Journal of Network and Computer Applications 32 (2009) 550–567

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Reducing query overhead through route learning in unstructured peer-to-peer network ☆

Selim Ciraci, İbrahim Körpeoğlu *, Özgür Ulusoy

Department of Computer Engineering, Bilkent University, 06800 Ankara, Turkey

ARTICLE INFO

Article history:
Received 14 January 2008
Received in revised form
20 July 2008
Accepted 16 September 2008

Keywords:
P2P query routing
Parzen Windows estimation
Unstructured
P2P networks
Query caching

ABSTRACT

In unstructured peer-to-peer networks, such as Gnutella, peers propagate query messages towards the resource holders by flooding them through the network. This is, however, a costly operation since it consumes node and link resources excessively and often unnecessarily. There is no reason, for example, for a peer to receive a query message if the peer has no matching resource or is not on the path to a peer holding a matching resource. In this paper, we present a solution to this problem, which we call *Route Learning*, aiming to reduce query traffic in unstructured peer-to-peer networks. In Route Learning, peers try to identify the most likely neighbors through which replies can be obtained to submitted queries. In this way, a query is forwarded only to a subset of the neighbors of a peer, or it is dropped if no neighbor, likely to reply, is found. The scheme also has mechanisms to cope with variations in user submitted queries, like changes in the keywords. The scheme can also evaluate the route for a query for which it is not trained. We show through simulation results that when compared to a pure flooding based querying approach, our scheme reduces bandwidth overhead significantly without sacrificing user satisfaction.

1. Introduction

Querying in unstructured peer-to-peer (P2P) networks like Gnutella is performed by flooding a query message through the network (Androutsellis-Theotokis and Spinellis, 2002). In flooding, each node receiving the query forwards the query to all of its neighbors. Although it is simple and robust, this approach wastes too much bandwidth, because sometimes most of the neighbors who receive the query do not reply. Semantic routing (Joseph, 2002) is one of the many schemes developed to solve this problem, which reduces querying overhead by trying to find the nodes that might send an answer and forwarding the queries towards those nodes. In this paper we provide a solution to this problem by following a similar approach.

Our scheme, which we call *Route Learning*, exploits the keyword repetitions in queries. We observed that in Gnutella there is a high repetition rate for keywords seen in queries (Ciraci et al., 2005; Karakaya et al., 2005) and this repetition follows power-law distribution. To utilize the repetitions seen in queries, the Route Learning scheme employs an adapted solution for the classification problem. Using a solution based on the Parzen Windows technique, the Route Learning scheme learns routes with time, and makes routing decisions for query messages based on this learning.

During the learning process (*training* phase of the scheme), nodes store query messages they receive before forwarding them to all their neighbors using the flooding approach. When these nodes receive query hit messages, they find the matching query messages that were stored earlier and record the number of results returned to the queries. These recorded results are used in the next phase of the scheme, the *evaluation* phase, to make intelligent routing decisions. Intelligent routing allows a node to forward a query to only a subset of neighbors. However, it is still possible for a query to be forwarded to all the neighbors, or to be dropped, depending on the probability (*containing probability*) estimated for each neighbor. The containing probabilities are estimated for each keyword seen in the query. Flooding the query in the evaluation phase occurs only when the containing probability is found to be *not trained* for each neighbor. Although flooding is not desirable in the evaluation phase, it allows the scheme to continue to learn routes. In this way, the scheme can adapt itself to the dynamic nature of P2P networks, such as node disconnections or new node arrivals.

Our experiments in real Gnutella network and our simulation studies show that Route Learning significantly reduces the bandwidth overhead necessary for querying compared to a pure flooding based approach that uses no intelligent routing. It is shown through experiments in Gnutella network that our scheme consumes only about 28% of the bandwidth that would be

☆ This work is partially supported by The Scientific and Technical Research Council of Turkey (TUBITAK) through Grants 104E028 and 105E065.

* Corresponding author.
E-mail address: korpe@cs.bilkent.edu.tr (İ. Körpeoğlu).

1084-8045/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2008.09.001

consumed in a pure flooding based P2P network. Our simulation results show that the decrease in the rate and quality of answers in a P2P system using Route Learning is not much compared to the rate and quality of answers in flooding based systems. We also conducted experiments in the Gnutella network to compare Route Learning to random walk (Lv et al., 2002) and query caching (Kalogeraki et al., 2002; Yang and Garcia-Molina, 2002) schemes. In query caching, the whole query text is cached with the neighbor information that supplied the answer, and if the same query text is seen again, the query is forwarded to this neighbor. Our experiments show that the intelligent decisions that Route Learning makes helps to reduce the querying overhead significantly without reducing the answer rate much. Route Learning, however, requires more memory space at a node, and therefore it should be used in platforms where memory capacity is not very low.

The rest of this paper is organized as follows. A brief description of classification problems and Parzen Windows estimation is provided in Section 2. Our scheme and the related algorithms are described in Section 3. The experiments we conducted to evaluate our routing scheme and the respective results are presented in Section 4. In Section 5, recent research related to our work is described. Finally, in Section 6 we summarize our conclusions.

2. Background

Our semantic routing scheme is based on the solution of the general classification problem. Therefore, in this section we briefly describe what the classification problem is and give one popular solution approach for it that is based on Parzen Windows estimation.

A classification problem consists of a classifier and two or more classes to be chosen. The classifier tries to classify a given object according to its features; ideally the features that belong to a class should be distinct from other classes (Witten et al., 2000). The classifier has to be trained to get information about classes before starting classifying. This information determines the location of each class in the feature space. For example, assume that we are classifying red objects. For this problem we have two classes, red and non-red, and the features of each class are the red, green, blue (RGB) values of the objects. Here we model a very simple classifier that works according to the nearest-neighbor rule, where an object is labeled with the class whose features are nearest to the features of the object. In the learning phase, we provide the classifier with some samples and specifically mark which class each sample belongs to. Then, in the evaluation phase we provide an object to the classifier, which labels it with the class whose "color" is closest to that of the object.

The simple classifier we have used in the above problem gives good results (twice the error rate of a Bayesian classifier, Witten et al., 2000) with unlimited samples. However, in most cases we do not have that many samples. Thus we have to use classifiers that take a statistical approach. The primary statistical solution to the classification problem lies in the Bayesian decision theory:

$$P(\omega_i|\bar{x}) = \frac{P(\bar{x}|\omega_i)P(\omega_i)}{\int_i P(\bar{x}|\omega_i)P(\omega_i)} \quad (1)$$

Here, ω_i represents class i, and $\bar{x}$ represents the features of an object for which we want to determine the class. The left-hand side of the equation, $P(\omega_i|\bar{x})$, is the probability that an object with features $\bar{x}$ is in class ω_i. The term $P(\bar{x}|\omega_i)$ is called *the likelihood* or *the class conditional density*. Estimation of this parameter is the crucial part of the classification problem (Witten et al., 2000). If the probability distribution of the likelihood is known, then we could use some parameter estimation techniques to find the parameters of the distribution. However, in many cases we do not have this valuable information; thus we use some non-parametric techniques like *Parzen Windows* to estimate the distribution.

Next, we provide a simple description of Parzen Windows estimation and algorithms of a generic Parzen Windows classifier. The theoretical background of this non-parametric techniques can be found in Witten et al. (2000).

The probability density of vector $\bar{x}$ that fall into a very small region R in feature space is given in (Witten et al., 2000)

$$P(x) \cong \frac{k/n}{V} \quad (2)$$

In this expression, k stands for the number of samples that fall in volume (V) enclosed by R, and n stands for the total number of samples in the set. Clearly, to estimate this probability we can take two approaches. In the first approach, the one used in Parzen Windows estimation, we can take a fixed volume and count the number of samples in the volume. In the second approach, we can select a constant number of samples each time, and then calculate the volume that encloses each of these samples.

The training phase of a Parzen Windows classifier is very simple. Samples are placed in the feature space of the class they belong to, which is a d-dimensional array. In the evaluation phase, the number of samples within the volume centered at the given feature vector (of an object) is counted for each class. Then the probabilities are estimated according to the formula given in Eq. (1). Eventually the object with that feature vector is assigned to the class that gives the maximum probability (Fig. 1).

3. Route Learning

Our scheme is designed for unstructured P2P networks like Gnutella. These networks are characterized by their totally decentralized nature where all nodes have equal responsibility, flooding is used as the query dissemination mechanism, and a query hit message is returned by a peer that has one or more matching resources. A query includes a search string which may consist of one or more keywords. Normally, without our scheme applied, a peer forwards a query message to all its neighbors if the time-to-live (TTL) value stored in the query is not zero.

In our Route Learning scheme, a peer tries to estimate the neighbors that will most likely reply to queries. Peers calculate this estimation based on knowledge that accumulates gradually from query and query hit messages sent to and received from neighbors.

Route Learning inherits its basic idea from the classification problem where a peer having n neighbors has n classes to choose from to forward a query. Each class corresponding to a neighbor i can be used to find out the probability of having the resource at or reachable by neighbor i. We call this probability the *containing probability* of that class. Since we do not know the probability distribution of user-submitted queries, we have solved this classification problem by using an adapted version of Parzen Windows density estimation technique (Witten et al., 2000).

The scheme, which is to be executed by all peers, maps (hashes) each keyword in a query to a point in a 1-D feature space that is created for each neighbor the peer has. As depicted in Fig. 2, each point k of the feature space has two numbers associated with itself. The first number (*answer-count*) is the number of answers returned through that neighbor for keywords mapping to point k, and the second number (*query-count*) is the number of queries made to point k. The system estimates the containing probability by applying division on these two numbers.

```
ParzenTrain(TrainingSet T)
for i = 1 to length(T) do
  FeatureSpaces[T[i].class][T[i].features] += 1;
end for

ParzenEvaluate(Feature F, int WindowSize)
for i = 1 to NoOfClasses do
  count[i] = 0;
end for
for j = 1 to NoOfClasses do
  for i = F - WindowSize / 2 to F + WindowSize / 2 do
    count[j] += FeatureSpaces[j][i];
  end for
end for
for i = 1 to NoOfClasses do
  probability[i] =
    (count[i] / FeatureSaces[i].totalNoOfSample) / WindowSize);
end for
return MaxClass(probability);
```

Fig. 1. Functions of a Parzen Windows classifier with 1-D features: training and evaluation phases.

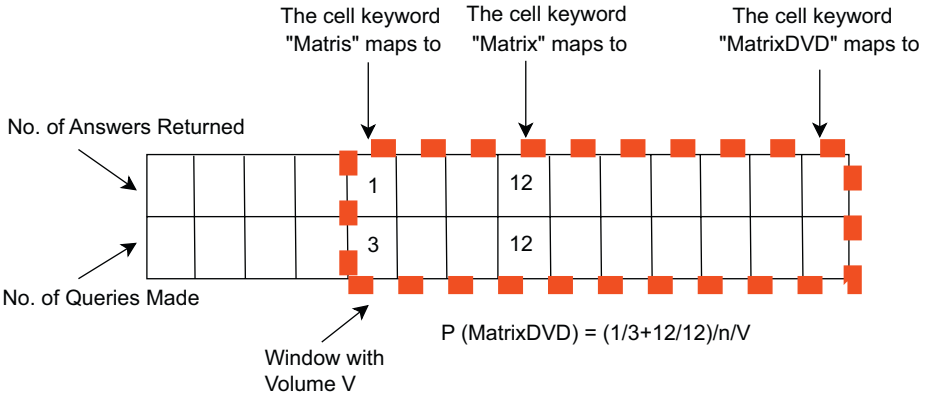

Fig. 2. Example of a feature space maintained by the scheme for a neighbor. The calculation of the containing probability of the keyword "MatrixDVD" is also shown. The window function only adds the containing probabilities that fall within the volume. *n* denotes the total number of containing probabilities in the feature space; the number of samples.

The proposed scheme investigates the keywords in a query and indexes (maps) them separately in order to cope with variety in the queries submitted by users. For example, assume that a user has submitted a query that is composed of keywords $K1$, $K2$, and $K3$. If the system indexed these three keywords together (not separately), it would estimate the containing probability for a query with keywords $K1$ and $K2$ as zero. At the same time, the scheme should also have a carefully designed mapping algorithm that maps related keywords (e.g., two keywords where one is the substring of the other) to points that are close to each other in the feature space. In this way, the Parzen Windows' window function can include these keywords' probabilities in estimating the containing probabilities. For example, in the scenario depicted in Fig. 2, the peer "knows" keywords "Matris", and "Matrix". That is the peer has received earlier queries for these keywords, and received related query hits from the neighbor for which the feature space shown in the figure is maintained. Now, if the peer receives a query "MatrixDvd" for the first time, it is not trained for this keyword, but still it does not need to flood the query, since the keywords "MatrixDvd" and "Matrix" can be placed in close proximity and therefore the query can only be forwarded to the neighbor that provided answer to the query "Matrix". When these keywords can be mapped close to each other in the feature space, the window function can include the containing probability of the keyword "Matrix" and can send the query "MatrixDvd" only to the neighbor that supplied the answer for "Matrix".

The Route Learning scheme consists of three phases: *training*, *evaluation*, and *network change adaptation*. In the following sections, we describe each of these phases.

3.1. Phase 1: training

Upon joining the P2P network, a peer has no knowledge about its neighbors, therefore it is impossible for the peer to make routing decisions. For this reason, the newly joined peer has to

flood a few queries and record the responses returned from its neighbors. In other words, Route Learning uses the ongoing query traffic to learn what the neighbors are sharing; the scheme does not make use of any previously assigned data set to train with or it does not need to send "probe" messages to neighbors to learn what the neighbors share. It is important to note here that when the training phase ends, the training of the scheme does not end. This is due to two reasons; the first one is that the peer may encounter a keyword or keywords that it has no knowledge of. The second reason is that in a P2P network peers frequently connect and disconnect, so the knowledge kept by the scheme should be updated and this is achieved by further training. However, as we discuss in Section 3.3, this further training is not done by flooding the query to all neighbors. So this initial training phase is the only time the scheme just floods queries, and it is executed only once.

To extract the keywords from a query, we split it to tokens with the same character set used by many Gnutella clients. This character set includes the characters "._-*()",;:!?". Here, deciding on the size of the feature space is an important issue, since there is no limit on the length of the keywords submitted in a query. Ideally, we should create an infinite feature space, but this is not feasible to implement and we need to limit the feature space size. A very small feature space will cause queries to be forwarded to every neighbor (like flooding). This will cause a lot of query overhead, which is contrary to the main purpose of our scheme. However, a small feature space may have more success in getting replies. On the other hand, a very large feature space will cause queries to be forwarded in a more directed manner to neighbors, thus decreasing the query overhead. But the chance of getting replies to queries may be less than that with a small feature space.

The algorithm for the training phase is divided into two parts (Fig. 3). The first part, the function RouteLearnTrain starts by creating the feature spaces and closed bucket hash tables for each neighbor the newly joined peer has. The hash tables are used to store the keyword maps of each query that arrives during the training session. It is important to note here that the CreateFeatureSpace function, after allocating the memory for the feature space, sets both query-count and answer-count values of each feature space point to −1. This is used to indicate that the keywords are not seen yet (unknown). In this way we can distinguish between keywords that are unknown and keywords that are seen but did not get any hits.

After creating the feature spaces, the training algorithm starts listening to the query messages. Upon receiving a query message, the algorithm calculates the points where each keyword in the query maps to; and updates the respective entries in the queries-hash-table and feature space. This is done for each neighbor. The mapping is done by calling the Map function, whose pseudo-code is given in Fig. 4. This function first divides the query text into keywords and then applies the mapping function to calculate the points where each keywords maps to. Updating the feature spaces for the keywords seen in a query is done by calling the InsertQuery function, whose pseudo-code is shown in Fig. 4. For each keyword, this function first checks the feature space point the keyword maps to. If a point's values (answer-count and query-count) are found to be equal to −1, meaning that the scheme did not receive earlier any keyword mapping to this point, the query-count for that point is set to 1 and the answer-count is set to 0.

```
RouteLearnTrain(int NumberOfNeighboors, int FeatureSpaceSize, int TrainCount)
int NoOfClasses = NumberOfNeighboors;
for i = 1 to NoOfClasses do
  FeatureSpaces[i] = CreateFeatureSpace(FeatureSpaceSize);
  queries_hash_table[i] = CreateHashTable();
end for
Train = 0;
while Train < TrainCount do
  Query_Data query = GetQuery();
  query.maps = Map(query.data);
  for i = 1 to NoOfClasses do
    queries_hash_table[i].Insert(query.QUID, query);
  end for
  InsertQuery(query, NoOfClasses);
  Train=Train+1;
  Flood(query);
end while

RouteLearnQueryHit(Query_Hit Q, int AnswerFromPeer)
count = CountAnswers(Q.data)
Query_Data qdata = queries_hash_table[AnswerFromPeer].Get(Q.QUID);
if qdata == NULL then
  return;
end if
for i = 1 to Length(qdata.maps) do
  InsertAnswer(qdata.maps[i], count, AnswerFromPeer);
end for
```

Fig. 3. The training algorithm. It has two main functions.

```
Maps[] MAP(String queryText)
StringTokenizer queryKeywords =
    new StringTokenizer(queryText,".-*()\",;:!?");
Maps[] toreturn = new Maps[queryKeywords.countTokens()];
int i = 0;
while queryKeywords.hasMoreTokens() do
  toreturn[i] = MappingFunction(queryKeywords.nextToken());
  i = i+1;
end while
return toreturn;

InsertQuery(Query_Data q, int NoOfClasses)
for j = 1 to NoOfClasses do
  for i = 1 to Length(q.maps) do
    if FeatureSpaces[j][q.maps[i]].QueryCount == -1 then
      FeatureSpaces[j][q.maps[i]].QueryCount = 1;
      FeatureSpaces[j][q.maps[i]].AnswerCount = 0;
    else
      FeatureSpace[j][q.maps[i]].QueryCount++;
    end if
  end for
end for

InsertAnswer(int keywordMap, int AnswerCount,
          int AnswerFromNeighboor)
FeatureSpace[AnswerFromNeighboor][KeywordMap].QueryCount +=
    (AnswerCount - 1);
FeatureSpace[AnswerFromNeighboor][KeywordMap].AnswerCount +=
    AnswerCount;
```

Fig. 4. Map, Insert Query and Insert Answer functions. Note that *NoOfClasses* is equal to the number of neighbors a peer has.

The answer-count is set to 0, because in a Gnutella type of network, the peers do not send a reply message when they do not have a resource that matches a query they receive. So until a reply arrives, the scheme declares that the neighbors cannot supply a resource that matches the query. If, on the other hand, the values at the point are not equal to -1, which means the peer has encountered the keyword earlier, the query-count value is increased by one.

The second part of the algorithm for the training phase, the function RouteLearnQueryHit in Fig. 3, is executed upon the arrival of query hit messages. When a query hit message is received, the mappings of the keywords seen earlier in the corresponding query are extracted from the queries-hash-table by using the *QUID* field value of the query hit message. Then, for each mapping, the InsertAnswer function, whose pseudo-code is shown in Fig. 4, is executed to update the feature space. By this operation, the answer-count in the respective point of the feature space is increased by the number of answers (resource names) found in the query hit message, and the query-count is increased by the number of answers found -1. We subtract one since the first part of the algorithm had increased the query-count by 1 already when the query had been forwarded to neighbors. Here, we see another difference between the Route Learning scheme and a generic classifier. In a generic classifier, the sample set used for training is already classified and these data are used to adjust parameters of the classifier. In Route Learning, on the other hand, the peer only uses ongoing query traffic, which does not provide which neighbor has the answers to the given query. When a query is received, the scheme puts each keyword of the query to the "class" for each neighbor, but marks the feature space cells as zero-answers from each neighbor; this is done by setting the answer-count variable of the appropriate cells of the feature space to 0 in the InsertQuery function. Then, Route Learning floods the query to actually find out which neighbor(s) can supply the answer. If answers are supplied to the query, then the appropriate cells in the feature space of the neighbors that have sent the answers are updated. This increases the containing probability of these neighbors above zero, which in turn indicates to Route Learning that these neighbors may supply answers to these keywords. Fig. 5 provides a simple example about how training is done.

As mentioned earlier, we require the mapping algorithm to map related keywords to close points in the feature space. To support this, we used a *Radix* 32 based mapping function shown in Eq. (3). In the equation, S denotes a keyword, and $|S|$ denotes the size of the keyword. With this function, if we would have known the maximum number of characters contained in a query keyword (this keyword would map to the last cell), then we could be able to initialize the feature space so that it could accommodate all the possible keywords in a separate cell. However, since we do not know (or it is impossible to know) the length of the largest keyword, we need to place a limit on the keyword length (the number of characters that the scheme will operate with) and create the feature space accordingly. For example, assume we

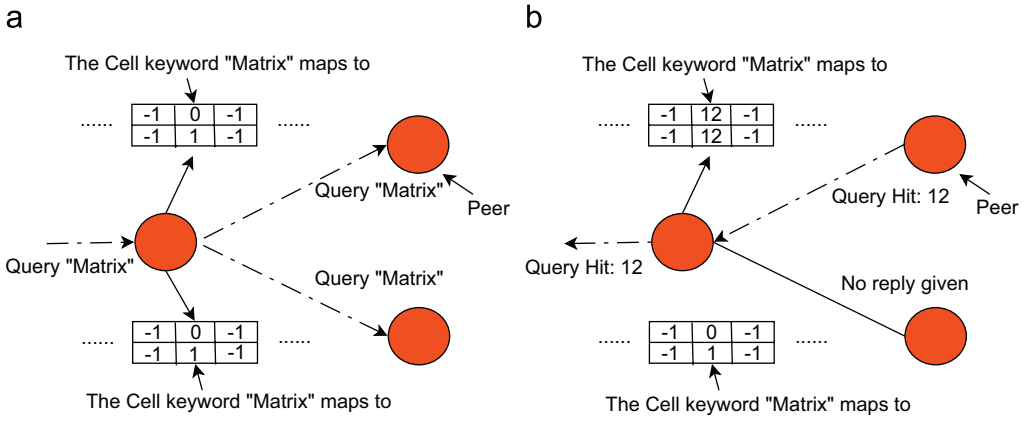

Fig. 5. Training algorithm execution. (a) The query "Matrix" arrives, the InsertQuery function is executed and then the query is flooded. (b) A query hit from a neighbor arrives and the feature space of that neighbor is updated. Then the query hit message is routed back.

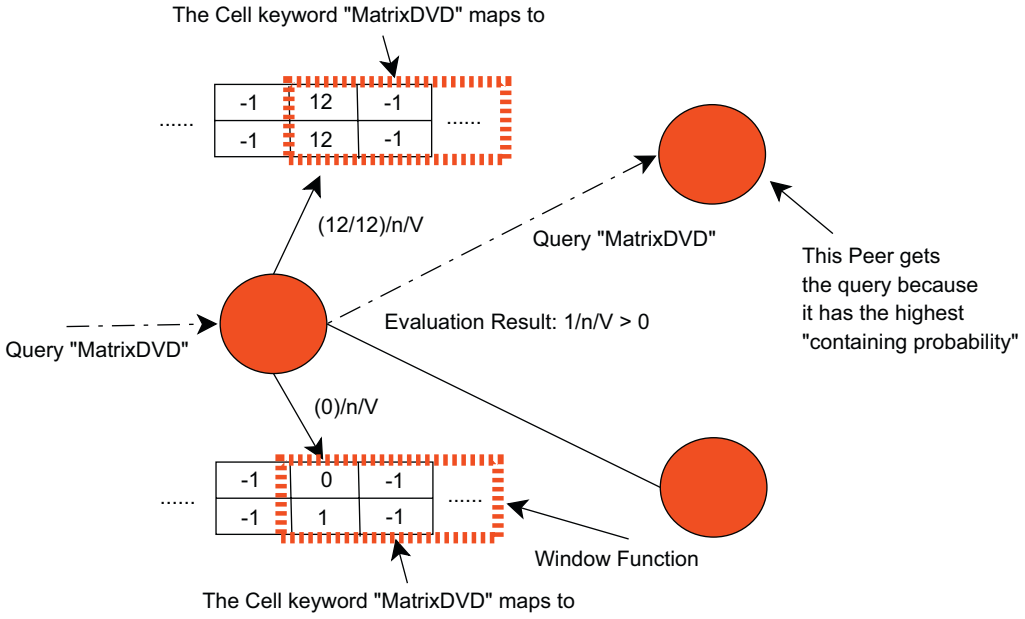

Fig. 6. Example of how evaluation algorithm works when the query "MatrixDVD" is submitted. Keyword "MatrixDVD" is extracted and the point where it maps is calculated. The window is centered at this point and the number of samples fall into the volume of the window is calculated for each neighbor. In this example, one peer has 1 ($\frac{12}{12}$) sample and the other one has 0 sample. The query is forwarded to the peer with one sample.

have set the limit to three characters, then the keyword *a* would map to the first cell and *zzz* would map to the last cell of the feature space. The keywords, *aaa* and *aab* would map to separate cells while the keyword *aaaa* and *aaab* would map to the same cell. Here, the keywords that map to the same cell may be very closely related (e.g. Matrix and MatrixDVD). But it is also possible that they may be very unrelated (e.g. Matrix and MatrixDVDRipper). Thus, the selection of the limit on the keyword length is a very important parameter of the scheme as we discuss in Section 4.1. A benefit of imposing such a limit, on the other hand, is that it reduces the memory requirement of the scheme, as we present in Section 4.2:

$$F(S) = \sum_{k=0}^{|S|} (S(k) - 96) \times 32^{-k} \tag{3}$$

The Radix 32 function only gives a floating point number, but we need a function that gives an integer number so that we can use this integer number as an index to address the feature space (which is an 1-D array). That is, we want the keyword *a* to map to index zero while the keyword *zzzzz*... (*i* number of *z*'s) to map to the last index of the array. To achieve this mapping, we modify the Radix 32 function and use the function given in Eq. (4), which we call *Array Radix* 32 function. This function limits the Radix 32 function to include only the first $i+1$ characters of a keyword. With this function, for example, the keyword *aaa* maps to 33 and the keyword *aab* maps to 34, assuming *i* is 2.

$$F(S) = \left(\sum_{k=0}^{i} (S(k) - 96) \times 32^{-k} \right) \times 32^{i} - 32^{i} \tag{4}$$

3.2. Phase 2: evaluation

This phase of the scheme can be executed at a peer after the training phase is finished. At that time, the peer becomes ready to make intelligent decisions about which routes to forward the queries through. Before describing the related algorithms,

```
RouteLearnEvaluate(Query_Data query, int toSendPeers,
        int WindowSize, int NoOfClasses)
query.maps = Map(query.data);
probs = RankDecision(query.maps, NoOfClasses, WindowSize);
if all elements in probs[0] is 1 then
  for i = 1 toNoOfClasses do
    queries_hash_table[i].Insert(query.QUID, query.data);
  end for
  InsertQuery(query.maps[j], NoOfClasses);
  Flood(query);
end if
for i = Length(probs) down to 1 do
  sendpeers = sendpeers
    ∪ FindMaxs(probs[i], toSendPeers);
  if Length(sendpeers) != toSendPeers) then
    toSendPeers -= Length(sendpeers);
    continue;
  else
    break;
  end if
end for
for j = 1 to Length(sendpeers) do
  FowardQuery(Q, sendpeers[j]);
end for
```

Fig. 7. Evaluation algorithm. Note that *NoOfClasses* is equal to the number of neighbors a peer has.

however, we will give a simple example about how evaluation works and routing is performed. Assume we have a peer having the two feature spaces shown in Fig. 6 for two of its neighbors. Assume the peer has finished the training phase. Now, when the peer receives a query with one keyword, *MatrixDVD*, the keyword is mapped to a 1-D point and the window is centered at that point in both feature spaces. Here, the length of the windows is the volume, in which the samples are searched. For both feature spaces, the number of samples that fall into the window is counted and containing probabilities are computed. In this example, only one neighbor will have containing probability greater than zero, and the query will be forwarded to that neighbor. The cells with value -1 are not considered in computing the containing probabilities. The -1 value indicates that the keyword is not trained yet (is not seen earlier).

Fig. 7 shows the evaluation algorithm used in this phase, which is an adaptation of the Parzen Windows evaluation algorithm. As seen in the algorithm, Route Learning does not route a query to all the neighbors, but rather to a subset of the neighbors. The size of this subset is fixed and is expressed by the value of the *toSendPeers* parameter. This is another difference between Route Learning scheme and a generic classifier. In a generic classifier, a feature vector is placed into one class; though in Route Learning a feature vector is placed into *toSendPeers* classes. When a query arrives, the algorithm first determines the mapping of each keyword in the query. Then for each neighbor and keyword, the windows are placed and the volume covered by the windows is searched (the size of the windows is determined by the *windowSize* parameter). For each cell that falls within the volume covered by the window, the *containing* probability is computed; this is the ratio of the answer-count to the query-count stored in the cell. The containing probability of a keyword is then calculated by adding these *containing* probabilities computed for the cells falling within the volume.

To decide to which neighbors to forward the query, we use a ranking mechanism that is similar to the one used in NeuroGrid (Joseph, 2003). Our decision algorithm, however, completely ignores neighbors whose containing probability for a query is found to be zero to save further bandwidth. The decision function, shown in Fig. 8, gives the highest rank to non-zero containing probabilities of all keyword matches and the lowest rank to the containing probabilities of one keyword matches. The containing probability of a neighbor for a query is calculated by *multiplying* the retrieved containing probabilities of each keyword (seen in the query) from the feature space of that neighbor. Thus, according to this formula, if one keyword's containing probability is found to be zero for a neighbor's feature space, then the query's containing probability becomes zero, for which there is no need to forward the query to. Here, we have used multiplication, since in Gnutella queries, each keyword is implicitly connected to other keywords with logical "*and*" operation. That is, a peer receiving a query with keywords $K1$ and $K2$ can answer the query only if a resource's name contains both keywords $K1$ and $K2$. If, during query evaluation, a keyword is found to be not known (never seen before), i.e. for all cells that fall into the volume both answer-count and query-count values are equal to -1, the neighbor is given a lower rank. If the query is found to be not known for all neighbors, then all neighbors will be added with rank 0 and a containing probability equal to 1 $(-1/-1)$. While discussing the training phase (Section 3.1), we said that the training of the Route Learning scheme does not end. When a query is declared to be not known, Route Learning has to learn which neighbors can supply answers to this query. Thus, the keywords of the query are inserted to the feature space of all neighbors using the InsertQuery

```
RankDecision(Maps M, int NoOfClasses, int WindowSize)
currentNeighbor = 0;
for j = 1 to NoOfClasses do
  rank = Length(M);
  neighborProb = 1;
  for i = 1 to Length(M) do
    allUnknown = true;
    for k= M[i] - (WindowSize / 2) to M[i] + WindowSize / 2 do
      if FeatureSpaces[j][k] != -1 then
        currentProb += FeatureSpaces[j][k];
        allUnknown = false;
      end if
    end for
    if allUnknown then
      rank -= 1;
    end if
    if currentProb == 0 then
      rank = -1;
      break;
    end if
    neighborProb *= currentProb;
  end for
  if rank != -1 then
    allProbs[rank].Add(j, neighborProb);
  end if
end for
return allProbs;
```

Fig. 8. Rank decision algorithm. Note that *NoOfClasses* is equal to the number of neighbors a peer has.

function (Fig. 4) and then the query is flooded. When a query hit message arrives for this query, the RouteLearnQueryHit function is executed and Route Learning learns the neighbors that can answer the query with these keywords. In summary, Route Learning drops the query (that is, the query is not forwarded to any neighbor) if it decides that query cannot be answered (for all neighbors the containing probability is 0), forwards the query to a selected number of peers (the peers with the highest rank and containing probabilities), or floods the query (that is the query is sent to all neighbors) if it decides that it did not see the query before, i.e. during training phase (this is found out if the containing probabilities for all neighbors are found to be 1 at the lowest rank). Thus, flooding a query only happens when all keywords in a query are not seen before for all neighbors.

After ranking each neighbor, the evaluation algorithm continues by calculating the set of neighbors to which the query will be forwarded. To achieve this, the algorithm invokes the FindMaxs function with the highest rank. This sorts the results obtained from each neighbor's feature space and returns the top *toSendPeers* number of neighbors for that rank. If the number of neighbors at that rank is lower than *toSendPeers*, the evaluation algorithm lowers the rank and re-executes the steps described above.[1] This process continues until *toSendPeers* number of neighbors is added to the set of neighbors that will receive the query. Then the queries are forwarded to these neighbors.

[1] The highest rank is equal to the number of keywords in the query.

3.2.1. Improving decision making: Vote Route Learning

During the training phase, there may be cases where popular keywords (keywords that can generate answer) may not generate query hit messages. This may cause Route Learning to drop queries containing these popular keywords. For example, assume in the training phase the scheme has received the query "matrix avi" and this query has not generated hit messages. The keyword "avi" here is a popular keyword and has a high probability of being repeated. Now let us assume another query "eternal sun shine of the spotless mind avi" is received. Since "avi" was seen before and no answer has been received (i.e., the containing probability is zero), the query will be dropped. This reduces the number of answers generated for this query. To fix this, we add a comparison mechanism to the decision making algorithm. This mechanism counts the number of keywords that are unknown (i.e., not seen before) and the number of keywords that have zero containing probabilities for a query. If the number of unknown keywords is greater than the number queries with zero containing probabilities, the query is flooded. If, on the other hand, the number of keywords with zero containing probabilities is greater, then the query is dropped. We call this modified version of Route Learning as *Vote Route Learning*.

3.3. Phase 3: network change adaptation

The knowledge a peer builds about its neighbors can be outdated due to disconnections or connections of new neighbors. This phase

of the algorithm allows a peer to probe for changes in the neighboring peers and adapt its routing knowledge accordingly. Direct neighbor changes (the disconnection and the connection of a peer at one hop distance) can be detected easily by the peers connection manager algorithm (usually Gnutella clients employ a connection manager that waits for incoming connections and manages failed connections). When a neighboring peer disconnects, Route Learning removes the knowledge it builds up for the peer, so queries are no longer forwarded to this peer. Upon the arrival of a new neighbor, Route Learning initiates the training algorithm only for the newly joined peer. Thus, the queries are always forwarded to this neighbor until the training phase is over.

On the other hand, adapting to neighborhood changes at distances $TTL > 1$ cannot be detected by the connection manager. One method to cope with these changes is periodically running the training algorithm for all neighbors. As another method, peers can flood their feature spaces to keep the knowledge up to date (this approach is used in literature by various schemes in which the peers flood the knowledge they collect about their neighborhood, Kalogeraki et al., 2002; Yang and Garcia-Molina, 2002). However, running the training algorithm may not be very effective and flooding the knowledge may require extra messages to be employed by the Gnutella protocol. In Gnutella protocol, a peer sends *BYE* messages to notify the neighboring peers that it is disconnecting from the network. Thus, a better way is to detect changes in the network by using *BYE* messages. When Route Learning receives a certain amount of *BYE* messages from a neighbor, it can run the training algorithm only for that neighbor. This way, it can adapt itself to the changes in the network without employing extra messages and flooding the queries.

4. Performance evaluation

In this section, we report our results regarding the performance of the Route Learning scheme and its comparison against flooding. Besides that, since evaluating the scheme requires decision on the default values of some important parameters, we also describe in detail how we have determined those values.

4.1. Keyword length

As presented in Section 3.1, in order to reduce the memory requirements of the scheme, we need to limit the number of characters the scheme uses while calculating the array index. Thus, the value of this parameter should be selected in a way that the scheme should only need to truncate a small number of keywords. Fig. 9 shows the distribution of the number of characters contained in the keywords collected from the Gnutella network. These data show that the number of keywords that are longer than five characters is only about 35% of all the keywords. Therefore we set the keyword length parameter value to 5.

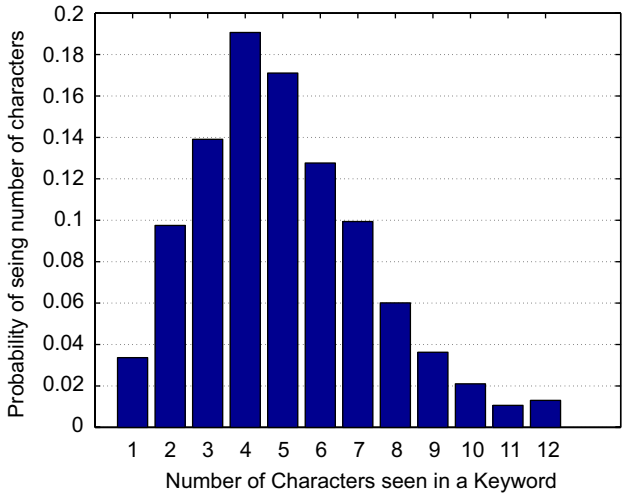

Fig. 9. Distribution of number of characters found in a keyword. The data are calculated over 100 000 queries collected from Gnutella network.

4.2. Feature space size

As shown above, if we limit the keyword size to 5, we would only be truncating about 35% of the keywords. In this case, the maximum index value required is 32^5 (the index the keyword *zzzzz* maps to). Since in an array cell the scheme holds two short values, the memory requirement for the feature space adds up to about 128 MB per neighbor. However, not all of these cells will be used. During the lifetime of a peer, it receives only a small percentage from all the possible keywords due to short lifetimes and high keyword repetitions. Saroiu et al. (2002) show that the median duration of a peer in Gnutella network is 60 min (we further comment on this in Section 4.5).

To see the effects of keyword repetitions, we conducted a simple experiment in which we extracted 361 889 keywords from 100 000 queries collected by our Gnutella crawler, and mapped them to an array of size 32^5 using the Array Radix 32 mapping function. In Fig. 10, the increase in the number of used cells is shown. Initially the increase is linear; however, as more keywords are inserted, the increase slows down. Overall, when 361 889 keywords are inserted into the array, only 57 762 cells are used; thus, due to high repetition rate the system wastes memory.

To reduce this memory waste, we used hash tables to store feature space values. To evaluate the performance of this implementation, we inserted 361 889 keywords into hash tables with sizes 32^2, 32^3, and 32^4, and recorded the number of rehashes made, which doubles the size of the hash table, with load factor set to 0.75. As can be seen from Table 1, the hash table size of 32^4, which requires about 4 MB of memory, has led to 0 rehashes; thus for our experiments we used this initial size. However, if memory size is a problem, a hash table of size 32^3 can also be used.

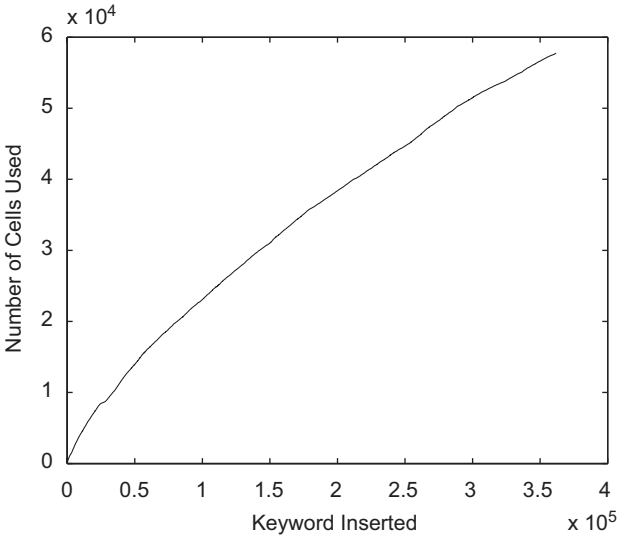

Fig. 10. Feature space cells used when 361 889 keywords are inserted into the feature space. Overall for a feature space size with 32^5 only 0.18% of cells are used.

Table 1
Number of hash table rehashes with different initial sizes when 361 889 keywords are inserted into the hash table

Hash table size	Number of rehashes
826	7.0
26 458	2
846 682	0

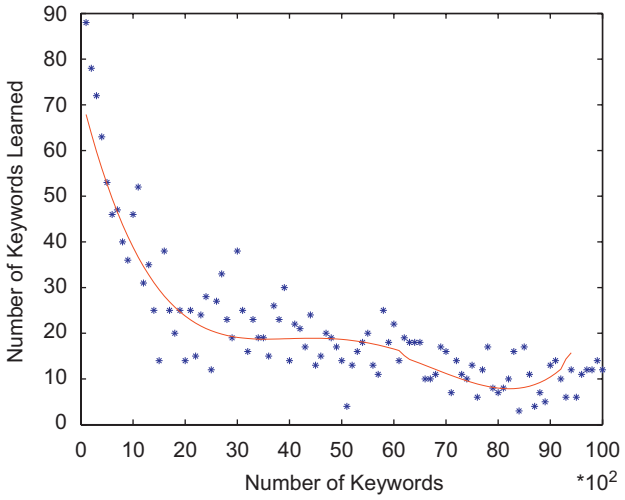

Fig. 11. The plot shows that as queries arrive the number new keywords seen drops rapidly. From the first 100 keywords, the scheme learns 98 of these, however, from keywords 900 to 1000, the scheme learns 46 new keywords. The line shows the smoothed data.

4.3. Number of queries used in training phase

To find the number of queries needed for initial training, we collected keyword sets, each containing 10 000 keywords, from different peer sets (a peer set contains five different connections) at Gnutella network. We inserted these 10 000 keywords to a feature space of size 32^4, and for each keyword we logged whether the keyword is seen before or not. Then, we counted how many new keywords are seen at 100 keyword intervals. To combine the results of the different keyword sets, we took the average of the number of new keywords seen. In Fig. 11, we see that in the beginning nearly all the keywords are new to the scheme; for example in the first 100 keywords, 98 keywords are new. As the scheme receives more keywords, however, the number of new keywords seen starts dropping rapidly (e.g. for the 900th–1000th keywords only 46 are new). From the figure, it can be clearly seen that after the 1000th keyword, the number of new keywords seen stays at lower rates. From this, we can say that training the system with 1000 keywords is sufficient. According to the data collected, a peer has to receive about 300 queries to receive 1000 keywords. Thus, we conclude that initial training should be done with 300 queries.

4.4. Window size and its effect on prediction accuracy

To determine the default value for the window size parameter, we have conducted experiments, in which we simulated a small but representative part of a P2P network (Fig. 12). In this small network, there is one Route Learning enabled *root* node with six neighboring nodes, one of which is assumed to be connected to the whole network and queries come from this node. The other

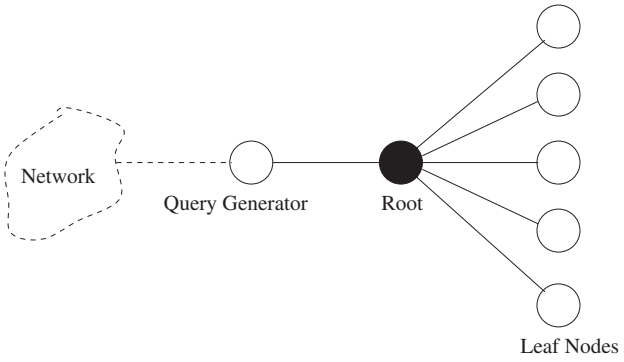

Fig. 12. Simple P2P network model.

five nodes are *leaf* nodes that share resources. The following experiments were conducted to determine the window size value:

Exact-match-error experiment: In this experiment we investigated how the protocol performs for repeated queries, and we tried to find an appropriate value for the window size parameter. This is important because, if this parameter is set to a large number, the protocol will estimate incorrect probabilities and will forward queries to wrong peers in identical repeating queries.

The experiment works as follows. Each leaf node randomly selects 20 queries from the query data collected by our Gnutella crawler. The keywords of those 20 queries are seen as the resources shared by that leaf node. In this way, the leaf node should successfully answer the queries containing those keywords.

In the training phase, the "network" queries each of these resources. Then in the evaluation phase, these resources are re-queried and the *root* node receiving a repeated query does not immediately broadcast the query, but first tries to guess the neighboring peer managing the resource. For this, it selects the neighbor that has the highest rank and containing probability for the set of keywords contained in the query string. There is of course the possibility of making a wrong guess. If the prediction of the root node about which neighboring peer owns the resource is incorrect, or if the root declares that the resource is not contained in the network, or if it broadcasts the query to all the neighbors, the simulator considers this as a failure of the scheme and increases the error count. The test continues until all the queries in the collected data are used. We had a total of 6800 queries in the test.

Fig. 13(a) shows the performance (in terms of errors) of the scheme on exact-match queries. Although there is a sharp increase in the number of errors with a window size of 32 768, the absolute increase is small. Therefore, in a system using Array Radix 32 based mapping function, we cannot immediately declare that the window size of 32 768 is unsuitable for the parameters we have used.

Modification-error experiment: In this experiment we investigated how the protocol performs on close keywords and how the rank decision algorithm improves the protocol's decision making process. The test environment is similar to the exact match experiment, except that, in the evaluation phase the "network" modifies at least one keyword in the query by adding random characters at the end of each keyword. We do not use random locations while padding the random characters because this could produce completely irrelevant keywords and broadcasting may not become an error. The definition of the cases that are declared errors is the same as in the previous test.

While discussing the Array Radix 32 based mapping function, we mentioned that the sensitivity of the system to modified queries greatly depends on the window size used. This prediction

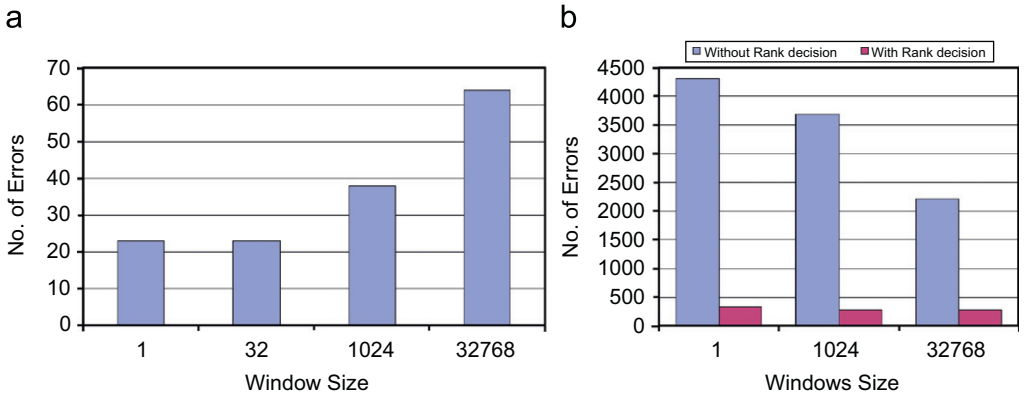

Fig. 13. (a) Number of errors made by the Array Radix 32 based mapping function in exact match experiment conducted with 6800 queries. (b) Radix 32 based mapping function modification behavior; experiment conducted with 6800 queries.

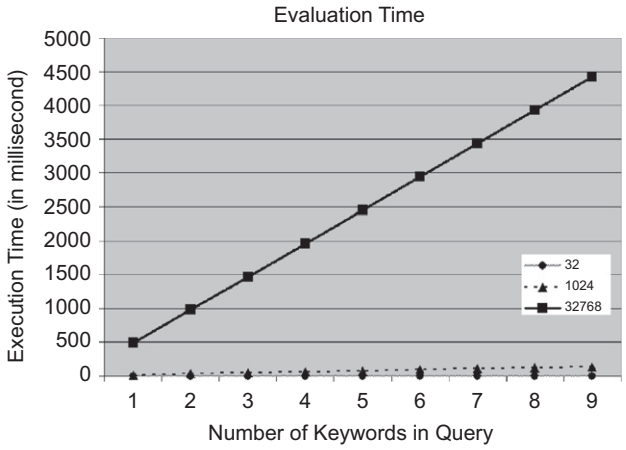

Fig. 14. Execution time of the evaluation phase with Radix 32 mapping function.

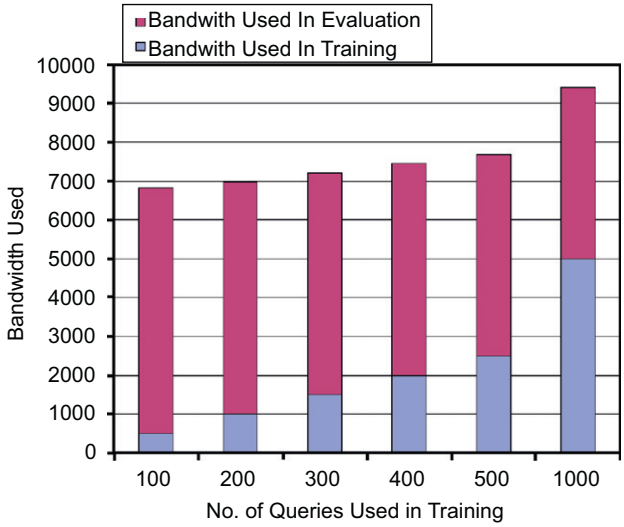

Fig. 15. Bandwidth used in Gnutella experiment conducted with 6800 queries; ~80% reduction to flooding requirements.

can easily be verified by the data shown in Fig. 13(b), since increasing the window size significantly decreases the number of errors. The high number of errors recorded when not using the rank decision algorithm is caused by the keywords that are shorter than the window's coverage area. The system declares the queries that include these keywords as "not trained"; but this is not the case. It is important to notice that the system has correctly predicted the routes of 2493 out of 6800 queries, although the window size was 1. This is because the keywords in these queries had more than five characters, so the Array Radix 32 mapping function simply ignored the padding, and made the correct decision. With the rank decision algorithm, we have even better results. There is a slight increase in the number of errors when a window size of 32 768 is used.

During the description of the evaluation algorithm, we stated that the running time of this algorithm greatly depends on the window size. To justify this, we measured the time it needs to evaluate queries having different number of keywords. Fig. 14 shows these time measurements taken using a PC with Pentium 4 processor at 2.6 GHz and 1 GB of RAM. Since in this experiment we measured the evaluation time of a query for a peer that has five neighbors, the window size becomes the dominating factor that determines the speed of the evaluation. With a window size of 32 768, the evaluation time increases rapidly, which causes queries to be routed slowly. However, a window size of 1024 does not greatly increase query processing time, indicating that it is suitable for the scheme. Thus, taking into account execution times and errors, we decided that 1024 is a good value for the window size. We use this value in the experiments we will discuss next.

4.5. Bandwidth usage and the effects of increasing the number of queries the system has trained with

The primary goal of the proposed scheme is to reduce the unnecessary bandwidth usage during propagation of queries in unstructured networks without sacrificing the query answer rate. Therefore it is important to know how much we can improve bandwidth efficiency and what its cost is.

Towards this goal, we first set a simulation experiment and measured the bandwidth usage of the Routing Learning scheme and compared it to pure broadcasting. The simulation model is based on the P2P network model in Fig. 12 which was described earlier. The "network" issues queries to the root peer in the order they are received by the crawler. The root peer trains itself with the first n queries it receives, and then starts the evaluation phase, in which the *toSendPeers* parameter is set to 1 (i.e., a query is sent to only one selected neighbor). We experimented with different values of n to see how training size affects the bandwidth used.

Fig. 15 shows the results of this test conducted with the Array Radix 32 based mapping function with a window size of 1024. If broadcasting were used, the bandwidth required, in terms of number of messages, would be 34 000 (5×6800). With Route

Learning scheme, however, only about 20% of this bandwidth is used in the system. The improvement over broadcasting is clear. As the system gets trained with more queries, the bandwidth wasted in the evaluation phase is reduced because the system declares fewer queries as "not trained". But the reduction becomes insignificant after a certain point. Therefore, we can conclude that training the system with large number of queries does not improve the system's overall performance enough to justify the cost of training. From Fig. 15, it can be seen that training the system even with 100 queries is adequate.

4.6. Experiments on Gnutella network

The simulation model on which we obtained the results is not a very realistic model of the Gnutella network (i.e., the frequent peer disconnections is not modeled) and we only used these simulations to adjust certain parameters of the Route Learning scheme. Simulating a P2P network is a stumbling block since there are many parameters to consider; using unrealistic approximations for one of these parameters may produce unrealistic results.

For this reason, we downloaded the source code of the Limewire program (version 4.12) and modified it to run Route Learning scheme on top of the Gnutella protocol. This allowed us to test the performance of the scheme on a real P2P network.

We present the architecture of the modified Limewire program in Fig. 16. It is important to note that the figure does not include the whole architecture of the Limewire program. We only depict the parts that are important. In this figure, the boxes represent the components (either software or entities that interact with the software on Internet) and the arrows represent the interaction between components. Just as the unmodified version, in our modified version of Limewire, the Gnutella protocol layer initiates connections and handles the routing of Gnutella messages. Unlike the unmodified version, the Gnutella layer forwards the query messages to the Route Learning layer. Both layers send their actions to the logging layer; thus for each query two actions are sent to the logging layer. Here an action can be to drop, broadcast, or send to neighbor (for the Gnutella layer the action is always broadcast). The logging layer writes to a database the time the query message is received, the action, the global unique identifier of the query message (GUID; in Gnutella a query hit message has the same GUID as the query message it is replying to, this way the reverse path of the query hit message can be found), the query text message, the number of neighbors the query is forwarded to, and the IP addresses of the neighbors the query is sent to. Then the query is given back to the Gnutella layer which broadcasts the query to the neighbors. For example, assume that the modified program is connected to 20 neighbors and Route Learning estimated that none of the neighbors can answer the query, so it should be dropped. Then, for this query two actions are sent to the logging layer, which are broadcast with 20 neighbors from the Gnutella layer and drop with 0 neighbors from the Route Learning layer; each action also contains the fields explained above. When a query hit message is received, the Gnutella layer forwards the query hit message to the logging layer and then routes the query hit message back to its destination just as a normal Gnutella client. For a query hit message, the logging layer finds the two associated actions from the database (the database is queried with GUID of the query hit message). The action logged from the Gnutella layer is updated with the number of replies the query hit message contains. However, for the action logged from the Route Learning layer, the logging layer checks if the action can produce the query hit (i.e., if the query hit message is received from a neighbor that Route Learning forwarded the query to). If the action is found to generate the query hit, then the action is updated with the number of replies received from the query hit message. The query text with the number of replies is then sent to the Route Learning layer so that it can update the feature space of the neighbor the query hit is received from. If, on the other hand, the action cannot be generated by Route Learning, no update is made. Returning back to our previous example, let us now assume that a query hit message from 14th neighbor is received, which contains three answers. The logging layer finds the associated actions from the database using the GUID of the query hit message; one for Gnutella and one for Route Learning. The action for Gnutella is updated with three replies. However, the action for Route Learning is not updated because Route Learning decided to drop the query, and for a dropped query a query hit message cannot be generated.

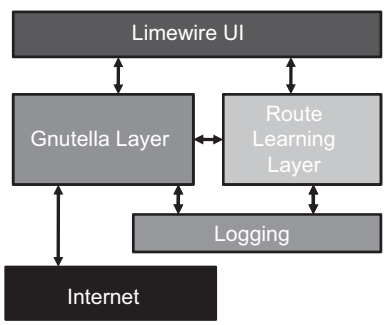

Fig. 16. The architecture of the modified Limewire program. The boxes represent the components and the arrows represent the interaction between components.

We also modified the Limewire program to operate in *ultrapeer* mode. In *leaf-node* mode, the program uses the query routing protocol in which a node sends a *hash* of its shares to the ultrapeer it is connected to. Then, ultrapeers do not forward most of the queries to the leafnodes, which reduces the number of queries and query reply messages greatly.

From the logs of our modified Limewire program, we use three metrics to compare Route Learning to Gnutella:

(1) Bandwidth used in terms of the number of neighbors a query is sent to (this metric is measured for each query to sum up the overall bandwidth usage).
(2) Answer rate which is calculated as

$$\frac{Q_{Route\ Learning}}{Q_{Gnutella}} \tag{5}$$

where $Q_{Route\ Learning}$ is the number of queries that got at least one reply with Route Learning, and $Q_{Gnutella}$ is the number of queries that got at least one reply with Gnutella.
(3) Answer quality calculated as

$$\frac{R_{Route\ Learning}}{R_{Gnutella}} \tag{6}$$

where $R_{Route\ Learning}$ is the total number of resources (i.e., answers/results) received in the query hits messages with Route Learning, $R_{Gnutella}$ is the total number of resources received in the query hits messages with Gnutella.

Here, we want to clarify the difference between the last two metrics with an example. Assume that for a query, flooding returned 10 query hits with a total of 30 resources included in these 10 query hits and Route Learning returned 4 query hits with a total of 15 resources included in these four query hits. Then, the answer rate is 1 since we got at least one query hit for both

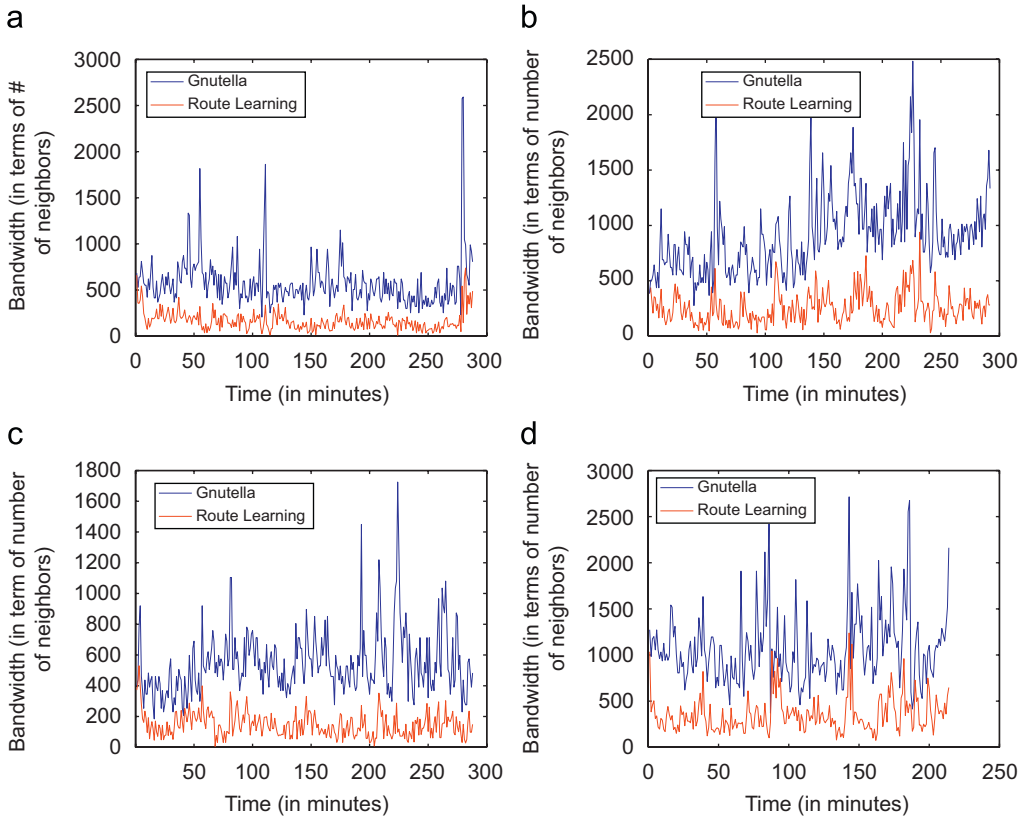

Fig. 17. The bandwidth used in terms of number of neighbors the query is sent to for 1000 queries collected from two different neighborhoods. The graphs on the left show the bandwidth used by Gnutella and the right shows the bandwidth used by Route Learning. Overall, Route Learning used only 28% of the bandwidth used by Gnutella.

Table 2
The results obtained from modified Limewire client

Neighborhood	No. of query messages	No. of query hit messages	Bandwidth used	Answer rate	Answer quality
1	7494	705	0.28	0.60	0.49
2	11 850	997	0.28	0.53	0.43
3	10 231	591	0.32	0.60	0.40
4	7144	739	0.26	0.61	0.35

Route Learning uses ~28% of the bandwidth used by Gnutella and is able to answer ~60% of the queries that are answered by Gnutella.

schemes, but the answer quality is 0.5 since we got less resources by using Route Learning.

We ran the modified program for 5 h with four different distinct neighbor sets (neighborhoods). It is important to note that, in *Ultrapeer* mode Limewire sets preferred number of connections to 32 and reserves nine of these connections for incoming connections. Therefore, the modified peer in the experiment has 23 neighbors. In this experiment, we set Route Learning to train with the first 100 queries and send the query to five neighbors with the highest rank after training is over. That is, if there are less than five neighbors at the highest rank, the query is only forwarded to these neighbors. It is also important to note that we have not modified the Gnutella messages; thus, the maximum TTL values on the queries were 4 (this limit is set by Gnutella peers; in our previous experiments with Gnutella network we have seen that Gnutella peers lower the TTL value to 4 when they receive a TTL larger than 4, Ciraci et al., 2005).

In Fig. 17, we show the bandwidth usage of Gnutella and Route Learning. From the four different neighborhoods, it can be clearly seen that Route Learning uses far less bandwidth than Gnutella. The plots show that for the first queries Route Learning uses the same bandwidth as Gnutella, which is because of the training phase. However, as soon as the training phase is over, the bandwidth usage of Route Learning drops below Gnutella. Overall, Route Learning used only 28% of the bandwidth used by Gnutella. The ratio of bandwidth used by Route Learning to that by Gnutella for each neighborhood is presented in Table 2.

Due to dropping and forwarding the query to a subset of neighbors, it is expected for Route Learning to reduce the number of query hit messages, which in turn also reduces the number of results (i.e., responses) returned to a query. With the answer-rate metric, we evaluate the rate of the number of query hits returned by Route Learning to the number of query hits by Gnutella. In Table 2, we show the answer rate for each neighborhood. Most of the time this rate is around 60%, which means that for the queries that generate answers, Route Learning is able correctly route the queries to the neighbors that contain the resources. For the 40% incorrectly routed queries, ~80% are caused due to dropping and ~20% are caused due to forwarding the query to neighbors. The main reason for this ~20% is the neighborhood changes that occur

at hop distances greater than 1. When a neighbor that can supply the answer leaves the network, Route Learning cannot update itself until another query arrives and the scheme decides to forward the query to that neighborhood. We found that the worst neighbor estimation occurred for neighborhood 2; thus, we concluded that the low answer rate for this neighborhood is caused by frequently disconnecting peers (that have hop distances greater than one hop). Besides the answer rate, we also compare the quality of the answers in Table 2. Because of selective forwarding, the answer quality of Route Learning scores between 35% and 50%. This answer quality is high enough to be acceptable, since a P2P network user usually does not use all of the many answers (i.e., responses) he receives to a query.

4.7. Effects of window size in Gnutella network

The window size experiment presented in Section 4.4 evaluated the impact of different window sizes on a fixed data set that is collected from the Gnutella network. However, it is important to evaluate the effects of this parameter with different query sets. Therefore, we repeat the window size experiment using the modified Gnutella client. For this experiment, we use the answer rate and bandwidth used metrics presented in the previous subsection. However, in this experiment the client evaluated the metrics for 10 000 queries and then disconnected from the network. We repeated this experiment for five different neighborhoods.

We first ran the modified client with window sizes of 1, 32, 1024 and 32 768. However, query evaluation started to take very long time with window size 32 768; therefore, we removed window size 32 768 from our tests.

Table 3 presents the results of this experiment. It is interesting to see that the scheme reaches the answer rate with window sizes 32 and 1024. With window size of 1024, however, the scheme reaches the same answer rate by using less bandwidth.

4.8. Comparison with other P2P routing schemes

We have run the modified LimeWire client with Route Learning, query caching (Yang and Garcia-Molina, 2002) and random-walk based protocols. In query caching scheme, the peers create caches containing (*Query*, *Answer from neighbor*) pairs (i.e., the query is not broken down to its keywords) using the query hit messages (Kalogeraki et al., 2002; Yang and Garcia-Molina, 2002). Upon the arrival of a query, a peer checks to see if the query can be answered from the cache. If so, the query message is only forwarded to the neighbor that can supply the answer; if not, the query is flooded. In the random-walk scheme, the peers generate the paths the queries will travel in a random manner (Lv et al., 2002). That is when a query is received, the peer generates the next hop randomly. We choose to compare random walk with Route Learning because we want to justify that the calculations made by Route Learning are in fact helpful. If random-walk scores better results, then there is no need to use Route Learning. Our random-walk implementation selects k distinct neighbors uniformly over the neighbors the peer has and we set k to be 60% of the neighbors. We have set the Route Learning's *toSendPeers* parameter to 1.

Table 3
The answer return rate and bandwidth usage of schemes for various window sizes

Window size	Answer rate (%)	Bandwidth used (%)
1	50	24
32	54	21
1024	54	18

Table 4
The answer return rate, the answer quality, bandwidth usage, and storage requirements of Route Learning, random walk and query caching

Protocol	Answer rate (%)	Bandwidth used (%)	Storage used (MB)
Route Learning	64	~26	1
Random walk	43	60	0
Query caching	88	97	0.009

To compare the schemes, we take into account the same metrics as the ones used in the previous subsection. However, this time we also measure the storage (i.e., memory) used by the schemes. For Route Learning, the *storage used* metric measures the size of the feature space, and for query caching it measures the size of the cache (which we implemented as a hashtable that stores an integer hash key for the query and the next hop as another integer). Since the random-walk based scheme does not keep track of query and query hits, it does not need a storage and thus, the storage used for this scheme is 0. We again used 10 000 queries and five different (distinct) neighborhoods for this experiment. The modified LimeWire client collected on the average 344 query hit messages using the Gnutella protocol. Thus, we take this number as the total number of query hits that can be returned to these queries and compare the schemes with respect to Gnutella.

In Table 4, the bandwidth usage and the answer rate of each protocol is shown. Route Learning was able to return 64% of the answers returned by flooding. Route Learning attained this answer rate by using only 26% of the bandwidth used by flooding (i.e., normal Gnutella protocol). Random-walk based protocol achieved the least answer rate: 43%. The query caching method achieved a good answer rate, but its bandwidth usage is very high, close to flooding.

The experiment shows that on the average Route Learning used 1 MB of memory, which is too high compared to the query caching scheme. This is because the implementation of Route Learning uses storage (which is a hashtable) for each neighbor and in this hashtable each keyword in a query is indexed according to the Radix 32 value. With this storage, however, Route Learning is able to estimate the next hops of the queries (either dropping or forwarding to a selected number of neighbors) resulting in much less bandwidth usage compared the pure flooding (Gnutella). Query caching, on the other hand, uses one hashtable to store the queries as a whole (i.e. the query is not divided to its keywords). However, storing the queries as a whole causes the query caching scheme to flood the queries even in minor changes in the queries (which causes too many cache misses) resulting in high bandwidth usage.

The memory usage of Route Learning on a peer depends on the number of neighbors the peer has, the size of the hashtable used (as a feature space), and the keyword length limit. Our results on the feature space size (Section 4.2) show that a hashtable of size 32^3 can be used, and when the table gets loaded, this size can be increased to 32^4. A major factor that causes the tables to become loaded is the keyword length limit. If this limit is set to a small number, then more keywords would map to the same feature space cell and less memory would be used. This, however, may cause queries to be routed to the neighbors that cannot supply an answer to the query.

The number of keywords that map to the same cell greatly depends on the query profile of the neighborhood the peer is connected to (e.g., query keywords can differ slightly). We can use this to adjust the keyword limit parameter to reduce memory

usage. Assuming the keyword length limit is set to l, for a keyword w from a query, the feature space lookup operation can calculate the distance of the keyword and the cell as $radix_{32}(w, l+d) - radix_{32}(w, l)$ (since the cell look up will use the formula $radix_{32}(w, l)$). If this distance is greater than 32^{-d}, the keywords that differ in $d+1$ or more characters map to the same cell, and therefore the keyword limit has to be increased. Here, the value of d can be set to a lower value (that is greater than zero) for peers with more memory (e.g., if d is set to 1, then the keywords with two character differences may trigger an increase in l).

4.9. Hops performance experiment

The experiments conducted so far does not give detailed information on how Route Learning succeeds at finding resources at hop distances greater than 1. To investigate this, we constructed a simulation environment using tree topology; four children per node with four levels that makes a total of 85 nodes (we used four levels because in our previous study of the Gnutella network (Ciraci et al., 2005), we found out that 91% of the queries submitted to the network have a TTL value of 4). In this simulation model, only the root of the tree is allowed to send queries. We selected such a simulation topology to find more accurately the query hit generation rate of Route Learning (that is the percentage of query hits generated by Route Learning) on different hop values. Besides the topology, we tried to keep everything as similar to Gnutella network as possible. We used the query, query hit, and pong messages collected by our crawler. Each peer in the simulation acts as a peer collected in the pong messages; that is they share the same number of resources as the peers discovered by the pong messages in real Gnutella network. This scenario resembles to real life in terms of the number of items the peers share. For the names of resources, we use the replies given in query hit messages, and for queries we use the collected query messages. During the simulation, the root peer simply re-plays the collected query messages.

We tested and compared four query routing techniques; namely, flooding, random-walk (Rand), Route Learning (RL) and Vote Route Learning (VRL). To induce comparable results between Route Learning and random-walk based routing, we allowed both schemes to send the query to only one neighbor (note that broadcasting and dropping queries are still valid decisions for Route Learning). We allowed RL and VRL to train with 100 queries. The root node queries the network by randomly selecting 1000 queries from the set of queries collected from Gnutella network. However, in order to better understand the query hit generation and answer rate, we gave 75% chance of being sent to the queries that have a matching query hit message. This causes the simulation to generate more query hit messages than Gnutella network. We programmed the simulation this way, because with more query hit messages we can better see and compare the bandwidth usage and answer rate of the schemes. We use the flooding to determine the number of query hits generated and the maximum bandwidth used by each hop.

In Fig. 18, we present the bandwidth used by peers at different hop levels. For example, for the hop level 0, there is only the root node, thus the plot shows the bandwidth used by this node. We see that for each hop level, Route Learning and Vote Route Learning bandwidth usage is always between flooding and random walk. To be more precise, Route Learning used 52%, 44%, 42% of the bandwidth used by flooding and VRL used 68%, 55%, 52% for the respective hop values. We see that for this experiment bandwidth usage of Route Learning scheme is higher than the usage in Gnutella experiment. This high bandwidth usage is also seen for flooding and random-walk

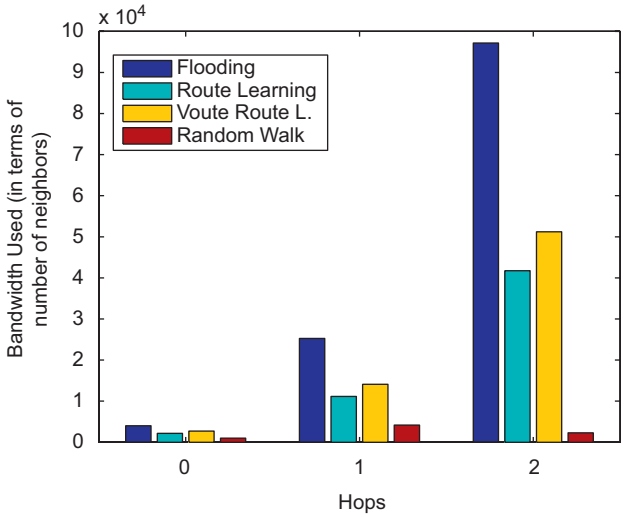

Fig. 18. The bandwidth used by peers at different hops values. Overall Route Learning and Vote Route Learning used ~40% of the bandwidth used by flooding.

Table 5
The percentage of query hit returned to the query hits returned when flooding is used

Hops	No. of peers	No. of query hits	RL QHR (%)	VRL QHR (%)	Rand QHR (%)
1	4	73	70	78	58
2	16	234	56	62	17
3	64	959	48	58	0.2

Route Learning and Vote Route Learning was able keep up with flooding as the hops distance increases.

Table 6
The answer rate of the schemes

Scheme	Answer rate (%)	Answer quality (%)
Route Learning	54	49
Vote Route Learning	65	58
Rand walk	8	7

We see that Route Learning got a reply to 54% of the replies returned by flooding by using 45% of the bandwidth.

schemes and it is caused by the many query hit messages being grouped at one node.

In Table 5, we present the percentage of query hit messages returned by the schemes. As discussed before, in the tree topology we first run the flooding scheme to count the number of query hit messages it returns. Then, we run other schemes and count the query hit messages they return. Finally, we find the Query Hit Rate (QHR) by dividing the query hit messages returned by other schemes to the ones returned by flooding and multiplying the result by 100. From the table, we see that for the first hop the ratio is greatest. This is because for that hop value there is less nodes, so the probability of finding a query hit message is higher. However, as the hops value increases this probability decreases rapidly. This is clearly the cause of low rates for random-walk based schemes. When we look at the QHR values of Route Learning and Vote Route Learning, we see that there is a decrease in the ratio; however, not as much as the decrease seen in the random-walk scheme. Thus, we can clearly say that both schemes' decisions help in finding the neighbor that contains the resource that answers the query.

Finally, in Table 6 we present the answer rate (Eq. (5)) and answer quality (Eq. (6)) of the schemes. In this experiment, Route Learning was able to find half of the answers found by flooding, and Route Learning achieved this rate by using ~45% of the bandwidth. We see that Route Learning scored a less answer rate than the experiments conducted in Gnutella network in this experiment. This is because of the high matching rate of query messages. Vote Route Learning, on the other hand, scored an answer rate higher than Route Learning and achieved this rate by using ~10% more bandwidth than Route Learning.

5. Related work

There is a substantial body of work on reducing query traffic overhead in unstructured P2P networks. In this section, we briefly summarize some of these works, which are most relevant to our study.

Queries in a Gnutella network exhibit a significant amount of locality: queries with the same or similar search strings are common (Markatos, 2002). Based on this observation, a caching scheme is proposed in which every peer stores query strings, TTL values, and the corresponding query hit messages while the peer is relaying Gnutella queries and query hit messages. Then, upon receiving a query message, a peer scans its own stored TTL and query string pairs for a match to the query string, and the TTL value stored in the received query. If such an entry is found, the peer returns the corresponding query hit (i.e., the answer) as a reply to the query. Our study is similar in that it also makes use of the locality in queries. However, rather than indexing the query string as a whole, we separately index each of the keywords seen in a query. This helps us to cope with changes in user-submitted query strings. Another difference between our work and (Markatos, 2002) is that the peers in our scheme do not answer queries on behalf of their neighbors or any other peer. They estimate the neighbors most likely to reply and forward the query messages only to them. In other words, we do not cache the query hit messages at intermediate peers.

Yang and Garcia-Molina (2002) propose three techniques to improve the search efficiency in their studies. In the "localized indices" technique the peers maintain an index of the shares their neighbors have; thus, when a peer receives a query it answers the query from this index and does not forward the query to its neighbors. Upon joining the network a peer sends the index of its shares to its neighbors and the neighbors receiving this index send their indices to the newly joined peer. This technique is different than the Route Learning scheme, because in Route Learning peers do not send an index of their shares; peers find out the resources reachable through its neighbors by observing the query and query hit messages. Another technique proposed in that study is the "iterative deepening", where a query is first sent to the network with TTL x. After a certain period of time, if the initiator of the query is not satisfied with the results, it sends a "Resend" message with TTL $y (y > x)$ and the receiving nodes at TTL x resend the same query with TTL $y - x$, which starts a second iteration for the query. There is no iterative deepening in Route Learning; queries are processed by the receiving peers and when the TTL of the query expires, the query is not forwarded anymore. However, such an iterative deepening mechanism can easily be incorporated to Route Learning to increase the answer rate and querying time of the search. Lastly, authors of the study propose "directed breadth first search" technique which tries to forward the queries to the neighbors that are highly likely to supply an answer. This technique and Route Learning share the same aim; however, the statistics collected by Route Learning is more complex than the statistics proposed by those authors. Therefore, Route Learning can be seen as an improvement over this technique. The authors propose to send the query to the neighbors that supplied the most query hits. Route Learning on the hand keeps track of the query to query hit ratio for every keyword in the queries a peer receives. Furthermore, this sophisticated statistics collection allows Route Learning to drop a query to save further bandwidth, in case none of the neighbors can answer the query.

Kalogeraki et al. (2002) propose a routing technique that also tries to find the highly likely neighbors that can supply the answer. In this scheme, peers hold a list of (Q, N) pairs, where N is the node that supplied answers to query Q. Similar to Route Learning, the nodes form this list by observing query and query hit messages. In this scheme, the nodes also export this list to their neighbors; in Route Learning, the peers do not distribute the information they gathered about the network to save more bandwidth. Upon arrival of a query message, a peer first calculates the distance between the received query and the queries maintained in the list, then forwards the query to the nodes that supplied answers to the most similar queries. We can say that Route Learning calculates the distance between a received query and the queries it has observed with the rank decision algorithm. However, Route Learning uses a mapping function that maps similar keywords to close points in the feature space. This way, Route Learning is able to cope with small differences in the keywords themselves. Furthermore, it is able to drop a query when it decides none of the neighbors can supply an answer.

Tsoumakos and Roussopoulos (2003) also propose to store index values for each query message a peer requests or forwards. An index holds the neighbor a query is forwarded to with the relative probability of this neighbor to be chosen as a next hop for the other queries. Initially, this probability is equal for all neighbors, thus the neighbor selection is random. The query initiator selects n of its neighbors and initiates the walkers. That is, the query is forwarded to those neighbors. The neighbors receiving the query forward the query to only one of their neighbors if they do not find a matching resource at their shares. If the walker is terminated (i.e., the TTL value expires) without a query hit, then the peers where the walker is terminated send an "update" message with the reverse path of the query. The peers receiving the update message lower the relative probabilities stored at their index. This scheme also tries to find the neighbors that are highly likely to answer a query as Route Learning does. However, Route Learning approach allows a peer to cope with minor differences in keywords of a query (e.g., "Matrix" and "MatrixDVD"). Also, Route Learning uses random guess only to find out if there has been a change in the network, and this guess is only initiated at certain intervals. If Route Learning does not know the query, it broadcasts the query to find out and learn which neighbors can supply an answer.

Applying a distributed hash table (DHT) structure to P2P networks is another approach that improves the scalability and the exact-match hit rate (Stoica et al., 2001; Ratnasamy et al., 2001). The problem with a hash table based structure is that it can only support exact match queries. Some studies used DHT-like schemes in Gnutella as well, where peers index their shares in a binary hash table and send it to their neighbors at certain time intervals (Rohrs,). An *n-gram*-based technique to allow fuzzy queries is described in Witten et al. (1999). This idea is borrowed by Harren et al. (2002) to bring DHTs to P2P systems with a complex query structure. These approaches are significantly different from ours in that we do not require peers to hash their resources and distribute them to their neighbors, a process which causes extra messaging overhead.

The routing method proposed in Menasce and Kanchanapalli (2002) may seem related to Route Learning since both schemes use a probabilistic approach. However, the protocol in Menasce

and Kanchanapalli (2002) makes use of caches and it forwards queries to neighbors based on "broadcast probability". Route Learning, on the contrary, does not make use of caches and forwards queries to neighbors based on the knowledge collected about them.

Most of the *learning* based routing protocols focus on clustering peers that have similar interests. In the scheme proposed in Ng et al. (2003), peers associate to the network using two connection methods. The first connection method, *random connection*, is the method used by Gnutella. A peer chooses the neighbors to connect to by availability, i.e. if a neighbor can accept the newcomer, then the connection is established. In the second connection method, *attractive connection*, the newcomer establishes a connection according to the answers it has received. If a peer X has supplied most of the answers, then the newcomer establishes a connection with the peer X. This way, most of the queries submitted will be answered by one-hop neighbors. The main difference between our work and the works presented in Ng et al. (2003) is that, in our system peers do not change connections. Changing neighboring connections is not a very attractive solution since P2P networks are highly dynamic and nodes frequently connect to or disconnect from the network. Another difference is that, Route Learning has methods to cope with changes in keywords; it does not rely only on keyword repetitions but it also uses similarity between keywords to further reduce the querying overhead. Finally, Route Learning can dynamically adjust to the changes in topology of a P2P network that arise from new connections or disconnections.

Joseph (2002) proposes a scheme, called "NeuroGrid", where peers associate keywords with neighbors. Similar to Route Learning, this association contains two numbers: (1) the number of times the keyword is seen in queries; (2) the number of times an answer is supplied to a query which results in a download. A query is forwarded to a number of peers that have the highest ratio between these two numbers. In NeuroGrid, peers can also update connections according to the replies they receive so that they can become neighbors with peers that answer their queries. The second number used in estimating the next hop for a query requires feedback messages to be given by peers in addition to query and query hit messages (a feedback message is sent for example when a peers starts downloading a resource), which in turn requires new messages to be added to the Gnutella protocol. Route Learning, on the other hand, does not require any extensions to the Gnutella protocol. It builds the knowledge based on only query and query hit messages. Besides this, NeuroGrid does not have any mechanisms to cope with changes in the keywords of the queries and Route Learning's estimation mechanism can cope with small differences in keywords.

The study presented in Handurukande et al. (2004) tries to exploit semantic proximity betweenpeers. Semantic relationship between peers is captured via one of two methods: *LRU* or *History*. In LRU, an uploader peer is added to the top of the list of semantic neighbors of a peer. In History method, a peer keeps track of the uploaders and builds a list of semantic neighbors that contains the best uploaders. The queries are then forwarded to semantics neighbors where answers are highly likely to come. The difference of this study from ours is that we try to capture the semantic relationship by looking at the keywords contained in queries, and we are not using flooding.

Some other schemes have been proposed to reduce the querying overhead through clustering of peers. In those schemes, a query is first routed directly to a related cluster and then to the peers in that cluster. The schemes differ from each other in the way they build the clusters. In Crespo et al. (2002), peers form multiple overlay networks by classifying the content they share. Peers with similar content are placed into the same cluster. In the clustering scheme presented in Guo et al. (2005), a group of peers that supply most of the answers to queries form a cluster. To further improve the search performance, a newly joined peer marks the source peers that send query hits. Then the new peer asks source peers to send the index of all the files they share. This allows the new peer to know about the files of peers with similar interests which enable a query to be routed directly to those peers. Another clustering method is described in Zhu et al. (2005). The method is based on an information retrieval technique called VSM (vector space model). Node vectors are derived for peers depending on the files they share, and those vectors are used to form groups of semantically related peers. A query is first routed directly to a related group, and then flooded inside that group. Datta et al. (2006) state the importance of using distributed data mining in P2P networks. They investigate the use of exact and approximate P2P algorithms for K-means clustering. The clustering schemes described here are all different from our work, since the routing scheme we propose does not involve clustering of peers.

In current P2P systems there is no strict stopping criteria for queries. Some P2P systems start timer when a query is sent and declare the query as finished when the timer expires. However, after declaring a query as finished, some peers containing the sources that can answer this query with better resources can join the network. Tas and Tas (2006) address this problem by applying the "*Search Theory with Learning Approach*" to P2P domain. In their proposal, the searching peer calculates the download time and sends the query. The query answers include the times the peers can supply the resource. The query is declared to be finished when at least one peer with supply time smaller than the download time calculated by the searcher is found. Since the distribution of download times for all resources over all peers cannot be found, Tas et al. employ a Bayesian learning system, where the searching peer learns the distribution of download times using the results of the queries. This study is similar to our Route Learning scheme, since we also employ a learning based algorithm. Though, our aim in using the learning algorithm is to anticipate the peers that can answer a given query.

6. Conclusion

In this paper, we present a new semantic routing scheme which we call *Route Learning* for P2P networks. This scheme is an adaptation of a classification problem to unstructured P2P networks aiming to reduce query overhead. In Route Learning, peers gradually build knowledge about their environment, so that they can predict which neighbors can supply an answer. This process reduces the amount of bandwidth required in querying. The main difference between the scheme presented and a classification problem is that our scheme tries continuously to adapt its knowledge in order to cope with frequent changes in the neighborhood.

Route Learning's efficiency in predicting the routes greatly depends on keyword repetition. It was already proven through the analysis of P2P networks that keywords submitted to a P2P network follow a Zipf distribution. Therefore, a high number of repetitions of some keywords can be expected. Route Learning also considers the similarity between keywords and tries to relate the keywords, aiming to save bandwidth. We employ a very simple similarity measure in our evaluation, the distance between keywords when they are mapped to the feature space. The experimental results show that Route Learning usually relates close keywords with each other and forwards the query to only one neighbor. The results also show that Route Learning can save significant amount of bandwidth.

References

Androutsellis-Theotokis S, Spinellis D. A survey of peer-to-peer file sharing technologies. Electronic Trading Research Unit (ELTRUN), Athens University of Economics and Business; 2002.

Ciraci S, Korpeoglu I, Ulusoy O. Characterizing Gnutella network properties for peer-to-peer network simulation. In: Yolum P, Gungor T, Gurgen F, Ozturan C, editors. Computer and information sciences—ISCIS 2005. Lecture notes in computer science, vol. 3733. Berlin: Springer; 2005. p. 274–83.

Crespo A, Molina, HG. Semantic overlay networks for P2P systems. Technical Report, Computer Science Department, Stanford University; 2002.

Datta S, Bhaduri K, Giannella C, Wolff R, Kargupta H. Distributed data mining in peer-to-peer networks. IEEE Internet Comput 2006;10(4).

Gnutella protocol v0.6. URL: ⟨http://rfc-gnutella.sourceforge.net/developer/testing/index.html⟩.

Guo L, Jiang S, Xiao L, Zhang X. Fast and low-cost search schemes by exploiting localities in P2P networks. J Parallel Distributed Comput 2005;65(6).

Handurukande, SB, Kermarrec A-M, Le Fessant F, Massoulie L. Exploiting semantic clustering in the eDonkey P2P network. In: EW11: proceedings of the 11th workshop on ACM SIGOPS european workshop: beyond the PC, Leuven, Belgium. ACM Press; 2004.

Harren M, Hellerstein JM, Huebsch R, Loo BT, Shenker S, Stoica I. Complex queries in DHT-based peer-to-peer networks. In: Revised papers from the first international workshop on peer-to-peer systems. London, UK: Springer; 2002. p. 242–59.

Joseph S. NeuroGrid: semantically routing queries in peer-to-peer networks. In: Revised papers from the networking workshops on web engineering and peer-to-peer computing. London, UK: Springer; 2002. p. 202–14.

Joseph S. P2P MetaData search layers. In: Second international workshop on agents and peer-to-peer computing, 2003.

Kalogeraki V, Gunopulos D, Zeinalipour-Yazti D. A local search mechanism for peer-to-peer networks. In: Proceedings of the eleventh international conference on information and knowledge management. New York: ACM Press; 2002. p. 300–7.

Karakaya M, Korpeoglu I, Ulusoy O. GnuSim: a general purpose simulator for Gnutella and unstructured P2P networks. Technical Report, Department of Computer Engineering, Bilkent University; 2005.

LimeWire Gnutella client. URL: ⟨http://www.limewire.com/⟩.

Lv Q, Cao P, Cohen E, Li K, Shenker S. Search and replication in unstructured peer-to-peer networks. In: Proceedings of the 16th international conference on supercomputing. New York: ACM; 2002.

Markatos EP. Tracing a large-scale peer to peer system: an hour in the life of Gnutella. In: Proceedings of the 2nd IEEE/ACM international symposium on cluster computing and the grid, Washington, DC, USA, 2002.

Menasce DA, Kanchanapalli L. Probabilistic scalable P2P resource location services. In: SIGMETRICS performance evaluation review, vol. 30, no. 2. New York: ACM Press; 2002. p. 48–58.

Ng CH, Sia KC, Chan C-H. Advanced peer clustering and firework query model in the peer-to-peer networks. WWW 2003 (Posters); 2003.

Ratnasamy S, Francis P, Handley M, Karp R, Schenker S. A Scalable content addressable network. In: Proceedings of the ACM SIGCOMM conference, 2001. p. 161–72.

Rohrs C. Query routing for the Gnutella network. URL: ⟨http://rfc-gnutella.sourceforge.net/src/qrp.htm⟩.

Saroiu S, Gummadi, PK, Gribble SD. A measurement study of peer-to-peer file sharing systems. In: Proceedings of multimedia computing and networking (MMCN '02), San Jose, CA, USA, January, 2002.

Stoica I, Morris R, Karger D, Kaashoek F, Balakrishnan H. Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the ACM SIGCOMM conference, 2001. p. 149–60.

Tas NC, Tas BKO. A search theoretical approach to P2P networks: analysis of learning. In: Proceedings of the advanced international conference on telecommunications and international conference on internet and web applications and services. IEEE Computer Society; 2006.

Tsoumakos D, Roussopoulos N. Adaptive probabilistic search for peer-to-peer networks. In: Proceedings of the 3rd international conference on peer-to-peer computing, 2003.

Witten IH, Moffat A, Bell TC. Managing gigabytes: compressing and indexing documents and images. Los Altos, CA: Morgan Kaufmann; 1999.

Witten IH, Moffat A, Bell TC. Pattern classification. New York: Wiley Interscience; 2000.

Yang B, Garcia-Molina H. Improving search in peer-to-peer networks. In: Proceedings of the 22nd international conference on distributed computing systems (ICDCS'02), Washington, DC, USA, 2002.

Zhu Y, Yang X, Hu Y. Making search efficient on Gnutella-like P2P systems. In: Proceedings of the 19th IEEE international parallel and distributed processing symposium (IPDPS'05), 2005.