Extending Peer-to-Peer Networks for Approximate Search

Alain Mowat
and Roman Schmidt
Ecole Polytechnique Fédérale
de Lausanne (EPFL)
1015 Lausanne, Switzerland

Michael Schumacher
University of Applied Sciences
Western Switzerland
3960 Sierre, Switzerland

Ion Constantinescu
Digital Optim
USA

ABSTRACT

This paper proposes a way to enable approximate queries in a peer-to-peer network by using a special encoding function and error correcting codes. The encoding function maintains neighborhood relationships so that two similar inputs will result in two similar outputs. The error correcting code is then used to group the similar encoded values around special codewords. In this manner, similar content is located as close as possible in the network. The algorithm is tested in a simulated environment on a HyperCube network overlay.

1. INTRODUCTION

When searching for information on internet, it often happens that the searcher does not know the correct spelling of an authors name or simply mistypes a word in the search query. In most P2P systems, this will often lead to erroneous or no results being returned by the system. Approximate queries extend the notion of normal queries by allowing entries that only partially match the initial query to be retrieved. We call the *c-neighborhood* of u all possible items in Σ^n that differ from u by at most c bits. The idea is that when searching for u in the network, we extend the search to all of its *c-neighborhood*.

Peer-to-peer (P2P) networks can be classified into three categories depending on how they index and search items in their network. Systems with local or central indices can implement approximate search easily. However, their search function is not as efficient as in distributed hash table (DHT) where. In DHTs, each peer is responsible for indexing a certain range of files. All files and peers are attributed an identifier that represents it in the network. The peers then index the files that have an ID closely related to their own. The problem arises when giving the ID to the different files. Indeed, in most networks, a hashing mechanism is used. The problem is that this destroys any information about the files and thus approximate search is made impossible.

In this paper we propose a new hashing (or encoding) function that preserves locality while trying to maintain an even distribution of hash values[1]. The particularity of our hashing function is that it will not map a query to one single bucket, instead it will return a range of buckets that contain the query and its *c-neighborhood*. Thus the hashing function will be: $h_{AQ} : \Sigma^n \rightarrow \{B_1, ..., B_p\} \times ... \times \{B_1, ..., B_p\}$. In addition to this encoding function, the perfect Golay error correcting code is used to group similar content and queries in the network, thus allowing a certain degree of approximation in the search algorithm.

2. ERROR CORRECTING CODE

The error correcting code (ECC) used for our approach is the *Golay code*. It is a (23, 2^{12}, 7) code, meaning that there are 4096 codewords and the length of a *received vector* is 23 bits. A *received vector* (RV) is a vector or string of bits that is received by the destination during a transmission on a noisy channel. The Golay code is additionally a perfect code, in the sense that each RV is mapped to one and only one codeword. In this way, the code can correct up to 3 erroneous bits. We can imagine this as each codeword being a point in a hyperspace surrounded by a ball of radius 3. The set of all codewords are disjoint and cover the whole hyperspace. A received vector can thus be positioned anywhere and will always be contained in one of the balls (Fig. 2).

In this paper, the ECC is used after the encoding of the query. The encoding creates a 23 bit vector and the Golay code then computes the corresponding codeword. As we will see, the encoding function is neighborhood-sensitive. Similar queries are mapped to similar received vectors. This is when the ECC comes into play. As long as two RVs differ by at most 3 bits, the Golay code will map them to the same codeword. This way they will be indexed at the same location in the network (see Fig. 1).

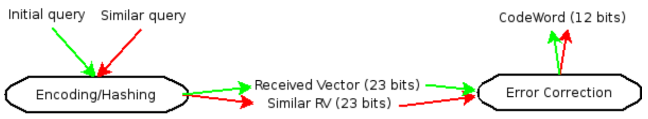

Figure 1: Sequence of modules transforming two similar queries into a single codeword

Alone, this system is not powerful enough to correct queries that contain many errors. Only 3 bits can be corrected, so

[1]The work presented in this paper was partly supported by the Swiss National Research Foundation grant Nr. PI0I2–115015 / 1 and by the Swiss National Funding Agency OFES as part of the European project NEPOMUK No FP6-027705

to enhance the results, the ECC does not only search for the codeword to which the RV is mapped to, but also the codewords that can be reached by all the vectors within the *c-neighborhood* of the original one. What happens for example if $c = 1$, is that an error is artificially introduced into the RV to create a new vector. Then this new vector is mapped to a Golay codeword. This is done for all of the 23 bits of the initial RV. Obviously the higher c is, the better the correction algorithm, since the desired codeword can be found with many errors in the vector. However we will see later on that for practical reasons, the value of c can't really be increased much higher because the number of neighboring codewords rapidly increases with respect to c.

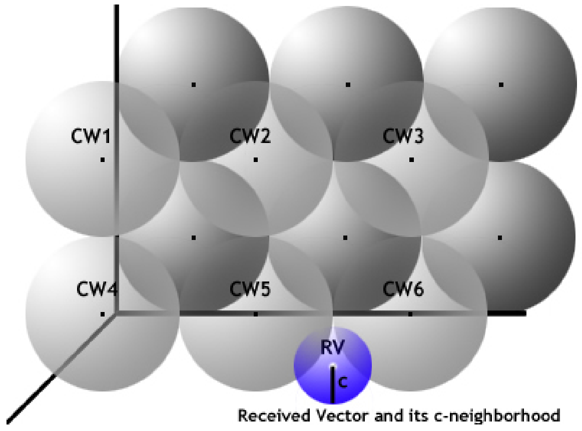

Figure 2: Representation of codewords and a received vector

A problem with the Golay code when used for our purpose is that 4096 codewords remains a relatively small number when compared to the number of documents that can be stored in a P2P network. This fact means that several documents will necessarily be mapped to the same codeword. It is not desirable to have several objects mapped to the same hashed value. This would result in a very poor accuracy during a search in the network, because all documents stored under the same codeword would be retrieved while they might have little in common.

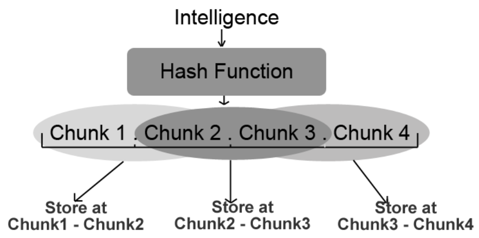

Figure 3: Storing a document "Intelligence" at different locations ($storesize = 2$ and $Nb_{chunks} = 4$)

To avoid this problem, we decided to map each document not to one single codeword, but to a sequence of codewords. By concatenating two codewords together and using these 'double codewords' to index a document, there are $2^{24} > 16$ million possible different locations in the network. To help the error correcting process and also to increase the size of the received vector, more than the two *chunks* used to index a document are created by the encoding functions. By adding more chunks to the vector, there is an increased possibility that some chunks of the RV will have very few errors. The resulting process of the encoding function is shown in Fig. 3. The outputs of this function are used to index the input document in the network.

The values used for most of the research are of 2 for the Storesize, because it allows for over 16 million different 'double codewords', and 3 for the number of chunks. It means that the received vector will be $23 * 3 = 69$ bits long. The number of chunks present in the received vector must be chosen carefully. If there are not enough, the error correcting process will suffer, whereas if there are too many, the network load increases. Choosing 3 chunks is a good compromise between the two.

The values $cDist_o$ and $cDist_q$ determine the number of codewords that will be used to respectively store and search items, i.e the values of c used while computing the *c-neighborhood* of a received vector. They greatly influence the error correcting process by determining how many artificial errors are added to a received vector in order to find the neighboring codewords. These two settings are critical and determine exactly how much of an approximation will be made on the documents and queries. We now look at how the received vectors are created.

3. COUNT ENCODING

The idea behind the *Count encoding* function is to select arbitrarily x 'counting' functions that will simply count the number of occurrences of certain characters in the input being processed. Each one of these functions outputs a value between 0 and $Size(RV) = Nb_{Chunks} \times 23$. This value is taken by adding together an initial value that is independent to the counting function and the number of occurrences of the characters it is responsible for (Fig. 4). An interesting feature of this encoding function is that if you swap the position of two letters in a word, the encoded received vector will still be the same. Very often when people type something quickly, some letters can get mixed up. This is not a problem anymore with this encoding function. But it also means that unrelated words that contain the same letters only in a different order will be found together all the time.

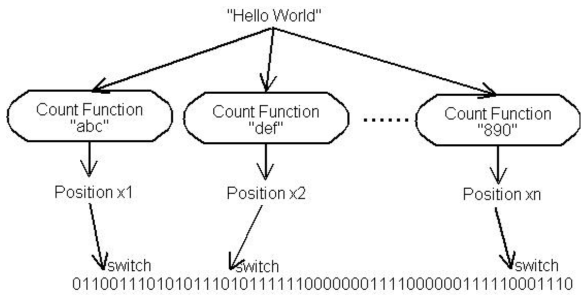

Figure 4: Count encoding technique

If two strings differing by only one character are encoded, two bits in the RV are modified. E.g., if the first word is "Hello" and the second "Helo", the only count function that will not output twice the same value is the one responsible for the letter 'l'. Say in the first case it outputs 27, for the second word it will output 26, thus bits 26 and 27 of the received vector will be different. In order to help the error correction process as much as possible, it is best to scatter the differences between two similar words on the different

”chunks” of our received vector. Since the ECC can correct up to 3 error bits in each chunk, if we have 1 error in each chunk it is better than having 2 errors in the same chunk. To do this, the outputted value of the count function is simply multiplied by : $\frac{Nb_{Chunks} \times 23}{2}$. This will make sure the two different bits are in two different chunks of the RV. This also helps in keeping the distribution as uniform as possible.

We see here another reason for using 3 as the number of chunks and not just 2. Since the encoding of two different characters creates two mistakes in the received vector, by using 3 chunks, there will be one chunk that is not affected by the change and thus the error correcting process has an increased chance of finding the correct codewords.

The distribution of codewords highly depends on how the different characters are distributed to the count functions. If the most frequent characters are all counted by the same functions, then most of the others will always return the same result, thus highly biasing the resulting RV.

4. RESULTS

Codeword Distribution In order to test the functioning of the system, a list of movie titles from the Internet Movie Database (IMDb) is used. As mentioned previously the uniformity of the distribution of codewords is a very important factor. To test this distribution, the first 10’000 movie titles are taken from the list and encoded.

Figure 5 shows the distribution. The horizontal axis represents the different codewords, while the vertical axis is the number of times a movie title is mapped to the corresponding codeword. We can rate the encoding function by computing the variance of the distribution The lower the variance, the better the distribution, since it means that each codeword is hit nearly the same number of times. The variance obtained with the encoding function for the first 10’000 movies in the list is 0.779, meaning that on average two codewords will have a difference of less than 1 in their number of hits. A hit being the fact a file is mapped to a given codeword. By tweaking the count functions and studying the frequency of all characters within this list it would still be possible to obtain a better distribution than the one shown here.

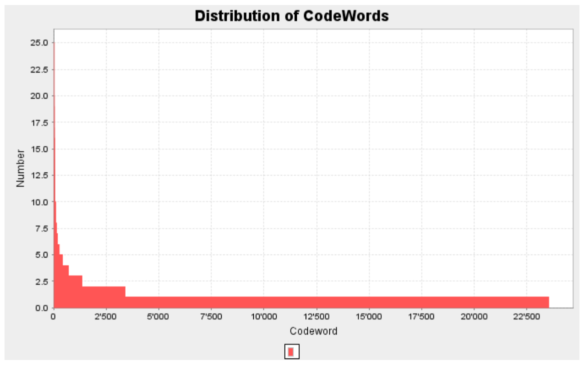

Figure 5: Ordered distribution of codewords for the Count encoding

Error Correction Rate To measure the effectiveness of the error correcting algorithm, several movie titles of different length were selected from the IMDb list. We measure the percentage of times a result is returned while iteratively inserting errors in the query such as removing or adding random characters to the string. In Fig. 6, the results with different values of $cDist$ are shown. We see that the curves are grouped along the different values of $(cDist_o + cDist_q)$. It means that the error correction works just as well whether a document is indexed at a higher number of peers, or if the search process looks for the document at more locations. The values used for $cDist_o$ and $cDist_q$ should therefore be selected depending on the network usage. If the number of resources on the network highly exceeds the expected number of queries, it would be wise to keep $cDist_o$ low and increase $cDist_q$. On the other hand if there are few resources, but a large amount of expected queries, $cDist_q$ should remain low in order to reduce the amount of messages sent on the network during a search process. In this case, $cDist_o$ could be increased since a low number of document shouldn’t exceed the peers indexing capacity.

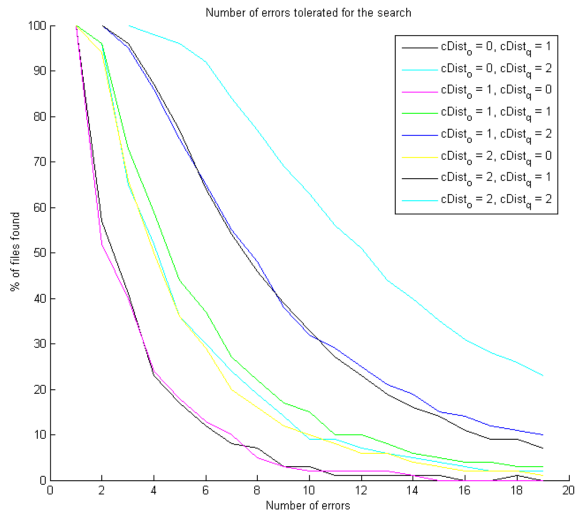

Figure 6: Error correction rate for the Count encoding

The graphic presents the percentage of chance a movie will be found depending on the number of errors introduced. For example if there is a 90% probability of finding the movie ’Matrix: Reloaded’ with one error, it means that 9 times out of ten the algorithm will find the movie if one character is randomly removed or added. Bare in mind that changing one character by another is counted as two errors, since one character is removed and another inserted.

The parameters used for encoding the titles (the number of chunks and the storesize) greatly influence the error correcting results. The higher they are the better the results. The downside being that they create more network traffic and the resulting codeword distribution is also worse. The values used here are of 3 for the Nb_{chunks} and 2 for $storesize$.

Network Performance We now present the results obtained from a simulated network using a hypercube structure. The goal of the simulation is to measure the network activity created by a query. The encoding functions proposed in this paper create keys (or received vectors) of length $Nb_{chunks} * 23$ bits. As in most Distributed Hash Tables, the peers in our network are also represented by a key in the same domain as the documents, so $ID_{peer} \in \Sigma^{Nb_{chunks}*23}$ where $\Sigma = \{0, 1\}$. Each peer is thus responsible for the keywords that are similar to its ID. In the case of a network with 2^{12} peers and $Nb_{chunks} = 1$, each peer is responsible for one single codeword that is equal to its ID.

When the number of peers is smaller than the number of codewords, one peer is responsible for several codewords. The way this is done in the simulation is that the peers ID is truncated to the x most significant bits, where x is the number of dimensions ($log(Nb_{peers})$). The peer is then responsible for all codewords starting with the same x first bits as its ID. E.g., for $Nb_{chunks} = 1$ and 2^{10} peers, each peer will have a 10 bit ID. The first peer might have ID "0000000000" and thus it will be responsible for indexing the documents related to codewords "000000000000", "000000000001", "000000000010" and "000000000011".

Each peer maintains a list of neighbors that contains d entries, where d is the dimension of the hypercube. So a peer has one neighbor for each dimension. In the simulation, the neighbor of peer "0000" (if $d = 4$) on link 0 is the peer that has the same ID as peer A, apart from bit 0 which is inverted, thus peer "1000".

A hypercube network of dimension 3 is shown in Fig. 7. Each peer has one neighbor in each dimension and the way those neighbors are chosen is done as described.

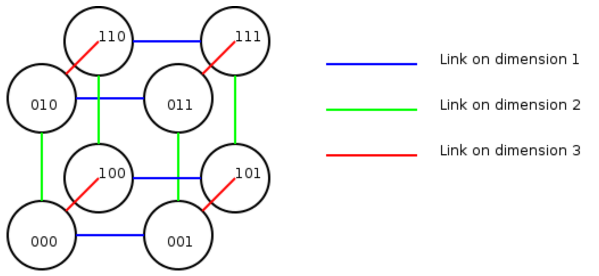

Figure 7: Hypercube network of dimension 3

When a search request is initiated at peer A, the following sequence of actions takes place: **I.** Create received vector from query; **II.** Compute corresponding codewords. **III.** Compute $diff = ID_{peer}$ xor $Codeword$. **IV.** Select x as the smallest index of diff that is equal to 1. **V.** Route search request to the peer on x^{th} link.

At step III, a bitwise xor between the codeword and the peers ID is made to find on which link the search request must be forwarded. If the search request is already at the correct peer B, then x will be null since there will be no difference between the id and the codeword. If this is the case, peer B will search its list of indices and see if it has anything stored under the given codeword. In case of a match, it will send the results back to peer A.

Once the peer responsible for the codeword is reached, if $cDist_q > 0$, peer B initiates the error correcting process. This means peer B computes all $cDist_q$-neighbors of the received vector and initiates a separate search process for each one of them. In the same way as the initial request, the search is forwarded to the peers corresponding to the new codewords and the results are returned to peer A.

The search process is presented in Fig. 8. The routing part corresponds steps III, IV and V in the list above. The number of messages transiting from peer to peer during a query are measured and displayed in Fig. 9. This number of messages is governed by the following equation :

$$Messages_{sent} = (Nb_{chunks} - Storesize + 1) * Avg(Nb_{c-neighbors}) * X \quad (1)$$

This represents the maximum number of messages that will be sent for a search request in the case there is one node

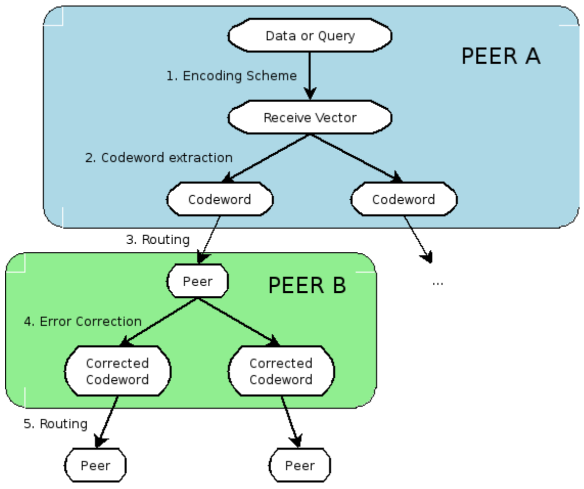

Figure 8: Routing mechanism with approximate queries

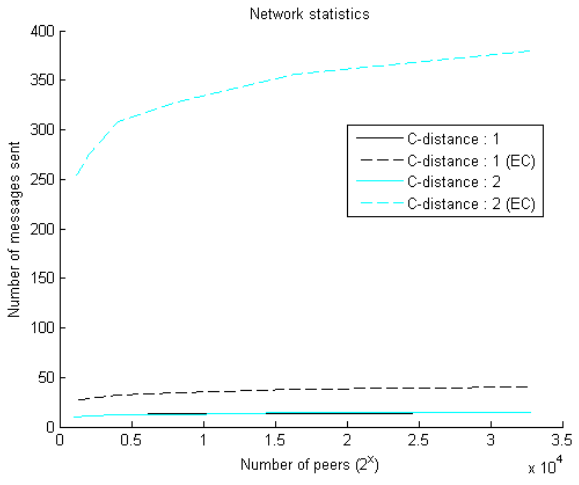

Figure 9: Number of messages sent in the network with $cDist_q = 1$ and $cDist_q = 2$

corresponding to each codeword (ie. a peer is not responsible for multiple codewords). $Avg(Nb_{cneghbors})$ corresponds to the number of neighbors of a codeword given the c-distance. X is the estimated number of messages that need to be sent from a peer to the peer that is responsible for the neighboring codeword. The closer together the Golay neighbors are, the lower this value is. In this simulation, it is usually somewhere between 1 and 3 for $cDist_q = 1$. In Fig. 9, the solid lines represent the number of messages sent by the first step of the search process. The dashed one on the other hand represents the number of messages sent for the error correction process. This number increases greatly with the value of c. Indeed the number of Golay neighbors increases dramatically as we can see below. Unfortunately this limits the use of this algorithm to smaller values of $cDist$.

$cDist_o$	Number of effective codewords
0	1
1	11
2	68
3	324
4	1300

5. RELATED WORK

Approximate queries are still a recent development in P2P research. In [2], Ahmed and Boutaba propose a partially decentralized architecture that also uses the Golay code to enable approximate search. They use a two-level network overlay with peers and superpeers. To represent a file in the network, they pass all the trigrams in the file names through k hash functions to generate a Bloom filter. The superpeers in the network are responsible for the different Golay codewords, while the peers within a subnet contain the actual information. They use special routing mechanisms to route a query from its originating peer to its destination. In a following work, the same authors wrote an article on a Distributed Pattern Matching System (DPMS) [3]. Similarly to their previous work, they use bloom filters as indices for the documents in the network. The q-grams of a query or document description are hashed k times to fill the filter. The bloom filter is then used for subset matching and very effectively enables wildcard searching. Their network overlay is more complicated however and is arranged in several layers taking into account node heterogeneity. The Golay code is not used any more.

Squid [6] proposes to use Space-filling curves (SFC) to preserve data locality when indexing documents. This allows for flexible queries. SFCs are a special mathematical function that transform a multi-dimensional dataset into a value along one single dimension. This replaces the encoding functions presented in this paper. However, data in the network have to be represented by a certain number of terms in d different dimensions so it can be mapped correctly. The advantage of the method proposed in this paper with respect to Squid is that there are no constraints on how the query is formulated. Any given string can be used and the routing hops required to execute the query are stable. Because SFCs do not map the items uniformly, they propose different load-balancing techniques to improve network statistics. Similar enhancements could be applied to our work. [5] presents results for different types of queries.

In [4], Karnstedt et al. propose a way of effecting advanced similarity queries using their *Vertical Query Language*. The concept is to store an item at all of its defining attributes, usually the q-grams present in the file name.

In E-llama [7], the queries are unconstrained, similarly to our work. The difference is in the way the documents and queries are encoded. Instead of using the different characters in the words, queries and documents are transformed into a point in a high dimensional hyperspace. Each dimension corresponds to a given string and an items coordinate in that dimension corresponds to the edit distance between the query or document and the dimensions assigned string. This means that similar strings will receive similar coordinates, thus regrouping them in the hyperspace. The system is then able to retrieve the k-closest elements to a search query. The node that responds to the query iteratively asks its near neighbors to return the closest documents to the query until k items are returned.

[1] employs the *soundex* algorithm to encode documents and queries. This technique uses the different phonemes of a word to create its encoded value. This means that two words that sound the same will share the same encoded value.

All the papers referring to approximate queries on P2P systems try to adapt the DHT hashing mechanism in order to keep similar documents close together. None seems to stand out more than others. All solutions present varied ways of connecting the peers, whether it be in a totally or partially distributed fashion. Again no solution seems to top the others in terms of network usage.

6. CONCLUSIONS AND FUTURE WORK

This paper explains a new algorithm for approximate queries on a totally distributed P2P environment. The initial results are encouraging and further research should be applied to the creation of the network and tuning of the parameters. The way the peers are interconnected highly influences the network load a query creates, and the different parameters drastically change the results of the error correction. It is expected that some fine tuning will result in interesting improvements in the results.

The Golay code might not be the most appropriate error correcting code for this application however. Because of its number of codewords being a little low, the concatenation of several of them is required. It might be more interesting to find a code that has more codewords, and if possible a higher number of bits that can be corrected. This would greatly enhance the error correcting process and thus the results of the algorithm proposed in this paper. The Reed-Solomon (R-S) code for example can be created with arbitrary values for n and k, which are respectively the length and dimension of the code. The minimum distance is then $n - k + 1$ which allows the code to correct up to $\frac{n-k}{2}$ erroneous bits.

In a next step it could be interesting to encode separately each word in a query instead of encoding the search string as a whole. This would allow to search for parts of a movie title thus including a second type of approximation. This would come at a cost since it would induce much more network activity since each word will require a new search process to be run. Also, because single words are shorter than the search queries used in this paper, the encoding algorithm would have to be adapted so that the distribution stays usable. Shorter received vectors would have to be used. Again, another ECC might help.

7. REFERENCES

[1] M. Aharia, A. Chandel, S. Saroiu, and S. Keshav. Finding content in file-sharing networks when you cant even spell. In *Proceedings of the 6th international Workshop on Peer-to-Peer Systems (IPTPS07)*, 2007.

[2] R. Ahmed and R. Boutaba. A scalable peer-to-peer protocol enabling efficient and flexible search, 2006.

[3] R. Ahmed and R. Boutaba. Distributed pattern matching for p2p systems. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS06)*, April 2007.

[4] Marcel Karnstedt and Kai-Uwe Sattler and Manfred Hauswirth and Roman Schmidt. Similarity Queries on Structured Data in Structured Overlays. In *2nd IEEE International Workshop on Networking Meets Databases (NetDB'06)*, Atlanta, GA, USA, April 2006.

[5] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

[6] C. Schmidt and M. Parashar. Analyzing the search characteristics of space filling curve-based indexing within the squid p2p data discovery system, 2004.

[7] B. Wong, Y. Vigfusson, and E. Sirer. Hyperspaces for object clustering and approximate matching in peer-to-peer overlays. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2007.