A Scalable Search Algorithm for Unstructured Peer-to-Peer Networks

Amit Gud Masaaki Mizuno Daniel Andresen
Department of Computing and Information Sciences
Kansas State University
234 Nichols Hall, Manhattan KS, 66506, USA

Abstract

Scalability in a peer-to-peer network is a challenging problem. Unstructured peer-to-peer networks inherently lack scalability, and structured networks are inefficient for a high churn rate. In this paper, we present a scalable search algorithm for a decentralized unstructured peer-to-peer network using a method to dynamically determine the number of nodes to forward a query to at once. The decision is based on the degree to which each neighbor has contributed to previous successful searches. The algorithm automatically creates a spanning graph of the high traffic links. Once a stable spanning graph is created, a query tends to travel along the edges of the spanning graph. This way, the number of hops required for a search is roughly bound by the diameter of the spanning graph. The simulation shows that our algorithm demonstrates significantly better performance in terms of the number of messages generated and hops required for a search over other popular algorithms.

Keywords: *P2P, resource discovery algorithms, distributed systems performance*

1. Introduction

Over the past decade peer-to-peer (P2P) network has shown rapid and significant growth along with the growing Internet traffic. P2P networks currently take a major share of the Internet bandwidth. The popularity of the P2P network has been triggered by the need of file-sharing over the network, which made the study of performance, and scalability of such networks an interesting area of research.

P2P networks are classified based on the topology that they create and their degree of centralization. A *centralized* P2P network is the one which has a dedicated central server for maintaining the information of the files shared along with their locations. This architecture is vulnerable to single point of failure. Napster is an example of such a network [9].

Decentralized P2P networks overcome this drawback by not having single central server. In a decentralized P2P network, no node has any special capabilities and all the nodes behave in the same manner. Decentralized networks are further classified as *structured* and *unstructured*. In structured networks, the object[1] placement is deterministic which helps in satisfying a query[2] easily. Such structured networks are confined only to the research literature and are practically non-existing in the real networks [3], [6] and [7] are examples of structured network.

In decentralized unstructured network, the precise location of an object is not known. Since this topology is more in use today there is a need to improve on the search mechanism of this architecture. Gnutella is an example of such a network. Further details of the classification of P2P can be found in [3].

A decentralized unstructured P2P overlay[3] network is built on top of the existing network topology. This is similar to building a connected graph, wherein every node knows only about a limited set of other nodes on the graph. In such networks, the precise object location is not known. A search technique has to be deployed in order to inquire at each node present in the network about the presence of the object.

Early search mechanisms primarily used flooding [1] or k-random walk [2] algorithms. In the flooding approach, each node propagates the query to all its neighbors. On a receipt of a query, the node searches in its local repository. If the object is found, it informs the query originator and further search in that path terminate. If not, the node further forwards the query to all its neighbors. To avoid duplicate query servicing, each node maintains a list of all the queries that it has serviced. As the number of nodes scanned per hop

1 The terms *object* and *file* are used interchangeably throughout the paper.

2 Throughout this paper, *query* signifies an object / file search in the P2P network and the terms *query* and *search* are used interchangeably.

3 An *overlay network* is built on top of the existing real network by logically connecting the involved nodes.

is high, the search time (i.e., the number of hops/query) required is reduced.

The flooding method generates a large amount of network traffic. To overcome this problem, random walk algorithms are often used. In the random walk search algorithm, the number of nodes to which a query is forwarded by each node N_i is limited to a number k [2], where value of k is between 1 and NB_i, where NB_i is the number of known neighbors of node N_i. These k nodes are selected randomly. Though highly random and non-deterministic, this approach is demonstrated to work more efficiently than flooding [4], [8].

This technique uses feedback from the previous searches which helped probabilistically guide the future ones. The grade of a node is determined by this feedback information, which indicates the possible success rate.

This paper presents an algorithm for improving the object search mechanism in decentralized unstructured P2P network by using heuristics based on the configuration, performance and historical contribution of a node with respect to successful object searches.

Our search method, *Activity Based Search (ABS)*, improves the previous probabilistic search techniques by making the k value adaptive[4]. Under ABS, each node keeps a list of known neighbor nodes sorted by the degree to which each neighbor has contributed to the previous successful search (such degree values are called "activity levels"). Every query is sent to the top k_d nodes. A distinctive feature of our algorithm is that, unlike [11], a node does not consider the exact semantic preference when choosing the nodes to forward a query to. A node simply considers the previous success ratio, regardless of the objects being searched, to determine a set of nodes to forward a query to. Thus, its scalability is better than that of APS in terms of the storage size.

Simulations show that our algorithm automatically creates a spanning graph of the nodes fairly quickly (see Figure 1), where the nodes in the graph represents the nodes in the network and the edges in the graph represents a subset of the links in the network. The communication traffic on the edges in the spanning graph is substantially higher than that on links not in the spanning graph. Efficient creation of such a spanning graph is realized by the dynamically adapting value of k_d and selecting nodes with the highest activity levels to forward a query to.

The performance simulation demonstrates that our algorithm demonstrates significantly better performance than k-random walk and 1-random walk. Our messages-per-query is much less than that of k-random walk, yet our number of hops to reach the target nodes is much less than that of 1-random walk and even slightly less than k-random walk.

The organization of the paper is as follows: related previous work is briefly described in Section 2, our algorithm is presented in detail in Section 3, and the experimental results are presented in Section 4. We conclude in Section 5.

4 To indicate that our k value dynamically changes, we use notation k_d to denote our k value.

2. Previous Work

Random walk search algorithms have demonstrated to perform well in unstructured decentralized P2P networks [4]. However there has been attempts to further improve the basic random walk algorithms. These works try to improve the random walk algorithms either by using feedback from the previous search results [5], [10], [8] or by constructing different overlays depicting node-level semantics, e.g. temporal and geographical localities [11], [12], [13], [14].

The feedback mechanism is used in building a profile of the known neighboring nodes. When a query has to be forwarded further, the node uses these profiles to rank the neighboring nodes. The feedback information is maintained in the form of grade for each node. This grade is then increased or decreased depending upon the result of the object search passing through the node. Thus both learning and unlearning characteristics are exhibited by the nodes. The query is forwarded to top-performing k nodes, i.e. the nodes having top k grades.

Dimitrios et. al. [5] discusses such a method. In this, only k walks are generated, i.e., the query originator forwards the query to k nodes in the first level of search. If the object is not found in the local repository, the node further forwards the query to only one top performing node. So from the second level onwards the query is forwarded only to one node. In-effect this limits the search within the k paths. Furthermore, the value of k is fixed at each node and is not adaptive to dynamically changing network sizes. The k value of our ABS algorithm, denoted the k_d value, is dynamically adjusted to improve search performance.

[11], [13] and [14] present a technique to make the network scalable by exploiting the common interests of the peer nodes. Their protocol has demonstrated to improve the search results. However, they emphasize creation and maintenance of the semantic overlays, rather than enhancing the scalability of the search. Our approach dynamically adjusts the degree of flooding to make the search more scalable.

Chawathe et. al. [15] suggests a topology-aware P2P system, which dynamically adapts to the changing network topology. This involves balancing the degrees of a node proportional to its capacity as defined by its access bandwidth, processing power and disk speed. Search mechanism, however, does not attempt to scale with the network size and the number of nodes involved in an object search is static. node

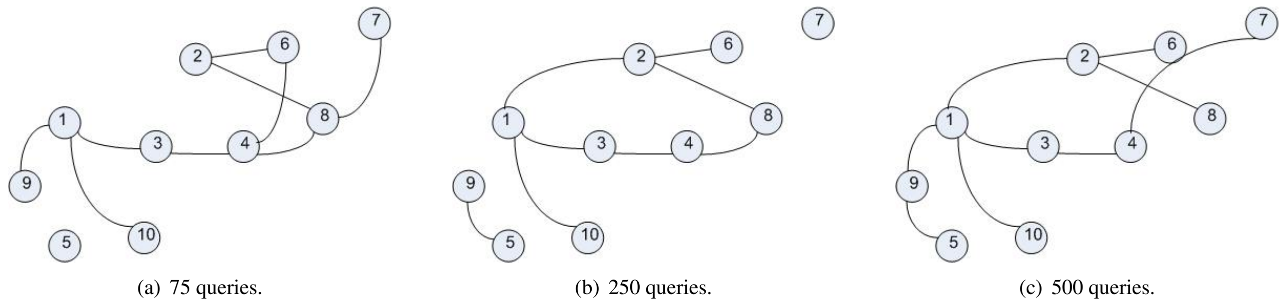


Figure 1. Connectivity graph for 10 nodes. Note similarity of overall patterns between graphs, indicating each node has formed a short list of "preferred partner," essentially forming an implicit spanning graph. Edges shown have the heaviest (top 25%) activity levels.

does not weigh much, as much as the perceived track record of a node.

3. The Activity Based Walker algorithm

When a node, say N_I, tries to search a file F_T in the set of nodes U, N_I initiates a query message. The query message is passed among nodes. When a node which owns F_T, say node N_T, receives the query, it informs N_I by initiating a "success" message. In the above scenario, N_I, F_T, N_T are called the *initiator* of the query, the target file, and a *target node*, respectively.

Each node N_i maintains a set of files (names or references) in data structure $FILES_i$. When a node N_i receives a query from a node N_j, N_j is called the *sender* of N_i. Upon a receipt of the query, N_i checks whether it has the target file by referring to $FILES_i$ or whether it knows a node which owns the target file (the later case is discussed below). If so, it initiates a "success" message; otherwise, it forwards the query to a set of nodes that N_i knows. Our protocol has salient features in how to determine a set of nodes to forward a query to.

To determine a set of nodes to send a query to, each node N_i maintains, in data structure $NBRS_i$, a list of neighbors, that is, the nodes that N_i knows. $NBRS_i$ is a finite list of tuples [node N_k (its address such as IP), its activity level (denoted AL_k, discussed later), and $FILES_k$], ordered by the activity levels. Since $NBRS_i$ is finite, when a new tuple is added, a existing tuple with the lowest activity level may have to be dropped from the list.

The activity level AL_j of node N_j in $NBRS_i$ is a value calculated based on the success ratio in the previous searches; that is, the ratio at which N_j was on the paths to the target nodes in the previous searches, regardless of the target files. AL_j in $NBRS_i$ is increased by some predefined value, denoted AL_{INC1}, if the query sent to N_j leads to a target node. In addition, if N_k is a target node of the search, AL_k of N_k's tuple in $NBRS_i$ is increased by some predefined value AL_{INC2}; or if N_k's tuple is not in $NBRS_i$, a new tuple [N_k, AL_{INIT}, $FILES_k$] is added to $NBRS_i$, where AL_{INIT} is a predefined initial activity level. All activity levels of $NBRS_i$ are between 0 and 1, normalized with respect to the highest activity level in $NBRS_i$.

When initiating a new query or upon a receipt of a query, the number of nodes to which node N_i forwards a query at once dynamically changes by the following three factors. (1) a predefined value *k-threshold*, which is the lower bound on the activity levels of nodes to which the query is forwarded, (2) a predefined value *k-min*, which is the lower bound on the number of nodes to forward the query to, and (3) the number of nodes in $NBRS_i$ that the query has not yet visited. Since each query contains a list of nodes that it has already visited, it is easily known whether the query has already visited a given node. However, this does not mean that a node is not visited more than once in one search. A query branches out to k_d queries at each node, resulting many different search paths. However, a list of visited nodes is only maintained in each message; therefore, the list in a query message may not contain a node which is visited by a query message on a different search path.

When initiating a new query or upon a receipt of a query, node N_i sends the query concurrently to all nodes N_j such that (1) $AL_j \geq$ *k-threshold* or (2) N_j's tuple is in the top *k-min* among the nodes in $NBRS_i$ that the query has not yet visited. We refer to this the *search-node-determination* algorithm, and the number of query messages to send determined this way is called the k_d value.

3.1. Success and failure of a search

The $FILES_j$ field of each node N_j in $NBRS_i$ works as "one-hop replica", that is used to accelerate the search process. When a node N_i initiates a new query or receives a

query for a target file F_T, it tries to locate F_T not only its own $FILES_i$, but also $FILES_j$ in all tuples in $NBRS_i$, before forwarding the query to its neighbor nodes.

A node N_i claims a search for a target file F_T to be "success" if it locates F_T in $FILE_i$ or in its one-hop replica. If N_i is not the initiator of the query, N_i initiates a "success" message and sends it to its sender N_j. The "success" message contains N_i (the target node) and $FILES_i$. N_j then forwards the "success" message to its sender. This process continues until the "success" message reaches the initiator of the query. All nodes on the path, say N_j, update N_i's tuple in $NBRS_j$ in the manner discussed above.

A node N_i claims a search for a target file F_T to be "fail" if (1) it cannot locate F_T in $FILE_i$ or its one-hop replica, and (2) all nodes in $NBRS_i$ to which N_i could forward the query have claimed the search to be "fail." Then, N_i initiates a "fail" message and sends it back to the sender N_j. If N_i has received "fail" from all nodes that it had sent the query to, it determines a new set of nodes in $NBRS_i$ by applying the above search-node-determination algorithm and sends them the query. This process repeats until either N_i receives "success" or there is no more node in $NBRS_i$ to try (in such a case, it claims the search "fail").

If a query visits a node which has already been visited by a query on a different search path, the node immediately returns "fail" to its sender.

In order to efficiently terminates a search process, we apply a technique called *checkback* [2] A query message contains an integer field called *LiveCount*. When the query visits a new node N_i, N_i decrements LiveCount by one. When the count becomes zero, N_i communicates with the initiator by sending a "checkback" message. Upon a receipt of a "checkback" message, the initiator decides whether it should continue or terminate the search and informs N_i accordingly. If the search is terminated, N_i just stops the search process; otherwise, N_i resets the LiveCount to a predefined value, denoted LC_{INIT}, and continues the search.

3.2. Node join

When a new node N_i joins, it must know at least one node in the system. Let N_j be such a node. N_i first sends a "join" message with its $FILES_i$ to N_j. N_j then replies N_i with a "join ok" message with $NBRS_j$ and $FILES_j$. N_j adds a tuple $[N_i, AL_{DEF}, FILES_i]$ in $NBRS_j$, where AL_{DEF} is a predefined default initial activity level. N_i adds a tuple $[N_j, AL_{DEF}, FILES_j]$ in $NBRS_j$ and uses it as its own neighbor list $NBRS_i$.

3.3. Protocol description

The query message consists of the following Fields: (1) N_I (initiator), (2) F_T (target file), (3) PATH (a list of all nodes that the query has visited), and (4) LiveCount.

The initiator N_I executes the following algorithm when searching target file F_T:

```
var
   search: list of tuples;
   outstanding, found: set of nodes;

if (F_T ∈ FILES_I) return {N_I};
if (F_T ∈ FILES_j such that tuple
   [N_j, *, FILES_j] ∈ NBRS_I) return {N_j};
search = NBRS_I; found = ∅
while search ≠ ∅ do {
   outstanding = set of nodes selected by
            search-node-determination algorithm;
   search -= outstanding;
   send query {N_I, F_T, PATH = {N_I}, LiveCount = LC_INIT}
         to each node in outstanding;
   while outstanding ≠ ∅ do {
      receive a reply from a node N_j;
      outstanding -= {N_j};
      if the reply from N_j is "success" {
         add AL_INC1 to AL_j in NBRS_I;
         if target node N_T is in NBRS_I
            add AL_INC2 to AL_T in NBRS_I
         else
            add [N_T, AL_INIT, FILES_T] to NBRS_I
         Normalize the activity levels in NBRS_I
         found ∪ = {N_T}
      }
   }
}
   return found;
```

When a node N_i receives a query $\{N_I, F_T, \text{PATH, LiveCount}\}$ from N_S, it executes the following:

```
var
   search: list of tuples;
   outstanding, found: set of nodes;
   isSuccess : boolean ;

if (- -LiveCount == 0) {
   send "checkback" to N_T; wait for a reply from N_T;
   if (N_T requests the search to be continued)
      LiveCount = LC_INIT;
   else terminate the search
}
if (F_T ∈ FILES_i) send N_S "success" with {N_i, FILES_i };
if (F_T ∈ FILES_j such that [N_j, *, FILES_j] ∈ NBRS_i)
   send N_S "success" with {N_j, FILES_j };
search = NBRS_i − PATH; isSuccess = false
while search ≠ ∅ do {
   outstanding = set of nodes selected by
         search-node-determination algorithm;
   search -= outstanding;
   send query {N_I, F_T, PATH+ = {N_I}, LiveCount}
      to each node in outstanding;
   while outstanding ≠ ∅ do {
      receive a reply from a node N_j;
```

```
        outstanding -= {N_j};
        if the reply from N_j is "success" {
            add AL_INC1 to AL_j in NBRS_i;
            if target node N_T is in NBRS_i
                add AL_INC2 to AL_T in NBRS_i
            else
                add tuple [N_T, AL_INIT, FILES_T] in NBRS_i
            Normalize the activity levels in NBRS_i
            isSuccess = true;
            forward "success" to N_S;
        }
    }
}
if (¬ isSuccess) send "fail" to N_S;
```

ABS succeeds by actively grading the nodes in the $NBRS$ lists. This grading, called the activity level, is based on previous performance of the node and helps determine the level of flooding; that is, the number of nodes to which a query is forwarded depends on the activity level of a node. The parameter k-threshold is used to control trade-of between the amount of flooding vs. number of hops required to complete a query. The higher the value of k-threshold, lower is the k_d value, and thus less number of messages. This serves to minimize free-riders, and helps create a spanning graph of search paths efficiently (Figure 1), and results in efficient communication in finding the target nodes.

4. Experimental results

We compare the performance of our algorithm with two types of random-walker algorithms: k-RWALK and 1-RWALK, using a well-known discrete event simulator for networks, Network Simulator 2 (NS2) [16]. The difference between k-RWALK and 1-RWALK is that in k-RWALK, the initiator as well as all intermediate nodes send a query concurrently to k randomly selected nodes in their $NBRS$ lists, whereas in 1-RWALK, only the initiator sends a query to k randomly selected nodes and other nodes forward the query only to one randomly-selected node in their $NBRS$ lists.

Adaptive Probabilistic Search (APS)[5] is similar to 1-RWALK. However, in APS, each node maintains a success ratio of previous searches for each object and uses the success ratio to select nodes. In our simulation, since the objects are distributed randomly and the number of the objects is much larger than the number of queries, we expect that few queries will search for the same objects, and therefore, we believe that APS will behave similarly to 1-RWALK.

We measure the performance in two primary metrics: the number of messages generated, and the number of hops per query. The number of messages is the total number of all messages generated by the involved nodes until the target node is found. The number of hops for a query is the number of nodes on the path from the query initiator to the target node. By assuming that forwarding a query to other nodes takes roughly the same amount of time, the number of hops for a query is a good indication of the amount of time that the query took to complete.

We tested our algorithm with 1000 nodes with 1600 queries. In both k-RWALK and 1-RWALK, the k value is static and set to 3. This is simply because with a k value larger than 3, k-RWALK requires too many messages to complete the simulation in reasonable time. In our ABS algorithm, k-min is set to 3, k-threshold varies from 0.4 to 0.8, and the k value (denoted the k_d value) is dynamically determined based on the k-min and k-threshold values. ABS-0.4, ABS-0.6 and ABS-0.8 are used to denote the ABS algorithm with k-threshold values 0.4, 0.6, and 0.8, respectively.

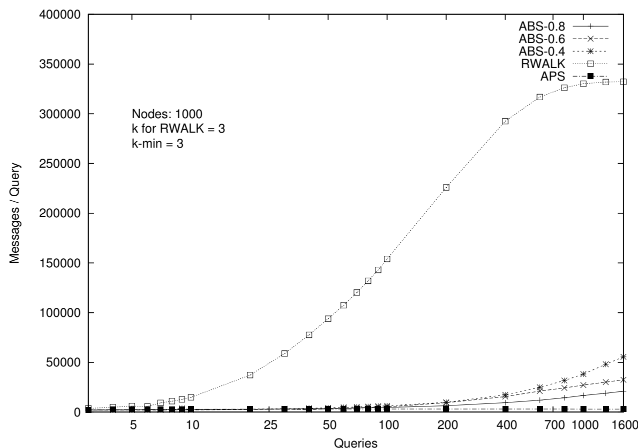
In all three algorithms, each node N_i maintains $NBRS_i$, a set of nodes that N_i knows. In k-RWALK and 1-RWALK, $NBRS_i$ is a list of nodes, and as each node N_i encounters new nodes, the nodes are added to $NBRS_i$. In ABS, $NBRS_i$ is a list of tuples, and only the tuple of a target node is added in the $NBRS$ lists of the nodes on the path from the initiator to the target nodes. Thus, ABS's $NBRS$ lists do not grow as fast as those of k-RWALK or 1-RWALK. In the simulation, the size of the $NBRS$ lists is not limited.

Nodes were created and linked randomly. For each link, its bandwidth, delay, and the queue handling mechanism are also randomly determined. Object placement on nodes is also done randomly with each nodes being assigned 10 objects in average. Thus, each node may or may not have any objects.

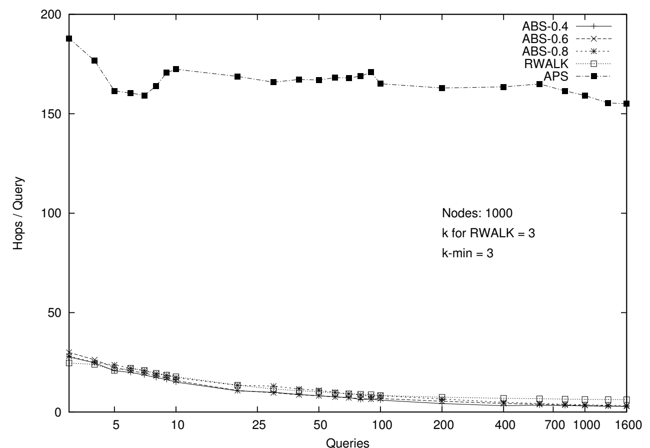
4.1. Performance Comparison

Figure 2(a) compares the number of messages required to complete searches, and Figure 2(b) compares the number of hops from an initiator to a target node. Also note that the average number of nodes that each node concurrently forwards a query to is not always k_d (in ABS), 3 (in k-RWALK), or 1 (in 1-RWALK). This is because as a search progresses, eventually some nodes will claim a search to be "fail"; at this moment, the number of nodes that this node forwards the query to is 0. We use k_r to denote the actual number of nodes that a node forwards a query to. We use notation $\#QUERY$ to denote the number of queries that have been completed.

4.1.1. Comparison of k-RWALK and 1-RWALK First, we compare the performance of k-RWALK and 1-RWALK. We expect that if each node forwards more query messages at once, the total number of messages would increase, however the hops required to complete a search would decrease. In addition, if the number of search paths increases, the probability of the same node being visited more than once



(a) Total messages generated.



(b) Number of hops per query.

Figure 2. ABS performance under NS2 for 1000 nodes.

will increase (as discussed in Section 3). This would result in increase in the number of wasted messages. However, it would contribute to shortening search paths that claim "fail". This is because when a node is visited by a query for the second time and onward, the node immediately returns "fail" to its sender.

When k-RWALK and 1-RWALK are compared, k-RWALK requires significantly more messages but much fewer hops than 1-RWALK. In fact, 1-RWALK constantly requires an extremely small number of messages, however a large number of hops. In contrast, even though k-RWALK requires a constantly low number of hops, its number of messages increases sharply. The reason for this is because the size of $NBRS$ lists increases as the execution progresses. Then, the number of nodes that each node must explore before it claims "fail" will increase. This will result in the sharp increase in the number of required messages as $NBRS$ lists expands, until the lists become large enough so that a search would terminate with "success" before all nodes in the lists are explored. Note that as #QUERY increases in k-RWALK the number of messages increases sharply, but the number of hops does not change much.

Next, we analyze why this phenomenon is not clearly observed in 1-RWALK. It is assumed that since the total number of search paths is limited to k, the possibility of a node being visited multiple times is low. This will make each search path long; that is, each search path tends to visit many nodes sequentially before it claims "success" or "fail". Thus, in 1-RWALK, it is assumed that initial $NBRS$ lists are already large and a search would terminate before all nodes in $NBRS$ have been explored. Therefore, the further increase in the $NBRS$ lists would have little effect on the number of messages.

In the performance of k-RWALK, the number of hops is stabilized at around 8 after #QUERY reaches 75. The number of messages sharply increases until it stabilizes at around 320000 when #QUERY reaches around 700. At this point, it is assumed that $NBRS$ lists become so large that virtually no node claims "fail" before a search terminates, or that most $NBRS$ lists cover all the nodes in the system and cannot expand any further. In the performance of 1-RWALK, after #QUERY reaches somewhere around 10 and 20, both the number of hops and the number of messages are stabilized at around 170 (very large) and 3000 (very few), respectively.

4.1.2. Performance analysis of ABS Based on the above observation, we now analyze the performance and behavior of ABS. In both metrics, ABS demonstrates extremely good performance.

First, we compare ABS with 1-RWALK. Even when the average k_r values of ABS is small (up to 1.5 when #QUERY is 9), there is a significant difference in the number of hops: about 30 (ABS) vs. 170. This is because in ABS, even when the average k_r value is small, k_r would vary from 0 to a value larger than 1, and as a result, there are many concurrent search paths. In 1-RWALK, individual k_r values of nodes other than the initializing node N_I are either 0 or 1, and at most three search paths exist concurrently. The number of hops in ABS stabilizes at k_r value of around 8. The number of messages in ABS gradually increases to 15000, whereas 1-RWALK stabilizes at around 3000.

Next we compare ABS with k-RWALK. Up to #QUERY being around 200, k-RWALK shows slightly better performance than ABS in terms of the number of hops. After that, ABS demonstrates even slightly fewer number of hops than

	1	2	4	8	10	50	100	300	800	1600
ABS	0.72	0.89	1.29	1.45	1.54	2.42	4.24	25.85	50.98	82.31
APS	0.53	0.63	0.74	0.85	0.87	0.97	0.98	1.00	1.00	1.00
RWALK	0.82	1.22	1.8	2.37	2.46	2.89	2.94	2.98	2.99	

Table 1. k-values as the number of requests increases.

k-RWALK.Importantly, however, ABS achieves such performance with a significantly fewer (80% at 18 hops) messages than k-RWALK.

Among the ABSs, a smaller k-threshold value results in a larger k_d value. As expected, ABSs with smaller k-threshold values require more messages than those with larger k-threefold values. However, no significant difference is found in the number of hops among ABSs with various k-threshold values.

5. Conclusions and future work

In this paper we have presented a scalable P2P search algorithm using a method to dynamically determine the value of k_d, the number of nodes to send a query to at once. The decision is based on the degree to which each neighbor has contributed to a successful search, called the activity level. Each node sends a query to K_d neighbors with the highest activity levels at once. A distinctive feature of the algorithm is that, unlike APS, a node does not consider the semantic preference when choosing the nodes to forward a query to. A node simply considers the previous success ratio, regardless of the objects being searched, to determine k_d and the nodes to send a query to. Therefore, its scalability is better than that of APS in terms of the storage size.

Our algorithm automatically creates a spanning graph consisting of high traffic links. Once a stable spanning graph is created, the number of hops required for a search is roughly bound by the diameter of the spanning graph. The simulation exhibits that our algorithm demonstrates significantly better performance than both k-random walk and 1-random walk. Our message-per-query is much less than k-random walk, yet our number of hops to reach the target nodes is much less than 1-random walk and slightly less than even k-RWALK.

Our algorithm has significant potential for further improvement. For example, as APS does, we could reflect the success ratio of each object or of each category of objects in evaluation of activity levels. Another possibility is to consider the capacities of nodes and links as [15] does when determining the activity levels.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under the award number CCR-0082667. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Gnutella website: http://gnutella.wego.com.

[2] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. of ICS*, 2002.

[3] S. Androutsellis-Theotokis. A Survey of Peer-to-Peer File Sharing Technologies. White paper WHP-2002-03, ELTRUN, 2002.

[4] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *Proc. of IEEE INFOCOM*, 2004.

[5] Dimitrios Tsoumakos and Nick Roussopoulos. Adaptive Probabilistic Search (APS) for Peer-to-Peer Networks. In *IEEE Intl. Conf. on P2P Computing*, 2003.

[6] S.Ratnasamy, P.Francis, M.Handley, and R.Karp. A scalable content-addressable network. In *Proc. of SIGCOMM*, 2001.

[7] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, April 2001.

[8] Ronaldo A. Ferreira, Murali Krishna Ramanathan, Suresh Jagannathan, and Ananth Grama. Efficient Randomized Search Algorithms in Unstructured Peer-to-Peer Networks. Purdue University Technical Report PUTR-CS04-022, 2004.

[9] Napster website: http://www.napster.com.

[10] Kunwadee Sripanidkulchai, Bruc Maggs, and Hui Zhang. Efficient content location using interest based locality in peer-to-peer system. In *Proc. of IEEE INFOCOM*, 2003.

[11] Alexander Löser and Christoph Tempich. On Ranking Peers in Semantic Overlay Networks. In *3rd Conference on Professional Knowledge Management*, 2005.

[12] Hailong Cai and Jun Wang. Foreseer: A Novel, Locality-Aware Peer-to-Peer System Architecture for Keyword Searches. In *Middleware 2004, ACM/IFIP/USENIX International Middleware Conference*, 2004.

[13] Alexander Löser and Steffen Staab and Christoph Tempich. Semantic Methods for P2P Query Routing. In *3rd. German Conference on Multi Agent Technologies*, 2005.

[14] V. Cholvi, P. A. Felber, and E. W. Biersack. Efficient search in unstructured peer-to-peer networks. Technical Report RR-03-090, Institut Eurècom, 2003.

[15] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham and Scott Shenker. Making Gnutella-like P2P Systems Scalable. In *Proc. of ACM SIGCOMM*, 2003.

[16] Network Simulator 2 http://www.isi.edu/nsnam/ns/