Adaptive Probabilistic Search (APS) for Peer-to-Peer Networks

Dimitrios Tsoumakos, Nick Roussopoulos
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland,
College Park, MD, 20742
{dtsouma, nick}@cs.umd.edu

CS-TR-4451, UMIACS-TR-2003-21

February 25, 2003

Abstract

Peer-to-Peer networks are gaining increasing attention from both the scientific and the large Internet user community. Popular applications utilizing this new technology offer many attractive features to a growing number of users. At the heart of such networks lies the data retrieval algorithm. Proposed methods either depend on the network-disastrous flooding and its variations or utilize various indices too expensive to maintain. In this report we describe an adaptive, bandwidth-efficient and easy to deploy search algorithm for *unstructured* Peer-to-Peer networks, the *Adaptive Probabilistic Search* method (*APS*). Our scheme utilizes feedback from previous searches to probabilistically guide future ones. Extensive simulation results show that *APS* achieves high success rates, increased number of discovered objects, very low bandwidth consumption and adaptation to changing topologies.

1 Introduction

Peer-to-Peer (P2P) networking has been growing rapidly in the last few years. P2P represents the notion of sharing resources available at the edges of the Internet. Its success was originally boosted by some very popular file-sharing applications (e.g., [NAP, AUD]). Numerous systems that utilize or support P2P technology have emerged since (e.g., [DFM01, GNU, SET, NET]). Bandwidth consumption attributed to P2P file-sharing systems amounts to a considerable fraction of the total Internet traffic, despite the shutdown of several systems due to legal rulings.

We can roughly classify P2P architectures into two categories: *Centralized* approaches utilize a central directory for object location, ID assignment, etc. *Decentralized* approaches abandon this solution to employ a distributed directory structure. *Pure* decentralized systems exhibit a fully distributed behavior with all peers (or nodes) being equal, while in *hybrid* systems some nodes act as *super-peers*, serving the shielded peers they neighbor with. Another taxonomy classifies P2P networks into *structured* and *unstructured*. *Structured* networks provide strict rules for file placement and object discovery, while *unstructured* approaches offer arbitrary network topology, file placement and search. We focus our attention on *decentralized unstructured* P2P systems, which have proved to be of greater impact to the network community [SAN] than systems with strict guarantees.

Due to the large popularity and enormous amounts of data exchanged, it is vital for P2P search to be performed in a fully distributed, bandwidth-efficient and adaptive manner. A search for an object in a P2P network is *successful* if it discovers at least one replica of the object. The ratio of successful to total searches made is the *success rate* (or *accuracy*) of the algorithm. A search can result to multiple discoveries (or *hits*), which are copies of the same object stored at *distinct* nodes. *Duplicate* messages are copies of the same query sent to a node that has already processed it. The *performance* of an algorithm is associated with its success rate and number of hits, while its *cost* relates to the number of messages it produces.

Search methods can be categorized as either *blind* or *informed*. In a *blind* search, nodes do not store any information regarding file locations. In *informed* approaches, nodes locally store metadata that assist in the search for the queried objects. Current blind search methods waste a lot of bandwidth to achieve high performance. Every search requires contacting many nodes within some distance called *time-to-live* (TTL), generating huge overhead to all nodes involved. This approach aims at finding the maximum number of results within an area of the network with the originating node being at the center and the radius being a TTL-related parameter. Informed methods use their indices in order to achieve similar quality results (by choosing "good" neighbors to forward the query to) and to reduce overhead. The shortcoming of most informed methods is the maintenance cost of the indices after peers join/leave the network or update their collections. In most cases, these events trigger *floods* of update messages, inflating network traffic.

Current P2P search algorithms can be roughly divided in two categories: The ones that exhibit good performance at high cost (either during the search or the metadata updates), and those that are bandwidth-efficient but exhibit varying performance. The important observation is that popular P2P networks display a highly dynamic behavior, with most users connecting for small periods of time and then leaving the system [CLL02]. Any algorithm that fails to scale along this pattern, inevitably puts excessive burden on network traffic.

Many search protocols for *unstructured* networks have been proposed with an intention to reduce the overhead of the original flooding scheme. In the *Random Walks* algorithm [LCC+02], the requesting node sends out k query messages to an equal number of randomly chosen neighbors. Each of these queries follows its own path, having intermediate nodes forward it to a randomly chosen neighbor at each step. These queries are known as *walkers*. While this approach manages to reduce messages by more than an order of magnitude, it exhibits low performance due to its random nature and inability to adapt to different query loads.

In this work, we propose a new search algorithm that achieves high performance at low cost, the *Adaptive Probabilistic Search* method (*APS*). In *APS*, a node deploys k walkers for object discovery, but the forwarding process is probabilistic instead of random. Peers effectively direct walkers using feedback from previous searches, while keeping information only about their neighbors. As we show in this work, *APS* exhibits many plausible characteristics, such as:

- High accuracy
- Low bandwidth consumption
- Large number of discovered objects
- Robust and adaptive behavior in rapidly-changing environments

These features come as a result of our algorithm's *learning* character, which enables peers to share, refine and adjust their search knowledge with time. Furthermore, *APS* induces zero overhead over the network at join/leave/update operations.

We make the following research contributions in this report:

1. We define the *APS* algorithm for search in unstructured P2P networks. We describe the main idea, the indexing scheme, the search and update procedures and analyze its performance.
2. We present two improved versions of the algorithm which exhibit significant gains in message reduction and the number of objects discovered near the requesters.
3. We perform extensive simulations and compare *APS* with the *Random Walks* and *GUESS* methods over different environments. Our algorithm achieves excellent results in the success rate, number of discovered objects, message consumption and adaptation to changing topologies.

Section 2 gives an overview of the related work. In Section 3 we present our algorithm, discuss its characteristics and describe two improved versions of the scheme. Extensive simulation results follow in Section 4. Finally, Section 5 contains our conclusions.

2 Related Work

Structured Peer-to-Peer networks have received a lot of attention in the last few years, from both academia and industry. Systems in that category include, but are not limited to, OceanStore [KBC+00], CFS [DKK+01], PAST [RD01b] and Publius [WRC00]. All operations in these systems are based on an "overlay" network, which handles file and replica placement and guarantees bounded number of steps and reliable storage. Examples of such overlays include CAN [RFH+00], Chord [SMK+01], Pastry [RD01a] and Tapestry [ZKJ01]. All these networks operate well under the assumption that the primary goals are consistency and object persistence.

Unstructured Peer-to-Peer networks have also been studied a lot in the last few years. In this category we recognize popular file sharing systems like Gnutella [GNU] and Kazaa [KAZ]. The original Gnutella algorithm uses flooding (BFS traversal of the underlying graph) for object discovery. [KGZY02] proposes a variation of the flooding scheme with peers randomly choosing only a ratio of their neighbors to forward the query to. Two similar approaches that use consecutive BFS searches at increasing depths are described in [YGM02, LCC+02]. The notion of *Ultrapeers* [SR] is used by two new search protocols for Gnutella-type networks. We attempt an initial description at this point, although the references were not complete. Nodes are categorized as either ultrapeers or leaf-nodes. Each ultrapeer is connected to other ultrapeers and to a set of leaf-nodes (peers shielded from other nodes), acting as their proxy. In *GUESS* [DF], a search is conducted by iteratively contacting different ultrapeers and having them ask all their leaf-nodes, until some predefined condition is met. In Gnutella2 [Sto], when an ultrapeer (or *hub*) receives a query from a leaf, it initiates the equivalent of the original Gnutella search with $TTL = 2$. Neighboring hubs regularly exchange local repository tables to filter out unnecessary traffic. Compared to these approaches, *APS* uses informed walkers instead of flooding and its variations. Moreover, our indices reflect the relative probability of success through a neighbor, not the neighbors' local repository.

An approach similar to [LCC+02] was described in [ALPH01], where the degrees of the nodes are used to guide the walkers in a power-law graph. Our algorithm uses hints that relate to search results, not network topology. This way, both performance enhancement and knowledge build-up can be achieved.

Several informed methods have also been proposed. In [KGZY02], each node forwards keyword requests to a set number of neighbors that have answered the most requests "similar" to the current one, according to a query similarity metric. Nodes store information about recently answered queries in order to rank their neighbors. This approach focuses more on object discovery than message reduction and seems tailored for applications where nodes store correlated data. Instead of just receiving positive feedback from previous searches, our approach also utilizes negative feedback from the walkers so that efficient unlearning is performed, while neighbors are probabilistically chosen, not ranked. In our work, we also focus on reducing both search messages and the distance to the discovered objects.

In [YGM02], a node holds information about all files stored at nodes within a certain radius and can answer queries on behalf of all of them. In [CGM02], nodes store file content metadata for each of their outgoing links, enabling them to forward a query to the neighbor with the highest value of a defined metric. In [RK02], each node holds d *bloom* filters for each neighbor. A filter at depth i summarizes documents that can be found through that specific link i hops away. Nodes forward queries to the neighbor whose higher depth bloom filter matches a hashed representation of the document ID. In our approach, nodes keep indices regarding only their neighbors, avoiding the cost of updates at every change. Furthermore, our index semantics relate to previous search results, not file locations.

Freenet [CSWH01] uses a DFS-like search with the exception that nodes forward to neighbors after consulting a routing table from previous searches. Discovered documents are also cached along the reverse path. In our method, the algorithm does not dictate object replication, while it deploys k walkers simultaneously instead of one at a time.

3 The *APS* Algorithm

3.1 Our Search Model

The following assumptions are made throughout the discussion that follows:

Peers initiate searches for various objects. These objects are distributed across the network according to a *replication distribution*, which dictates what objects are stored at each node. The *query distribution* dictates how many requests are made for each object (e.g., popular objects get many more requests than unpopular ones). The search algorithms cannot in any way dictate object placement and replication in the system. They are also not allowed to alter the topology of the P2P overlay. A node is directly connected to its *neighbors*, and these are the only peers whose addresses the node is aware of.

Nodes can keep some *soft state* (i.e., information that is erased after a short amount of time) for each query they process. Each search is assigned an identifier, which, together with the soft state, enables peers to make the distinction between new and duplicate messages. Identifiers are also assigned to objects and nodes from a flat, non-hierarchical space. The TTL parameter represents the maximum hop-distance a query can reach before it gets discarded, while k denotes the number of walkers deployed for search from a requester node.

Finally, the metrics we use to evaluate search algorithms are the success rate, the number of discovered objects, the number of messages and duplicate messages produced and the distance of the discovered objects from the requester nodes. For simplicity reasons, we ignore network and processing delays. While such delays affect response time, they cannot impact our metrics.

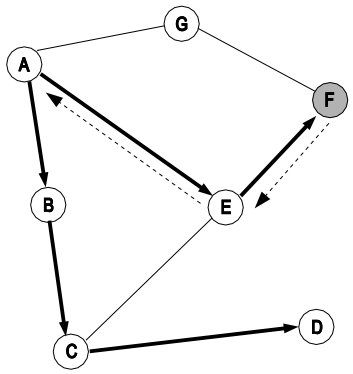
3.2 Algorithm Description

In *APS*, each node keeps a local index consisting of one entry for each object it has requested, or forwarded a request for, per neighbor. The value of each entry reflects the relative probability of this node's neighbor to be chosen as the next hop in a future request for the specific object.

Searching is based on the deployment of k walkers and probabilistic forwarding: The requester chooses k out of its N neighboring nodes[1] to forward the request to. Each of these nodes evaluates the query against its local repository and if a hit occurs, the walker terminates successfully. On a miss, the query is forwarded to one of the node's neighbors. This procedure continues until all k walkers have terminated, either with a success or a failure. So, while the requesting node forwards the query to k neighbors, the rest of the nodes forward it to only one. In the forwarding process, a node chooses its next-hop neighbor(s) not randomly, but using the probabilities given by its index values. At each forwarding step, nodes append their identifiers in the search message and keep a soft state about the search they have processed. If two walkers from the same request cross paths (i.e., a node receives a duplicate message due to a cycle), the second walker is assumed to have terminated with a failure and the duplicate message is discarded.

Index values stored at peers are updated in the following manner: When a node chooses one

[1] If $k \geq N$, the query is sent to all neighbors



Indices	Initially	At walker termination	After index updates
A→B	30	20	20
B→C	30	20	20
C→D	30	20	20
A→E	30	20	40
E→F	30	20	40
A→G	30	30	30

Figure 1: Search for an object stored at node F using the *pessimistic* approach of *APS* with two walkers. The table shows how various index values change, where X→Y denotes the index value stored at node X for neighbor Y relative to the requested object.

or k peers to forward the request to, it pro-actively either increases the relative probability of the peer(s) it picked, assuming the walker(s) will be successful (*optimistic* approach), or it decreases the relative probability of the chosen peer(s), assuming the walker(s) will fail (*pessimistic* approach).

Upon walker termination, if the walker is successful, there is *nothing* to be done in the *optimistic* approach. If the walker fails, index values relative to the requested object along the walker's path must be corrected. Using information available inside the search message, the last node in the path sends an *"update"* message to the preceding node. This node, after receiving the update message, *decreases* its index value for the last node to reflect the failure. The update procedure continues along the reverse path towards the requester, with intermediate nodes decreasing their local index values relative to the next hops for that walker. Finally, the requester decreases its index value that relates to its neighbor for that walker. If we employ the *pessimistic* approach, this update procedure takes place after a walker succeeds, having nodes increase the index values along the walker's path. There is nothing to be done when a walker fails.

Figure 1 shows an example of how the whole process works. Node A initiates a request for an object owned by node F using two walkers. Assume that all index values relative to this object are initially equal to 30 and the *pessimistic* approach is used. The paths of the two walkers are shown with thicker arrows. During the search, the index value for a chosen neighbor is reduced by 10. One walker with path (A,B,C,D) fails, while the second with path (A,E,F) finds the object. The update process is initiated for the successful walker on the reverse path (along the dotted arrows). First node E, then node A increase the value of their indices for their next hops (nodes F, E respectively) by 20 to indicate object discovery through that path. In a subsequent search for the same object, peer A will choose peer B with probability 2/9 ($=\frac{20}{20+40+30}$), peer E with probability 4/9 and peer G with probability 3/9.

Our method utilizes "probabilistic" walkers with a *learning* feature that incorporates knowledge from past and present searches to enhance future performance. The learning

process adaptively directs the walkers to promising parts of the network, while keeping bandwidth consumption low.

APS requires no message exchange on any dynamic operation such as node arrivals or departures and object insertions or deletions. The nature of the indices makes the handling of these operations simple: If a node detects the arrival of a new neighbor, it will associate some initial index value with that neighbor when a search will take place. If a neighbor disconnects from the network, the node removes the relative entries and stops considering it in future queries. No action is required after object updates, since indices are not related to file content. So, although our algorithm actively uses information, its maintenance cost on any of these events is zero, a major advantage over most current approaches.

3.3 Discussion

Each node stores a relative probability (e.g., an unsigned integer value) for each of its neighbors for each (directly or indirectly) requested object. So, for $\mathcal{R}$ such objects and N neighbors, $O(\mathcal{R} \times N)$ space is needed. For a typical network node, this amount of space is not a burden. On nodes with limited storage capacities, index values for objects not requested for some time can be erased. This can be achieved by assigning a time-to-expire value on each newly-created or updated index, or by employing simple LRU or LFU replacement policies. Each search or update message carries path information, storing a maximum of TTL peer addresses. Alternatively, each node can associate the search and requester node IDs with the preceding peer in the path of the walker. Updates then follow the reverse path back to the requester. This information expires after a certain amount of time. A selection from the above techniques depends on the application, query workload and node capabilities.

Let us calculate how many messages it will take for the *APS* method to terminate. In the worst case — all walkers travel TTL hops and then invoke the update procedure — the number of messages exchanged will be $2 \times k \times TTL$, so the method has the same complexity with its random counterpart. The only extra messages that occur in *APS* are the update messages along the reverse path. This is where our two index update policies are used: If we expect or experience after a while that for a specific number of walkers k, only few of them terminate successfully, then the *pessimistic* mode should be employed. Conversely, if many of our walkers hit their targets on average, the *optimistic* approach should be considered. Naturally, the two approaches have the same performance in all other metrics.

Along the paths of all k walkers, indices are updated so that better next hop choices are made with bigger probability. Our learning feature includes both positive and negative feedback from the walkers in both update approaches. In the *pessimistic* approach, each node on the walker's path decreases the relative probability of its next hop for the requested object concurrently with the search. If the walker succeeds, the update procedure increases those index values by more than the subtracted amount (positive feedback). So, if the initial probability of a node for a certain object was P, it becomes bigger than P if the object was discovered through (or at) that node and smaller than P if the walker failed. The learning process in the *optimistic* approach operates in an opposite fashion, with negative feedback taking place after a walker fails. Our algorithm exhibits both *learning* and *unlearning* characteristics: *Learning* is important to achieve both high performance and discovery of newly

inserted objects. *Unlearning* helps our search process adjust to object deletions, redirecting the walkers elsewhere.

Another characteristic of the algorithm is its ability to obtain more knowledge with more questions. The more feedback from the walkers, the more precise the indices become. That particularly suits the discovery of popular objects in the P2P network, which, according to studies [CLL02], constitute over 60% of all searches. Another similar observation is that all nodes participating in a search will benefit from the process. This is a distinctive feature of our method, with indices being constantly updated using search results and not after object updates. In our case, *both* requesters and peers on the paths of all walkers actively adjust their knowledge about the specific object. A node that has never before requested an object but is "near" peers that have done so, inherits this knowledge by proximity. Besides standard resource-sharing in P2P systems, our algorithm achieves the distribution of *search knowledge* over a large number of peers.

3.4 Algorithm Improvements

APS produces update messages to adjust index values along the paths of some walkers. Our goal is to minimize these messages in order to further reduce the level of bandwidth consumption. Obviously, if fewer than $k/2$ walkers are successful, then the *pessimistic* approach should be employed instead of the *optimistic* and vice versa. Choosing one strategy over the other for queries over all objects is not optimal, as many unnecessary update messages are produced for both popular and unpopular object requests. In *swapping-APS* (*s-APS*), the algorithm constantly monitors the ratio of successful walkers for each request and accordingly switches to the update policy that produces fewer messages. This makes our *s-APS* improvement more bandwidth efficient, sometimes producing a lower total number of messages *even* from *Random Walks*, which uses no update messages. The number of requests for which nodes monitor the successful walker ratio depends on available node storage, although the overhead will be very small in most cases.

Another improvement relates to the index update procedure. In the original scheme, all index values are updated by the same amount, using a simple step function. In *weighted-APS* (*w-APS*), we incorporate a distance-based function for modifying the relative probabilities stored at each node. Index values for peers closer to the discovered object are increased more than those for distant nodes[2]. Distance information is directly accessible from the stored path inside the search messages. With this method, peers are biased to direct walkers to objects that are near. Our results show a significant increase in the number of discovered objects located near the requesters.

4 Simulation Results

We performed extensive simulations to evaluate the performance of *APS*. We first briefly describe our simulation model and then present results for both static and fully dynamic

[2]The *weighted* modification is used with the *pessimistic* approach, where the distance from an object is known.

network operation.

4.1 Simulation Methodology

We used two different graph models to simulate our network: The *random graph* model and the popular *power-law* model. Since the application overlay works with logical connections and does not reflect the underlying network, we mainly used the random graph model to simulate our P2P overlay structure. We also used two P2P models, *pure* and *hybrid*: In the pure model all peers equally pause and answer requests; in the hybrid model nodes are organized in an *Ultrapeer*-leaf hierarchy. We utilized two well-known topology generators: GT-ITM [ZCB96] for the pure and hybrid random graph models and Inet-3.0 [JCJ00] for the power-law graph model.

Before each simulation, object replication and query distributions are set. We choose from three different distributions, namely uniform, zipf and 80/20. Requester nodes and query loads are also saved so that each run uses the same configuration. Requesters are randomly chosen and always represent a noticeable fraction[3] of the size of the graph. Requests arrive "sequentially" in the system. In order to simulate a dynamic network behavior, we insert "on-line" nodes and remove active ones every 1000 queries. On average, the network changes more than 3000 times during each simulation. We always keep approximately 80% of the network nodes active. Arriving nodes start functioning without any prior knowledge.

The approach we use for walker termination is the TTL-based method. The initial value of an index is 30 and its minimum value is 10, so that no nodes are precluded from the forwarding process. Index values are increased (decreased) by 10 during the search and decreased (increased) by 20 during the updates with the *optimistic* (*pessimistic*) approach. For the *weighted* scheme, the amount of increase is inversely proportional to the distance from the object raised to the power of 2.5.

Table 1 summarizes our simulation parameters and their default values.

We found that it is best to set k equal or higher to the average out-degree and TTL to 5 or 6. We used 100 objects in most simulations for simplicity and speed. An increase in that number does not affect the quality of the results. For the query and replication strategies, we followed the observations in [CLL02] but preferred a less skewed distribution, where the highest-ranked 10% of objects amount to about 30% of the total number of stored objects and receive about 30% of the total requests. For our model with 100 objects this was a more realistic choice, making requests for all of them. A larger exponent ($a \cong 1.5$) can be used if the number of objects increases.

4.2 Non-Dynamic Behavior

In this section we examine the characteristics of APS with a static network. This is equivalent to taking a "snapshot" of the network, where all nodes remain active for the duration of the queries. In the following figures, if one or more of our algorithm's variations are compared, they will be specifically mentioned with their names (e.g., *w-APS*). The label "APS" is used when all our other methods have very similar performance in a particular metric.

[3] More than 10%

Simulation Parameters	Default Values
Number of Nodes	10000
Graph model	Random
P2P model	Pure
Average node degree	10
Walkers deployed (k)	12
TTL	6
Number of Objects	100
Replication Distribution	Zipf ($a = 0.82$)
Query Distribution	Zipf ($a = 0.9$)
Number of Requester Nodes	1000
Number of Queries per Requester Node	3162

Table 1: Simulation parameters and their default values

For the default graph, our simulations show that the standard flooding scheme with $TTL = 4$ can be successful in over 99% of its searches, while producing over 9000 messages per search. Since these properties are well-known, we focus on the *Random Walks* and *APS* algorithms.

In the first set of simulations, we varied the number of walkers deployed (k) from 1 to 15 for the default parameters. Figure 2 presents the success rates of the two algorithms. We can see that *APS* achieves very high success rates even with few deployed walkers, while it manages to enhance its performance as more walkers are used. It is steadily around 40% more accurate than *Random Walks*, which exhibits low performance even with many walkers. We also notice that the *weighted* scheme shows an increase of around 4% in successful searches over our standard method.

In Figure 3 we present the average number of messages per search. One would expect that our method produces a much higher number of messages compared to *Random Walks* due to the update procedure, but this is not the case: The majority of walkers in *APS* are successful and only few of them reach TTL hops away. In *Random Walks*, about 70% of the walkers fail and waste TTL messages each. To a lesser extent, objects are equally discovered at all possible distances in the random method, while our scheme discovers more objects closer to the requesters. The results confirm our case: The difference varies between 0.2 and 15 messages for the *pessimistic* approach, drops to only 4 messages for the *optimistic* approach, while *s-APS*, using the best update strategy each time, *outperforms* the random algorithm. For an informed method, our technique achieves an amazingly low per query consumption. This effect is enhanced if we recall that no message exchange is necessary for peer join/leave/update operations. Not surprisingly, in some simulations (when 6 or more walkers are deployed) the *optimistic* approach produced more messages, while for fewer walkers the *pessimistic* approach proved more bandwidth-efficient.

Figure 4 displays the vast reduction *APS* achieves in wasted bandwidth. Duplicate messages are considered to be failure states for our walkers, therefore the learning process makes adjustments in order to minimize walker collisions. Our method constantly outperforms its competition, producing 1 to 2 orders of magnitude fewer duplicate messages. This is also im-

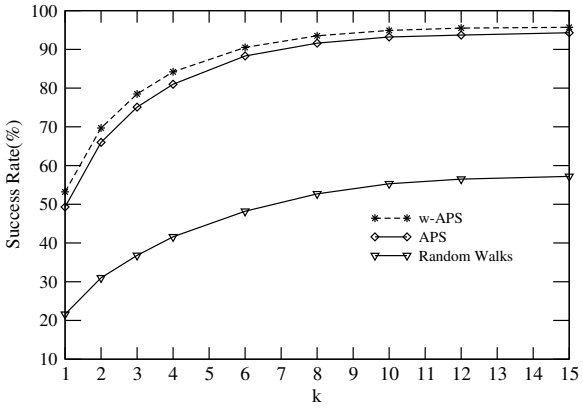


Figure 2: Success rate vs number of deployed walkers

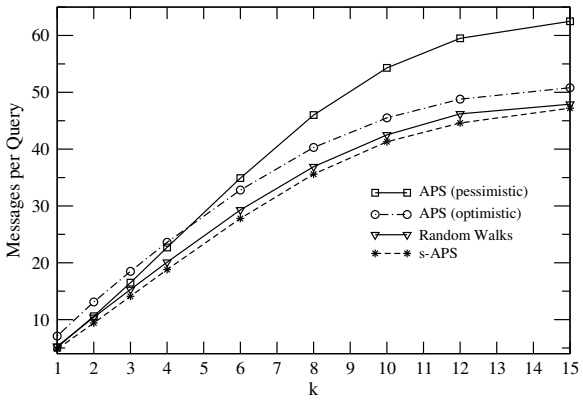


Figure 3: Messages per query vs number of deployed walkers

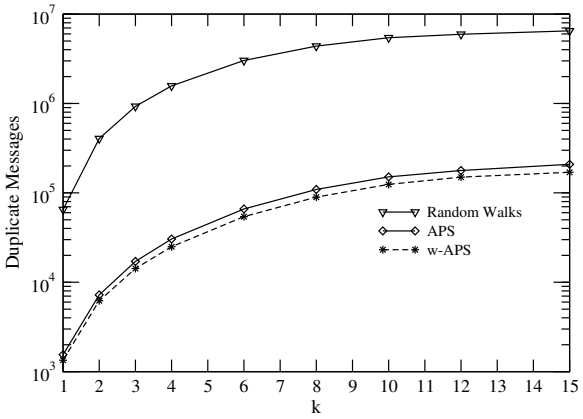


Figure 4: Duplicate messages vs number of deployed walkers

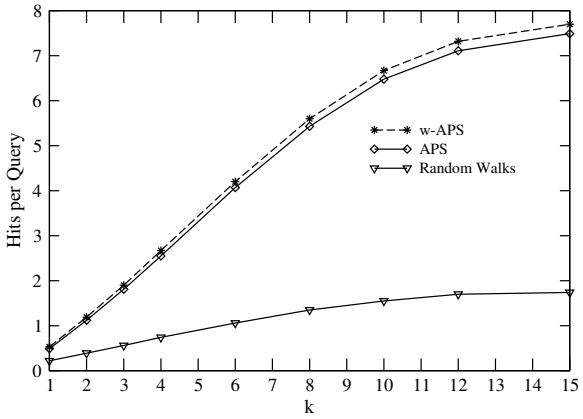


Figure 5: Number of hits per query vs number of deployed walkers

portant because it increases the useful processing time for each peer. The *weighted* approach exhibits almost 20% fewer duplicate messages than our default methods.

Figure 5 depicts the average number of discovered objects per query. *APS* puts the walkers to a much better use, achieving over 65% in walker accuracy (successful walkers over all walkers deployed) and around 4 times as many discovered objects as its competitor. This comes as an immediate side-effect of its high success rate and very few walker collisions. Since walkers are directed to (possibly) different parts of the network where different copies of an object exist, the successful walker percentage increases. This is extremely important for current popular P2P applications, giving the user a much broader choice for download. The *w-APS* variation produces marginally better results here.

Finally, Figure 6 shows how hits are distributed over their distance from the requesters, for the default parameters. *Random Walks* exhibits a flat curve, discovering about the same amount of objects throughout the 1 to TTL range. On the other hand, *APS* is more biased into the first half of this range. Our *weighted* technique significantly improves on this

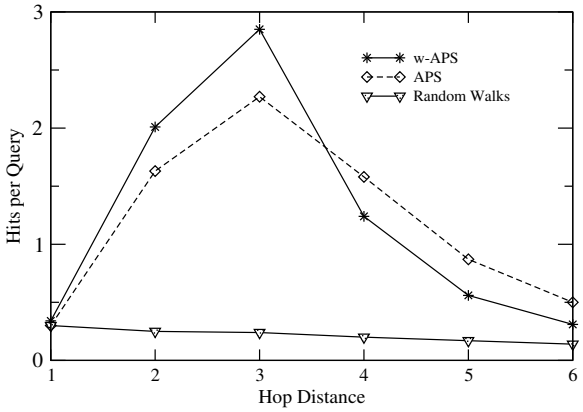


Figure 6: Hits per query vs hop distance from requesters

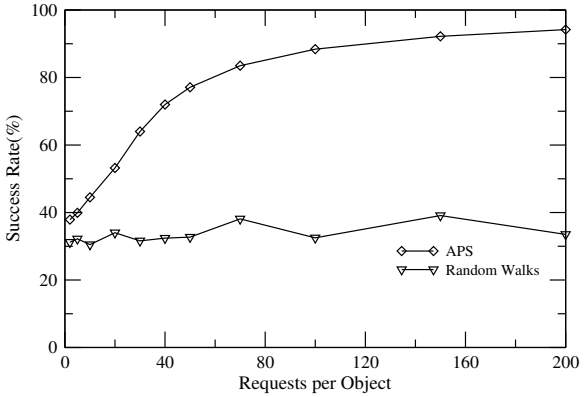


Figure 7: Success rate vs number of requests per object

characteristic, trading distant objects with closer ones. This way, the majority of discovered objects are found with few messages, while fewer objects are discovered with more messages. This is why *w-APS* performs better for distances 1 to 3 hops, while it produces fewer hits for hops 4 to 6.

Figure 7 displays how the number of requests affects accuracy. With just 1% object replication ratio, $k = 10$ and $TTL = 5$, we varied the number of requests per object using a uniform distribution for both storage and requests on the default graph. As we can see, accuracy improves significantly with only a small increase in requests, even though only about 100 copies of each object exist. At the same time, *Random Walks* is steadily below 40%, regardless of the number of requests.

Next, we analyze the behavior of our scheme's index values. *APS* is an inherently adaptive search algorithm, whose power lies in the use of the local indices. For the next experiment, we choose only one node from our default graph with degree 12 and examine how its local indices change. We make 1000 requests for 10 objects, with object 1 being the most popular and 10 the least. Replication and request distributions take their default values. Figure 8 displays the number of high-valued indices for that node for all 10 objects. Object popularity decreases from left to right on the x-axis. We monitor indices with very large values (over 1000) and indices that have a fairly large value (over 100 but below 1000). We notice that many indices with large values exist for the very popular objects, while this number decreases as popularity drops. Still, some indices with a relatively large value always exist for less popular objects. *APS* exhibits high precision for very popular objects, building up its "confidence" through large index values. On the other hand, the few fairly large indices for unpopular objects point out the algorithm's ability to locate them with good probability.

In Figure 9 we show the success rates for individual objects grouped according to their popularity, using all default parameters. Popularity decreases from left to right on the x-axis. *APS* shows almost perfect results for popular objects, while displaying a "graceful" decline for unpopular requests. *w-APS* exhibits improved resilience and manages to keep a higher accuracy level for unpopular requests. On the other hand, *Random Walks*' accuracy drops significantly after requests for the highest-ranked 10% of objects, reaching a mere 11% for

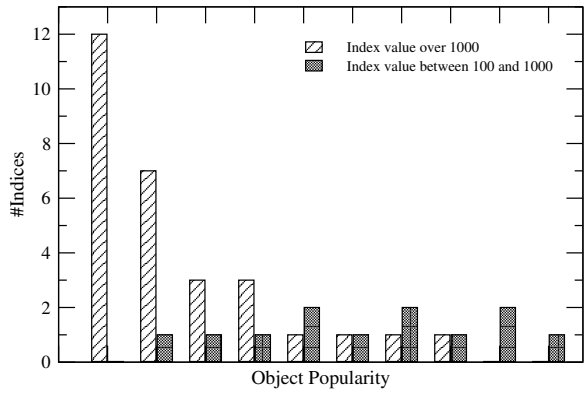


Figure 8: Number of indices with certain values according to object popularity

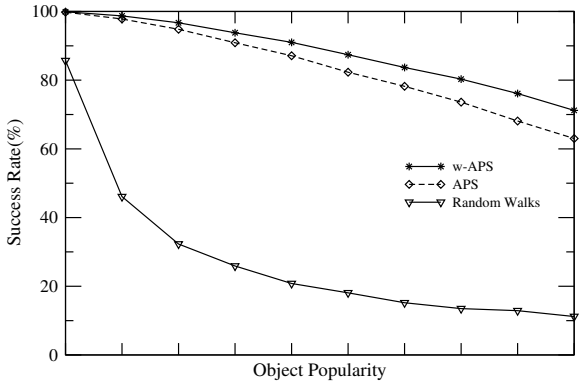


Figure 9: Individual success rate vs object popularity

the least popular objects.

In Table 2 we conclude our results on pure P2P networks. We compare *s-APS* with *Random Walks* over three different topologies: The default one, a 50000-node random graph with average degree 10 and a 10000-node power-law (PLAW) graph with average degree 4.4. We monitor the success rate and the average number of hits, messages and duplicate messages per query. Apart from the default setup, we also test the algorithms using the uniform distribution for both requests and storage. This may be a more suitable model for other kinds of P2P applications, for example sharing of sensor data between wireless ad-hoc peers. The replication ratio is set to 1% and each object is requested 30 times. We clearly notice that *s-APS* can greatly benefit from such a setup, delivering over 96% in success rate and discovering 4 times more results than *Random Walks*. Our simulations on the 50000-node random graph justify our prediction that the graph size cannot influence the performance of *APS*. The results were a little worse from the ones in the original graph, because the quality of the new graph was worse (many more disconnected components were present). Our results on the 10000-node power-law graph show an even greater gap in the performance of the two algorithms, although both produce worst numbers in all metrics compared to the original graph. This can be explained by the fact that the average out-degree of the Inet topology is less than half the value of the random graph, significantly reducing the average number of deployed walkers per search. Still, our method is over 40% more accurate and delivers 3 times as many results as the random method. In all these simulations, we notice that *s-APS* produces almost the same number of messages with *Random Walks*, wasting at most 4 extra messages per search.

Finally, we present results comparing *s-APS* with an implementation of *GUESS* [DF] on a random *hybrid* graph with 6500 peers, 500 of them being ultrapeers. Each ultrapeer is connected to 12 leaf-nodes on average. Links exist only between ultrapeers and between an ultrapeer and its leaf-nodes. In our *GUESS* implementation, initiating ultrapeers forward queries to k randomly chosen neighbor ultrapeers. Query and replication distributions are set to their default values. Since it is impossible to directly compare the two methods for the same k and TTL values, we selected simulations where the two algorithms had similar

Method	*s-APS*				*Random Walks*			
	Succ(%)	Mesg	Hits	Dupl	*Succ(%)*	Mesg	Hits	Dupl
Default graph(Zipf)	94.0	44.6	7.18	0.06	56.5	46.2	1.70	2.16
Default graph(Uniform)	96.1	53.5	7.19	0.07	38.2	49.6	0.50	2.34
50000-Node random graph(Zipf)	87.6	47.0	5.68	0.01	57.6	45.7	1.38	2.11
10000-Node PLAW graph(Zipf)	76.1	14.9	1.76	0.18	31.6	12.0	0.49	1.11

Table 2: Results for more graphs in the static case

Comparison metric	*s-APS*			*GUESS*		
	Succ(%)	Mesg	Hits	*Succ(%)*	Mesg	Hits
Messages per Search	97.7	**16.3**	5.22	63.9	**16.1**	1.28
	98.6	**22.0**	7.01	65.6	**22.2**	1.87
	99.7	**33.2**	11.39	84.0	**33.1**	2.55
Hits per Search	81.0	3.2	**1.33**	63.9	16.1	**1.28**
	94.6	8.7	**3.42**	86.4	45.0	**3.70**
	97.9	16.5	**5.42**	94.5	65.1	**5.60**

Table 3: Comparison with a GUESS implementation on a random hybrid graph

performance in one of two important metrics: Messages and hits per query. The results are presented in Table 3 and the comparison metric is typed in boldface. For similar message consumption, our scheme exhibits higher success rates and delivers 4 to 5 times more results. For similar hits per search, our scheme produces 4 to 5 times fewer messages and always outperforms *GUESS* in accuracy.

4.3 Dynamic Network Behavior

Robust behavior (fault-tolerant and adaptive operation in dynamically changing environments) should be a fundamental characteristic of every search scheme. *APS* maintains high levels of robustness for the following reasons: The query forwarding process is probabilistic, which means that nodes with the largest values do not get necessarily chosen. This is important as object locations may change frequently and wrong choices may occur in the beginning. No neighbors are excluded because of a low probability and node failures cannot interfere with the algorithm's normal operation[4]. In any case, k walkers will be deployed from the requesting node and no more traffic for index maintenance will be put on the network. Our algorithm also utilizes its *unlearning* feature, which enables walkers to be redirected if previously recovered objects are missing. Finally, the probability of query failure is greatly

[4]Unless the requester's neighbors are fewer than k, in which case all of them are chosen

Method	*s-APS*				*Random Walks*			
	Succ(%)	Mesg	Hits	Dupl	*Succ(%)*	Mesg	Hits	Dupl
Default graph(Zipf)	90.9	50.2	4.41	0.08	52.1	39.7	1.10	2.19
Default graph(Uniform)	94.1	58.5	4.22	0.10	32.3	41.8	0.39	2.38
50000-Node random graph(Zipf)	79.3	48.4	2.40	0.02	55.6	39.5	1.29	2.12
10000-Node PLAW graph(Zipf)	67.6	13.0	1.11	0.54	21.0	9.0	0.25	0.72

Table 4: Results for more graphs in the dynamic case

reduced because of the large percentage of successful walkers. The changes in topology or object locations must simultaneously affect all successful paths in order for a miss to occur.

In this section, we present simulation results with rapidly changing network topology, as we believe is the case with current P2P networks. Table 4 presents the most representative results in direct comparison to the static case.

First, we compare the two schemes on the default parameters. Our technique delivers amazingly high quality results. It shows a mere 3% decrease in accuracy and around 2.7 fewer hits per query, while the rest of the results are similar to the static run. The results for the uniform distributions on the default graph are qualitatively similar.

In the 50000-node random graph, we notice the success rate is about 8% lower from the static case, while the number of discovered objects is almost halved. The differences in performance become striking in the 10000-node power-law network. *s-APS* delivers as many as 4 times more results and exhibits a success rate three times bigger than *Random Walks'*. The success rate for *s-APS* drops by around 9% and discovered objects decrease by 37%, which seems reasonable to the static numbers if we reckon the rapidly changing topology and the graph's structure as described before.

In these simulations, our algorithm kept its message production at the same levels with the static runs, wasting at most 6 extra messages per search, a direct proof that it does not impose more burden on network traffic. The metric that is reasonably affected is the number of hits per search, as some paths to discovered objects frequently "disappear". As expected, the success rate shows only a small decrease, which ranges from 2% to 9%. Our scheme exhibits remarkable robustness even in such fast-changing environments. *Random Walks* shows a small decrease in performance, as walkers are not directed according to object locations, but randomly across the network.

Finally, a noteworthy observation; in choosing the total number of queries per simulation, we aimed for simplicity and realism. As we demonstrated, *APS* would benefit even more by increasing the number of queries per simulation.

5 Conclusions

In this work we introduced *APS*, an efficient, scalable and adaptive technique for Peer-to-Peer search. Our algorithm is suitable for P2P systems, where network traffic needs to be minimized, user-satisfaction and search accuracy must be high and co-operation between peers is important. *APS* utilizes probabilistically directed walkers for bandwidth-efficient search in *unstructured* P2P networks. Using its index scheme, *APS* incorporates information from past searches to enhance future performance. This also allows for fast, joint learning, since many nodes participate in the process initiated from a single node's request. Peers are required to keep indices only relative to their neighbors, while no message exchange is necessary for any dynamic network event, local or global. We also described two improved versions of our algorithm: *s-APS* further reduces message consumption by choosing the best update policy for each request, while *w-APS* significantly increases the number of objects discovered at short distances.

Our simulations on a variety of topologies demonstrated the versatility of the proposed technique. Results showed that *APS* achieves high performance at very low cost. It discovers 4 times as many objects and delivers very high success rates compared to the *Random Walks* and *GUESS* algorithms. It also proved to be as efficient as the random algorithm and much more efficient than *GUESS*, regarding message consumption. Another important result is that *APS* enhances its performance with more queries. Finally, we demonstrated our algorithm's ability to maintain these features even in rapidly changing environments, exhibiting a high degree of robustness and adaptation.

Acknowledgments

We thank Bobby Bhattacharjee and Antonios Deligiannakis for their helpful discussions and feedback.

References

[ALPH01] L. Adamic, R. Lukose, A. Puniyani, and B. Huberman. Search in power-law networks. *In press, Phys. Rev. E*, 2001.

[AUD] Audiogalaxy website: http://www.audiogalaxy.com.

[CGM02] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 23–34, July 2002.

[CLL02] J. Chu, K. Labonte, and B. Levine. Availability and locality measurements of peer-to-peer file systems. In *Proceedings of SPIE*, July 2002.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science*, 2001.

[DF] S. Daswani and A. Fisk. Gnutella UDP extension for scalable searches (GUESS) v0.1.

[DFM01] R. Dingledine, M. Freedman, and D. Molnar. The Free Haven Project: Distributed Anonymous Storage Service. *Lecture Notes in Computer Science*, 2001.

[DKK+01] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area cooperative storage with CFS. In Greg Ganger, editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

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

[JCJ00] C. Jin, Q. Chen, and S. Jamin. Inet: Internet Topology Generator. Technical Report CSE-TR443-00, Department of EECS, University of Michigan, 2000.

[KAZ] Kazaa website: http://www.kazaa.com.

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[KGZY02] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 300–307. ACM Press, 2002.

[LCC+02] C. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS*, 2002.

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

[NET] .NET home page: http://www.microsoft.com/net/.

[RD01a] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2001.

[RD01b] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[RFH+00] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, University of Berkeley, CA, 2000.

[RK02] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proceedings of INFOCOM*, 2002.

[SAN] The impact of file sharing on service provider networks. An Industry White Paper, Sandvine Inc.

[SET] SETI@home web site: http://setiathome.ssl.berkeley.edu/.

[SMK+01] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, 2001.

[SR] A. Singla and C. Rohrs. Ultrapeers: Another step towards gnutella scalability.

[Sto] M. Stokes. Gnutella2 specifications part one: http://www.gnutella2.com.

[WRC00] M. Waldman, A. Rubin, and L. Cranor. Publius: A robust, Tamper-Evident, Censorship-Resistant web publishing system. In *Proceedings of the 9th USENIX Security Symposium (SECURITY)*, August 2000.

[YGM02] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *22nd International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[ZCB96] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, March 1996.

[ZKJ01] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, 2001.