves by training.

4.2.1 Direct Index Construction

We start with a small network where every node maintains the complete Global Peer Index. You could think of starting with just one node where it is obvious, that it can maintain the Global Peer Index or having at first an update expensive P2P system which ensures that every node maintains the Global Peer Index by sending update information to every node. Then when the system 'thinks' bandwidth gets low it just switches over to the GALANX Peer Index Construction.
Well, the point is we can imaging a small starting network in a situation where every node has the Global Peer Index, which means in terms of GALANX Peer Indices that on every node all keywords are in the Direct Peer Index.

If now a node joins, it copies an arbitrary node's peer index and updates that with its own data.
In a growing network every node can decide to drop keywords from the Direct Peer Index to the Indirect Peer Index. To do that the keyword dropping peer has to do that:

- Randomly picking a few 'keyword-data' holding peers, which are in the Direct Peer Index entry of the keyword, that should be dropped
- Put that peers together with the keyword in the Indirect Peer Index as indexing peers for that keyword
- Notify those nodes that they are now indexing nodes for that keyword, so they can send a message whenever they drop the keyword. In that case the peer can update its Indirect Peer Index and put in another node.

When exactly nodes should drop keywords from the Direct to the Indirect Peer Index is a free decision of the nodes. The authors of the paper do not make any suggestion what would be a good choice, except for a small indirect hint in the experiments part.

Whenever a keyword update happens (e.g. new data inserted, joining nodes), there can be two cases:

- **The keyword is already in the network**, then the peer updating it has it already in its peer index. In that case **the update is just sent to the indexing nodes**, precisely the nodes which are in the Direct Peer Index for that keyword. (If the keyword updating peer has the keyword just in the Indirect Peer Index it can get the full current peer list of course by asking its indexing node for that keyword).
- If the keyword is new to the entire system, the update information has to be sent to all nodes in the network, so they can decide whether to put them in their Direct or Indirect Peer Index. This choice is again free for the nodes and there is again no concrete declaration by the authors what's a good choice. The peers have even the choice to put a new keyword not in their Peer Index, but having instead a wildcard entry, with a node they can ask if ever an unknown keyword appears. That could bring some danger for reliability, because you have to ensure there is another node which maintains a complete Peer Index and seems to be just an idea not further observed.

The freedom of the nodes using this pure design also brings other problems with it. Let's have a look at the Direct Construction Example (page 15). We have a starting network with P1 and P2, both having the complete Global Peer Index and a joining node P3 which has data with the keywords 'Math' and 'Matrix'. It asks for an peer Index and updates that with its own data (page 16). Because 'Matrix' is already in the system, the update information is just sent to its indexing nodes, which is in that case only P2. 'Math' on the other hand is completely new to the entire network, so the update is sent to all nodes and they decide to put it in their Direct or Indirect Peer Index. P1 decided to put it in the Direct Peer Index, while P2 prefers the Indirect Peer Index. Having now, starting there, a new joining node P4 with new data for the keywords 'Math' and 'Movie' (page 17). It first gets a peer index and updates that with its data. Because it wants to put 'Movie' to the Direct Peer Index it gets the peer list from P1. The choice to let the entry for 'Movie' in the Indirect Peer Index or moving it to the Direct Peer Index is again by the pure design free to be decided by the node. P4 then has a fully updated Peer Index and sends the update information according to that (page 18). So P3 gets informed about the new 'Math' entry and P1 about the new 'Movie' entry. Obviously P1 now lacks some information for 'Math', it does not know there is also data on P4 for that word. That is because P1 has no data with that word itself, so it is not informed about updates, because it does not appear at any Direct Peer Index for that keyword. To overcome this you could either change the routing a bit or, as the authors did it in the experiments, restrict the nodes to have only keywords in their Direct Peer Index for which they also have local data files.

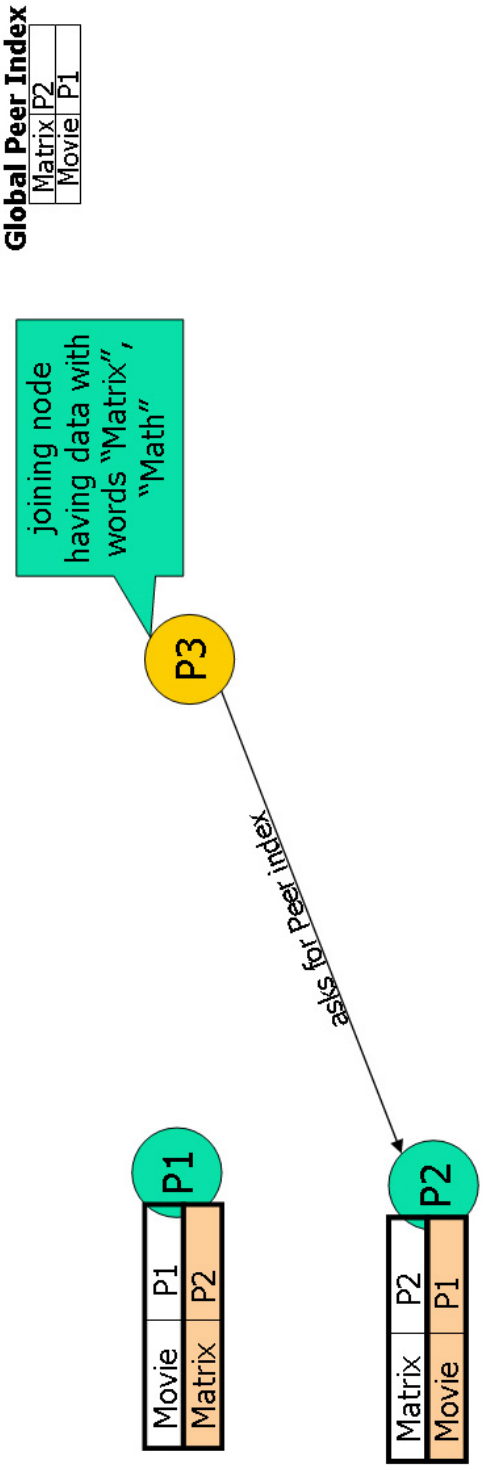

Figure 4.3: Direct Construction

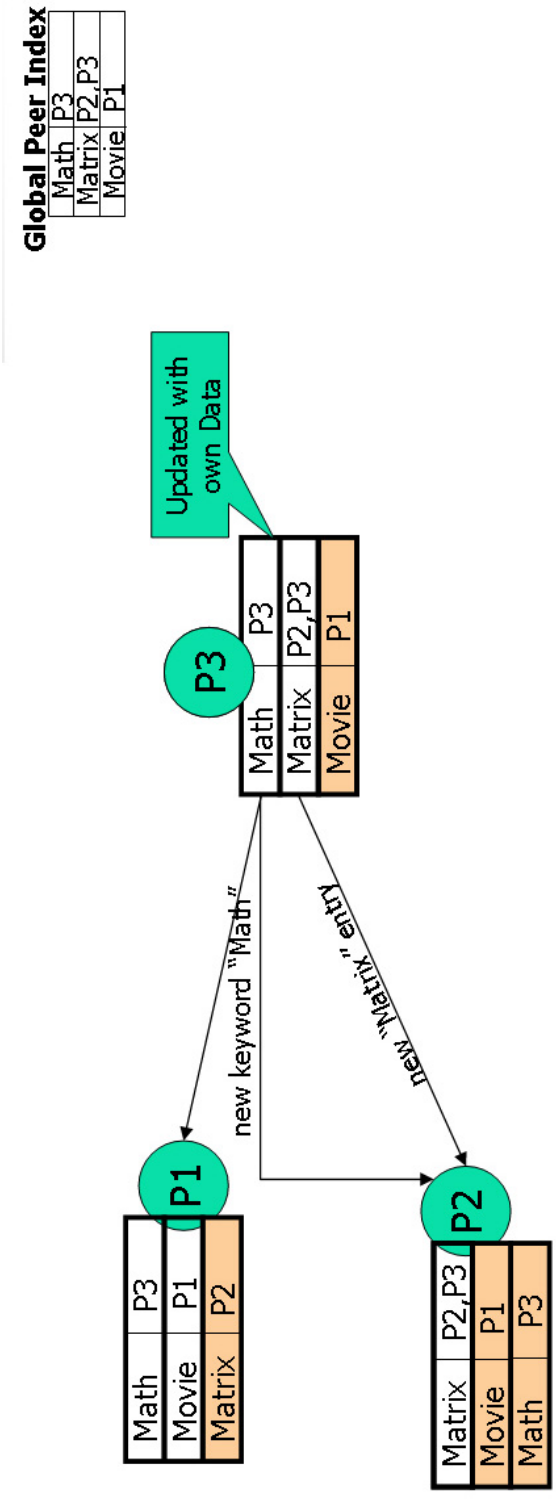

Figure 4.4: Direct Construction

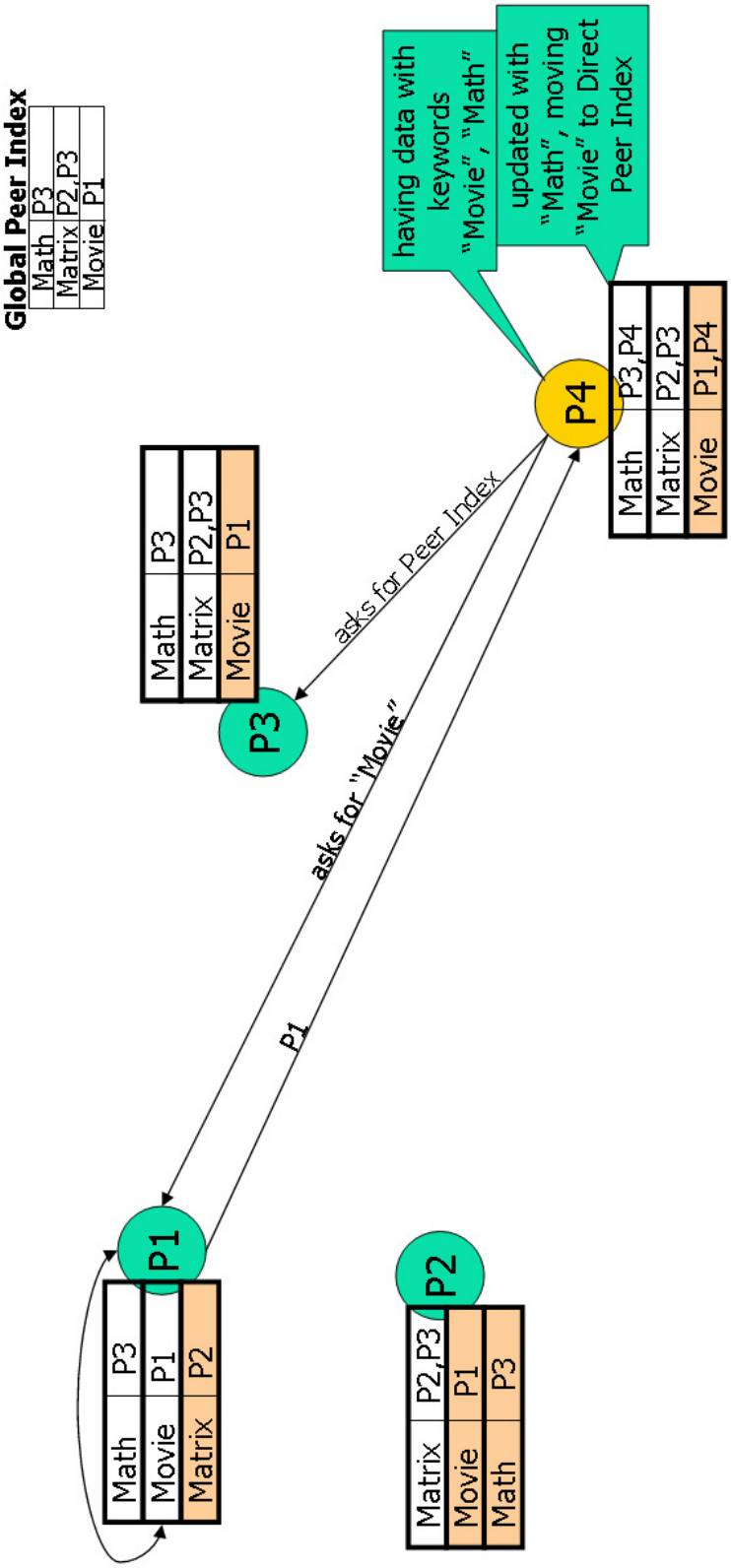

Figure 4.5: Direct Construction

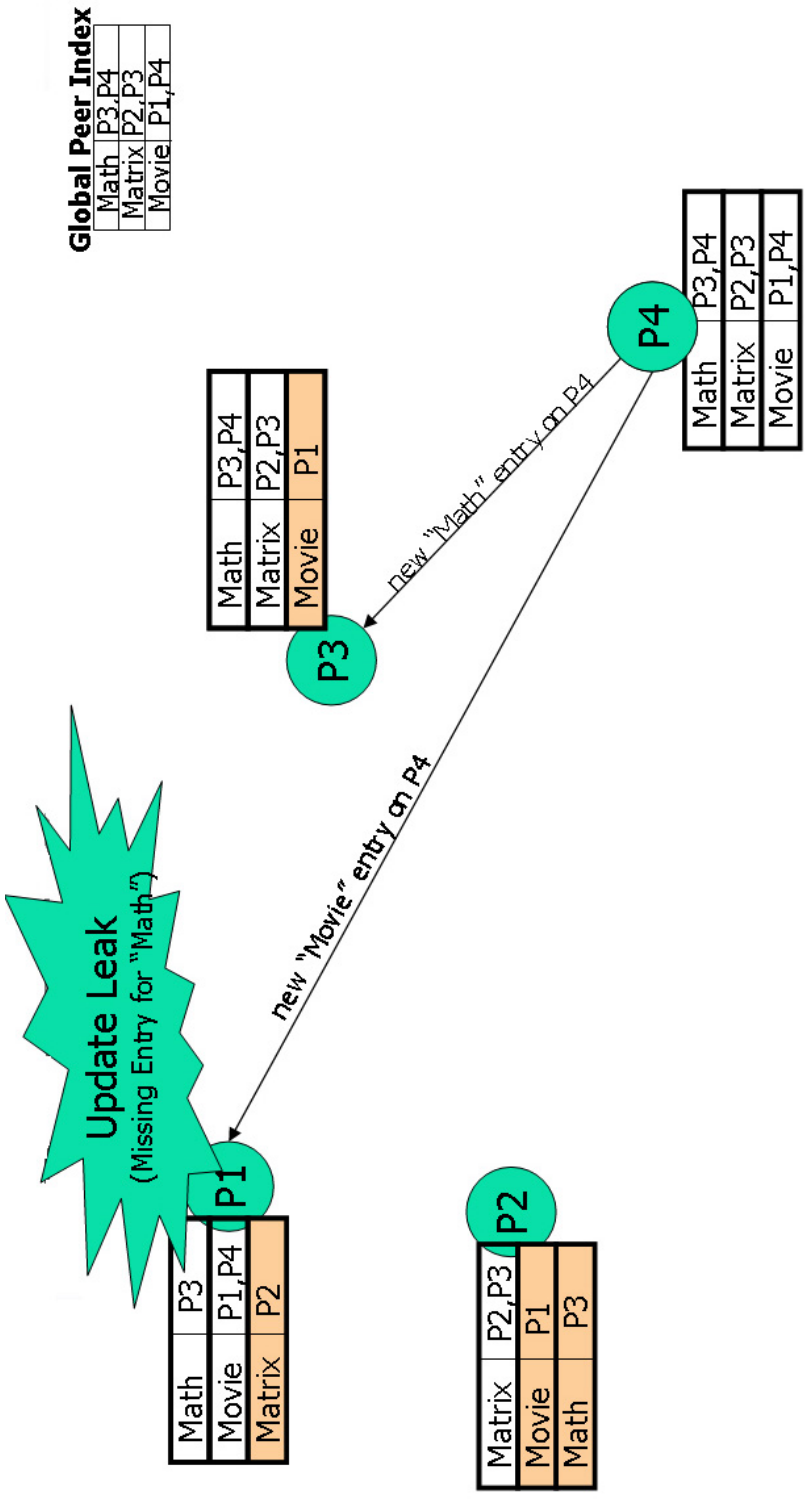

Figure 4.6: Direct Construction

4.2.2 DHT Index Construction

In that approach we just use the DHT solution we saw first as a base to build the system on. So we have a DHT mechanism which distributes the responsibility of knowing where data for a keyword can be found, exactly as in the first DHT solution. So every node is responsible for a certain keyword depending keyspace. Every time a node joins that responsibility is redistributed and the corresponding Peer Index entries are moved if necessary. So the DHT mechanism does the main work. Keyword Updates work nearly the same way as seen with Direct Construction. We just use now the DHT Hash function to find the indexing node for a certain keyword. The important point and the difference to the first DHT solution is, that the nodes, especially their Peer Indices get trained by queries (next page). So whenever a query comes in(1), if the keywords are not in the node's Direct Peer Index, it applies the DHT hash function(2) to find the nodes responsible for indexing that keyword(3). That peer(P1) sends back the corresponding peer lists(4). Having them the node(P2) can decide to which node(s) to route the query(P3) and does so(5). The other node then can answer the query(6). At the point it routes the query, the query processing node(P2) also puts the keyword with the indexing peer in its Indirect Peer Index. If a keyword which has an entry in the Indirect Peer Index appears often in user queries the node then can decide to put it in its Direct Peer Index. But there is again no suggestion by the authors when exactly that should happen to achieve the best results.

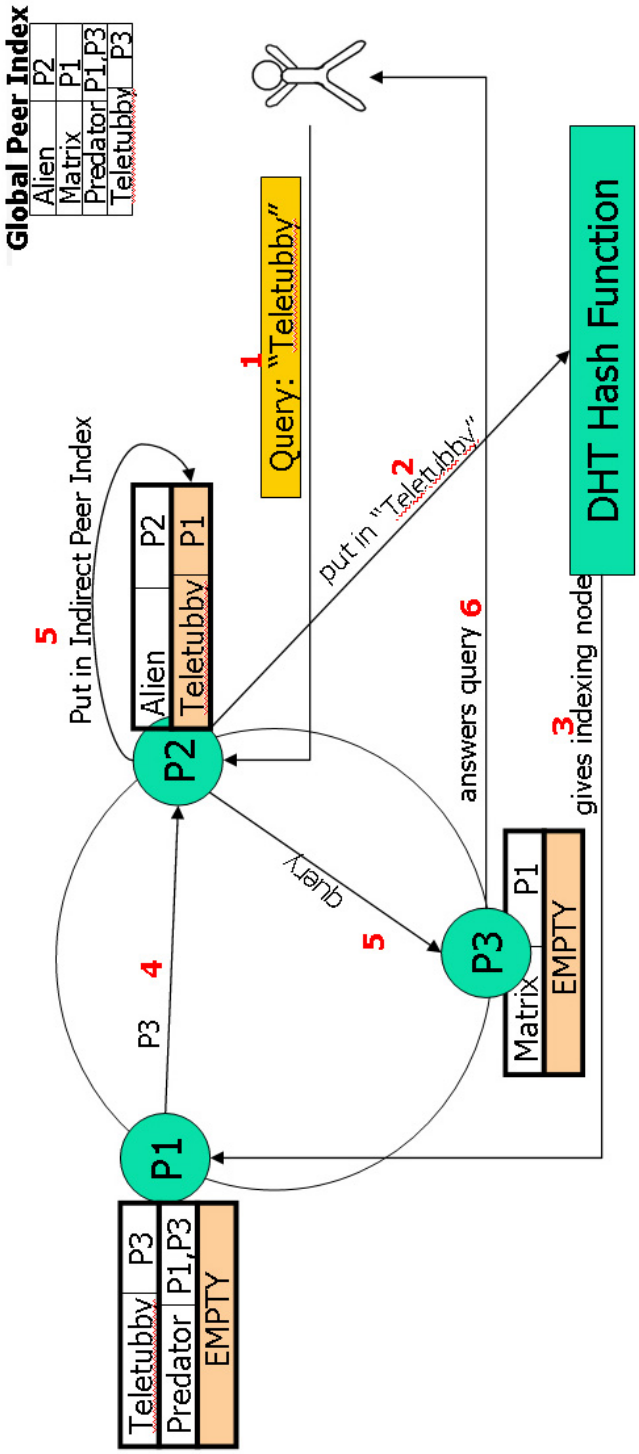

Figure 4.7: DHT Construction - Training

5 Implementation and Tests

Having seen all the theory, it is time to see some tests in practise.

5.1 A Prototype Implementation

As already said the authors of the paper have implemented a prototype GALANX system as follows: The main parts are the Apache HTTP Server and the Berkeley DB Data Store. While the Data Store

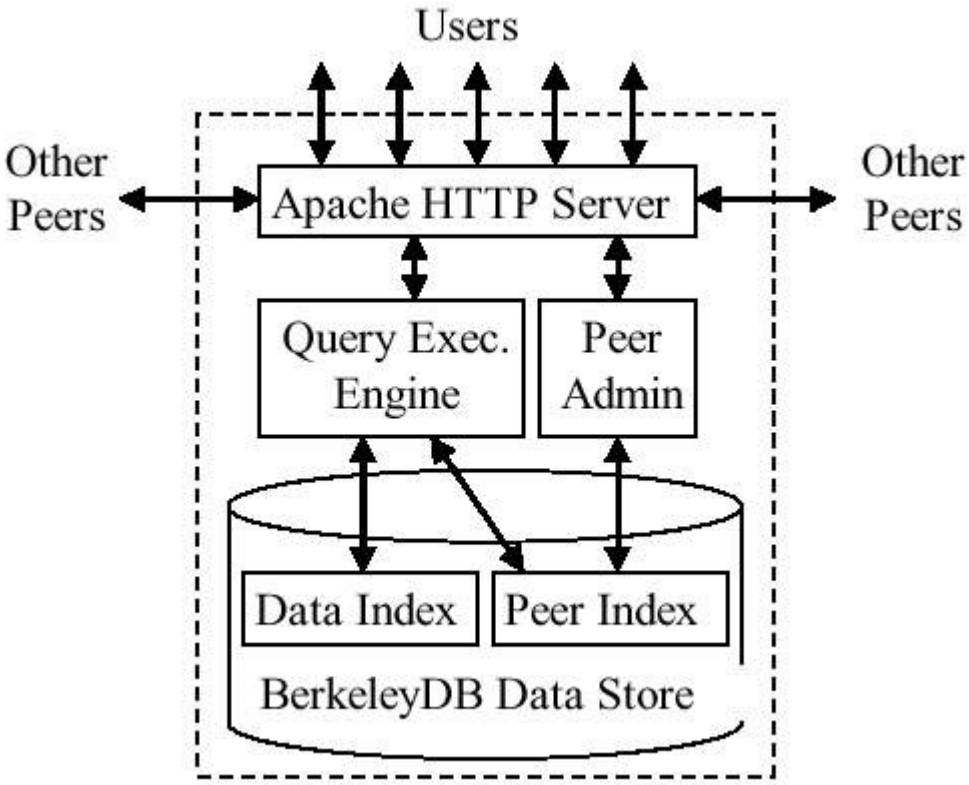

Figure 5.1: The Prototype Implementation Architecture

stores of course the Data Index and Peer Index, the Apache Server is responsible for all communication. The other parts, the Query execution engine, the Peer Admin and the User Interface are written in C and embedded as modules in the Apache Server. As the names already say, the Query Execution engine handles all query related tasks, while the Peer Admin handles the connections with other peers. The point is, that the Apache Server and the Data Store does not work well together in multi-process mode, so queries have to be queued more often than that would be the case with a better implementation. Therefore this implementation is slower than it could be.

5.2 Experimental Setup

For testing issues the authors set up a special environment. First some technical facts:

- 100-node computer LAN-cluster, 933/550MHz
- 1GB memory 20million+ HTML-pages distributed by category to nodes
- 2.49 million keywords, average 125.000 keywords per node
- Simulated user queries with dynamic expiration time Initial expiration time 10 sec, min 5 sec, max 30 sec
- Only queries which can be answered

The expiration time of user queries is dynamically changed as follows: Whenever a query is answered in time, the expiration time of the next query is reduced by 50 percent to a minimum of 5 seconds and the other way round increased by 50 percent to a maximum of 30 seconds when no answer is received in time.
As you can see at the results of the experiments (5.2, 5.3) we have also several other P2P implementations to compare GALANX with:

- Gnutella: Using flooding, that should be the slowest implementation
- Chord: The size of the logical node ring [24] is set to be 232, so every node maintains a finger table that has 32 entries. When a peer looks up the peer index of a keyword, it follows its finger table to reach the corresponding indexing node through a chain of messages.
- Chord+: In order to focus on the number of individual index lookup requests sent for each query, every node maintains a complete peer map. Thus, a peer can send a lookup request directly to the corresponding peer after applying the hash function on a keyword. Thus, no relay messages on the route to the indexing node as in the original Chord approach are required
- Complete: Here every node has a copy of the complete Global Peer Index. That should be the fastest way, if no changes happen.

And we have also two different GALANX system using the two different building strategies as follows:

- GALANX-1: With this approach the Direct Peer Index for each node contains all the keywords that are found in the nodes local data files while the rest of keywords are stored in the Indirect Peer Index
- GALANX-2: Starting from the peer indices built using Chord, every node continuously constructs the indices by running a 5000-query set. The queries in each query set are randomly selected from the global query pool. 50the other 50other peers.

5.3 Results

At most pictures on the next pages, in the x-direction you can see the number of simulated users per node. With a higher amount of users, the amount of queries increases of course, too. Because the main aim of the search engine system is to be fast in routing, the most important pictures are 5.1,5.4,5.5 and 5.6. They all show nearly the same result, namely that GALANX-1 and 2 are both very near to the 'perfect' Complete solution and a good thing faster than the DHT approaches and of course than Gnutella. We can see that for example on picture 5.1, where we see the Query Response time in average. So that is the time the system needs to deliver the first result after the query is asked a node. As you can see GALANX-1 and 2 are faster than the DHT approaches and nearly as fast as the Complete solution. It is nearly the same with pictures 5.4,5.5 and 5.6. Just to make clear what you see, some explanations about the terms:

- 5.4 Query Execution Ratio: The Perentage of incoming Queries, which are then executed (not expired yet)
- 5.5 Query Response Ratio: The percentage of queries that produce results.
- 5.6 Query Recall: the ratio of data files received as answer and the data files which are in the network matching that query

On 5.6 the authors refer all lost, not delivered data files in answers, although they are in the net, to be lost by not executing expired queries.
Picture 5.2 shows us how much queries are received by nodes in average. It's clear that the Complete peers as being faster, get more than GALANX, also by the flooding technique Gnutella sends more queries around by itself.
Picture 5.3 shows us only something obvious, namely that the GALANX system needs less index lookups than the DHT approach, which is clear because with GALANX more queries can be routed directly. It is just compared to Chord+, because there are no relay messages needed by the DHT mechanism, so that is a just comparison.
At last we have two pictures which deal with updates. At a constant number of 5 users at 5.7 there is new data inserted every 5 minutes. At 5.8 there are some new nodes inserted every 10 minutes. As we can see GALANX stays in both cases faster than the DHT approaches, but the difference becomes a lot smaller. Unfortunately we have no experiments that show us at which point GALANX gets actually slower than the DHT-approaches.

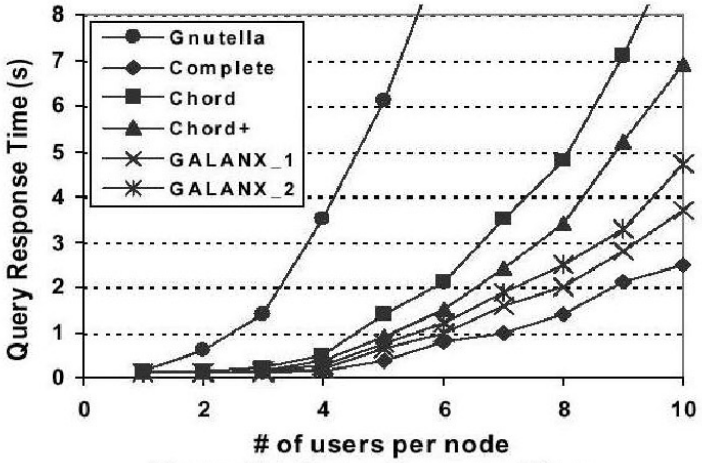

Figure 5.1 Query Response Time

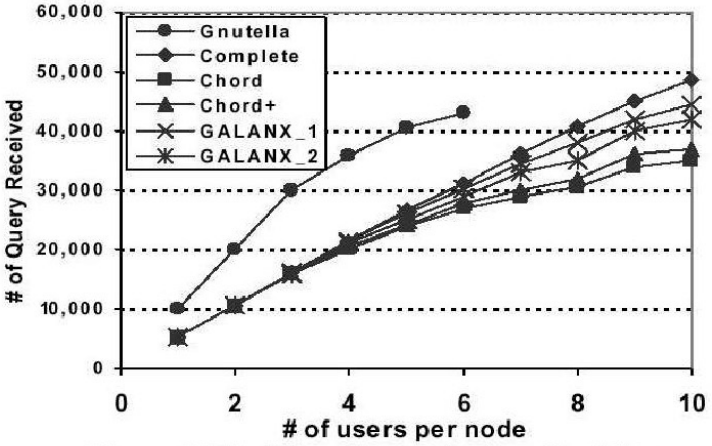

Figure 5.2 # of Queries received on Each Peer

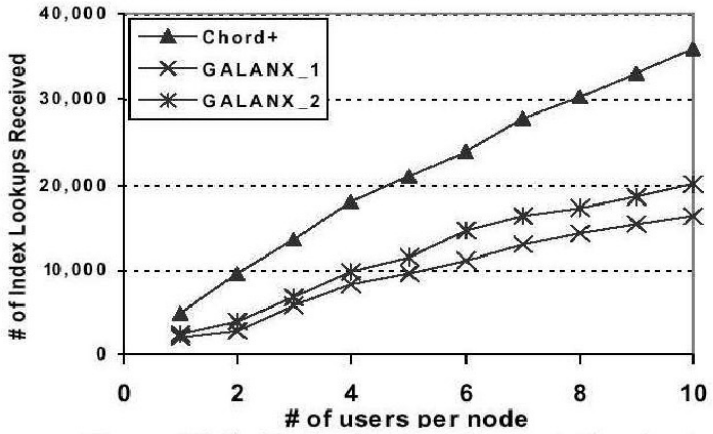

Figure 5.3 # of Index Lookup Requests Received

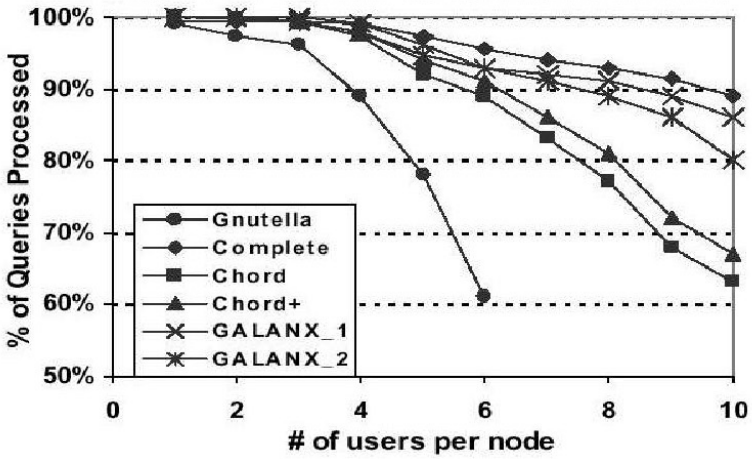

Figure 5.4 Query Execution Ratio

Figure 5.2: experiment results

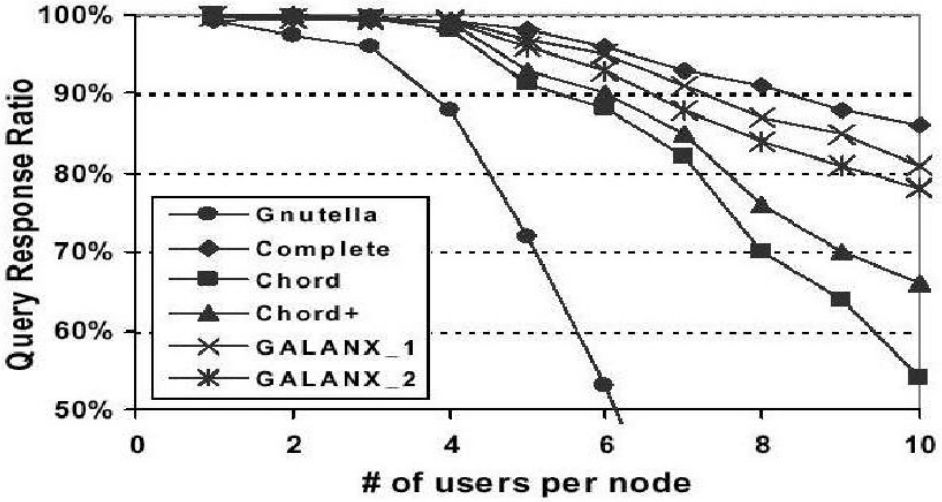

Figure 5.5 Query Response Ratio

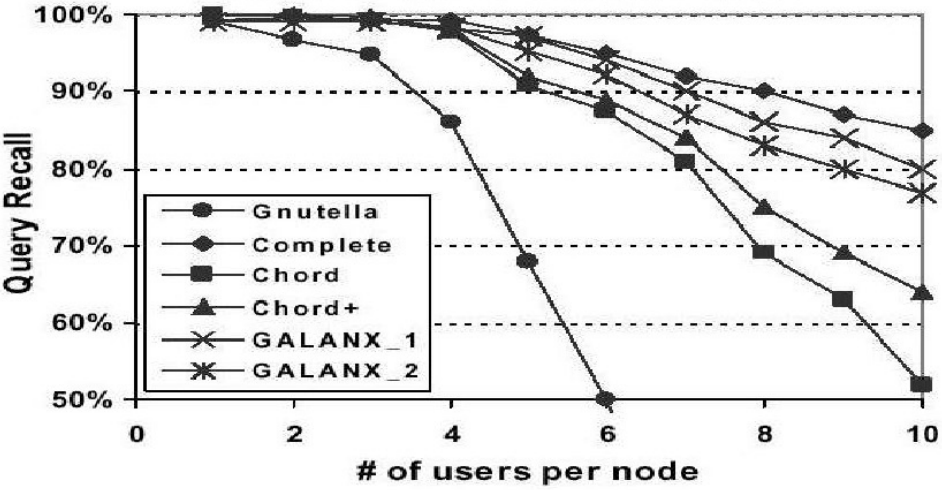

Figure 5.6 Query Recall

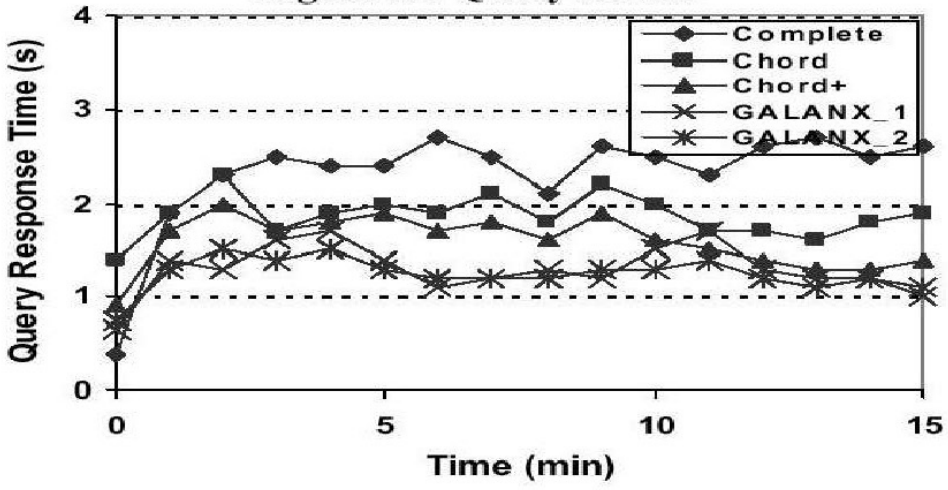

Figure 5.7 Query Response Time (with new data)

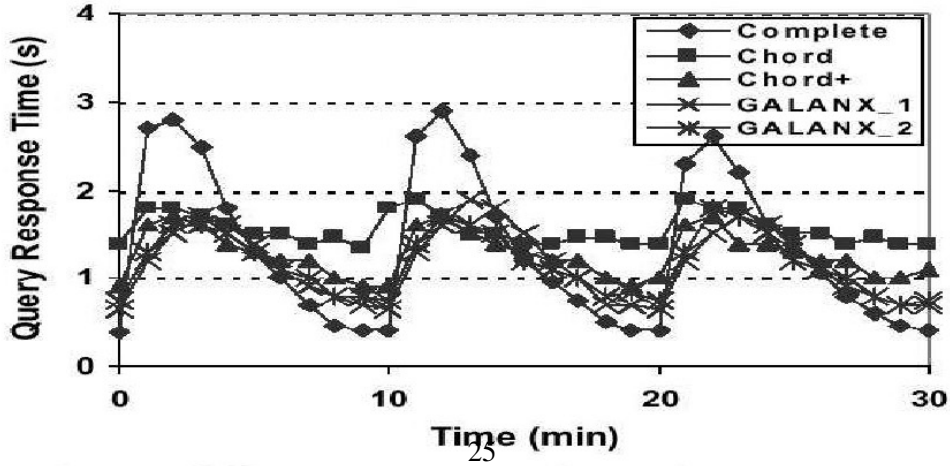

Figure 5.8 Query Response Time (with new nodes)

6 Conclusions

We have seen that GALANX can work, and it can even route very fast. It can deliver nodes much freedom, so they can decide how much they want to offer the network. But as with the pure design nodes have much freedom we do not know what are good decisions. We do not know when to put keywords in Direct or Indirect Peer Index to achieve the best results. And if the nodes make the ”wrong” decisions this freedom can in some cases be even dangerous for reliability as we saw. And we have no experiments or proofs that show us how good GALANX is in dynamic environments. So to get a really applicable solution lot has to be added and we need concrete solutions about how peers should make their decisions. Through the restriction made in the experiments, namely that peers have only keywords in their Direct Peer Index for which they have also local data is a hint, but we do not know how good this is compared to other possibilities.

The authors next aim is to extend GALANX to more complex forms of queries and to introduce information Retrieval Techniques like relevance ranking.

7 References

Some References to related work, partly referenced also in the text above. More to be found in the original paper.

[1] A.Crespo, H. Garcia-Molina. 'Routing Indices For Peer-to-Peer Systems' in Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 02), 2002.

[2] N.Daswani,H.Garcia-Molina, B. Yang. 'Open Problems in Data-Sharing Peer-to-Peer Systems', in Proceedings of the 9th International Conference on Database Theory (ICDT03), 2003.

[4] The Freenet website, http://freenet.sourceforge.net.

[6] The Gnutella website, http://www.gnutella.com.

[9] L.Galanis, Y. Wang, S.R. Jeffery, D.J. DeWitt. 'Processing Queries in a Large Peer-to-Peer System', in Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE 2003), 2003.

[10] L. Galanis, Y. Wang, S.R.Jeffery, D.J DeWitt. 'Locating Data Sources in Large Distributed Systems', in Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'03), 2003.

[11] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, I. Stoica. 'Complex Queries in DHT-based Peer-to-Peer Networks', 1st International Workshopon Peer-to-Peer Systems, 2002.

[14] The JXTA project website, http://www.jxta.org.

[16] The Morpheus website, http://www.musiccity.com.

[17] The Napster website, http://www.napster.com.

[18] C.R.Palmer, J.G. Steffan. 'Generating Network Topologies That Obey Power Laws', IEEE Globecom 2000, 2000.

[21] S. Ratnasamy, S. Shenker, I. Stoica. 'Routing Algorithms for DHTs: Some Open Questions', the First International Workshop on Peer-to-Peer Systems (IPTPS02), 2002.

[24] I.Stoica, R. Morris, D. Karger, M. Kaashoek, H. Balakrishnan. 'Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications', ACM SIGCOMM' 01, 2001.

[27]D. Tsoumakos, N. Roussopoulos. 'A Comparison of Peer-to-Peer Search Methods', in Proceedings of the International Workshop on Web and Databases (WebDB03), 2003.