ASAP: An Advertisement-based Search Algorithm for Unstructured Peer-to-peer Systems

Peng Gu, Jun Wang
Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL 32816
{penggu,jwang}@eecs.ucf.edu

Hailong Cai
1600 Amphitheatre Parkway
Google Inc, Mountain View, CA 94043
hailong@google.com

Abstract— **Most of existing search algorithms for unstructured peer-to-peer (P2P) systems share one common approach: the requesting node sends out a query and the query message is repeatedly routed and forwarded to other peers in the overlay network. Due to multiple hops involved in query forwarding, the search may result in a long delay before it is answered. Furthermore, some incapable nodes may be easily overloaded when the query traffic becomes intensive or bursty.**

In this paper, we present a novel content-pushing, Advertisement-based Search Algorithm for unstructured P2P systems called ASAP. An *advertisement* (ad in brief) is a synopsis of contents a peer tends to share, and appropriately distributed and selectively cached by other peers in the system. In ASAP, nodes proactively advertise their contents by delivering ads, and selectively store interesting ads received from other peers. Upon a request, a node can locate the destination nodes by looking up its local ads repository, and thus obtain a one-hop search latency with modest search cost. Comprehensive experimental results show that, compared with traditional query-based search algorithms, ASAP achieves much better search efficiency, and maintains system load [1] at a low level with small variances. In addition, ASAP works well under node churn.

Index Terms— **Peer-to-peer, advertisement, ASAP, search, unstructured P2P**

I. Introduction

How to develop efficient content location schemes remains to be one of the foremost challenges in P2P systems. The past few years have seen many search algorithms presented and studied, and even more being developed. In unstructured P2P systems, most of existing query-based search algorithms share one common approach: upon a search request, a node sends out a query and the query is repeatedly routed and forwarded to other peers in the overlay network. This leads to the following limitations.

1) When a query travels multiple hops, it may take an arbitrarily long time for the query to be answered. Additionally, the search process incurs multiple query-related messages across the network. As a result, a significant amount of both network bandwidth and computing power are consumed at all the involved nodes on the routing path.
2) Query-based searches can not persistently offer a high-quality service under dynamic environments. When the request workload fluctuates, the system load generated by query messages tends to vary sharply since each request may lead to many query messages. During rush hours when requests become bursty, a large volume of query messages may easily overwhelm some incapable nodes and thus throttle the system scalability.

Both aforementioned limitations stem from a nature of existing query-based search approaches: a potentially excessive usage of queries, which usually leads to long search latency and high system load. One possible solution is to process a search request in one hop and thus reduce the search cost to a deterministic constant. The past few years have seen several efficient one-hop lookup algorithms, such as the One-hop routing scheme [14] and Beehive [22]. However, these schemes only work on structured overlays. Since most commercial P2P systems are unstructured, it is a more important but challenging task to realize one-hop search in unstructured P2P systems.

In unstructured P2P systems, we find that it is hard to achieve the goal by following the traditional query-based, content-pulling approach. There are two reasons: *peer passiveness* and *overlay unstructuredness*. In the query-based approach, most peers sit silently and passively wait for requests to arrive. Upon a request, the peer knows little about where to find the content in the unstructured overlay. Therefore, many query messages are generated for the purpose of object location, which is subject to long latency and high overhead. To completely solve the problem, solutions need to be sought out in a new direction.

In this paper, we present a content-pushing, Advertisements-based Search Algorithm for unstructured P2P systems (called *ASAP*), which aims to achieve a *one-hop latency* with *modest search cost* in most cases. An *advertisement* (or *ad* for short) is a synopsis of contents a peer tends to share, appropriately distributed and selectively cached by other peers in the system. Rather than waiting for the peers to *reactively* send out query messages, we argue that it is more appropriate for the system to *proactively* prepare the interested content indices *before* searches are initiated. To do this, each node in ASAP proactively advertises its shared contents by delivering an ad—a compact information vehicle with high-level semantics, and selectively stores interesting ads received from other peers. Upon a request, the node simply looks up its local ads repository for results. In this way, ASAP obtains optimal one-hop search performance with modest search cost. In addition, the system overhead regarding the ad delivery can be controlled at an acceptable level by imposing a total budget limit used in reference [12].

ASAP offers several benefits for searching contents in unstructured P2P systems. *First*, from the end user's point of view, ASAP services search requests in very short latency.

[1]By *system load*, we refer to all P2P traffics triggered by external events such as a search request. This does not include the keep-alive messages between peers as they are internally used to maintain overlay connectivity. Downloading traffic is not counted because it is out of the scope of content location and unavoidable in any content-sharing P2P system.

From the system's point of view, ASAP maintains a low system load with small variance, showing the good potential of smoothing out the bursty traffic. *Second*, unlike traditional query-based search schemes, the service quality is independent on the popularity of contents. Because of the proactive content pushing technique, both hot and cold documents are treated equally. *Third*, ASAP works well on purely decentralized P2P systems, and does not require the presence and willingness of powerful nodes that act as super peers.

Comprehensive simulation results show that, compared with representative query-based approaches, ASAP improves the search performance by more than 62% in terms of response time and slashes the search cost by 2 to 3 orders of magnitude in terms of bandwidth consumed in a search. In addition, ASAP keeps the system load 2 to 5 times lower, and experiences only minor load variations.

II. Related Work

Traditional P2P search schemes, such as flooding, random walk [5], [11], [21], and routing indices [6] rely on queries for content location. The search performance has been continuously improved by new alternatives. In structured P2P systems [16], [23], a query reaches its destination within $O(logN)$ hops, and an object publish/removal also requires $O(logN)$ messages of overlay maintenance. DHTs achieve good search performance by deterministically routing queries to specific nodes where the objects are saved. As studied by Blake *et al.* [1], however, DHT overlays are constructed in such an algorithmic fashion that contents close in the ID space are published to a randomly picked node with the closest nodeId, while the user at that node may have no interest in these data replicas at all. Consequently, queries are still needed for users' requests, and this scheme is also subjected to the problem of arbitrarily long latency and labile system load.

To further improve the search efficiency, researchers have proposed several one-hop lookup algorithms upon DHTs. Based on Chord, Gupta *et al.* [14] proposed a one-hop lookup scheme for P2P overlays, in which each node maintains accurate routing tables with complete membership information. For the purpose of disseminating information about membership changes, the system requires relatively powerful and stable nodes to act as slice and unit leaders. Following another direction, Ramasubramanian *et al.* [22] presented Beehive, a proactive replication framework that delivers O(1) lookup performance for common Zipf-like query distributions. Beehive continuously monitors the changes of content popularity and query distribution, and quickly adapts its performance to dynamic environments. Presented by Li *et al.* [17], Accordion automatically tunes itself according to the operating environment, aiming to persistently deliver good performance in terms of lookup latency by adaptively adjusting the routing table size. Kelips [15] probabilistically offers O(1) lookup performance by dividing the network into $O(\sqrt{N})$ groups of $O(\sqrt{N})$ nodes, replicating every object on every node within an group, and using gossip to propagate updates.

Content pushing is widely used in pub-sub systems [27], where publishers generate events that are consumed by subscribers. Unlike P2P systems, Pub-sub systems bear a distinct system architecture in that they rely on a dedicated network of routing brokers working exclusively for event propagation. PlanetP [8] employs a gossiping layer to globally replicate a membership directory and content indices. While the search performance was reported promising, the system load tends to be high due to the global gossiping. This could limit the system scalability.

III. ASAP Design

We present the detailed design of ASAP in this section, starting with an explanation of the design rationale in Section III-A. Then we discuss the ad representation in Section III-B. The ASAP search algorithm is described in Section III-C in detail.

A. Design rationale

In this paper, we leverage the idea of preparing indices beforehand for unstructured P2P systems. Instead of placing them in an "ID-matching" way as used in DHTs, we develop a more aggressive scheme that pushes the content indices to their potential consumers, such that user requests can be resolved by simply looking up local indices. Borrowing ideas from the real life, we call these indices advertisements, which play an important role in our real life. People receive a lot of ads from a variety of sources, such as mailing, TV, radio and posters. With different background and specific interests, people collect, keep or remember some of the ads that may be useful to them. When having a request, they just find or recall the related ad, go directly and get the product or service. Clearly this is a short-cut than blindly going out to search for it. Following the same philosophy, we design ASAP for efficient content location in unstructured P2P systems.

ASAP is designed based on four observations. *First*, query messages contribute to a large portion of network traffics in today's P2P systems, and are likely to continuously increase with the emergence of new applications. The measurement study by Gummadi *et al.* [13] reports the finding of nearly 100 million transactions over a period of 200 days in a 24,578-user Kazaa network. This corresponds to an average of 6 transactions undergoing every second, and the number of requests is even larger because not all requests succeed with a download.

Second, in content sharing P2P systems, the arrival rate of search requests tends to fluctuate. Several studies have shown the presence of daily patterns in user requests [24]. This is reasonable since most requests are submitted in the daytime until early night. During this period, the number of requests in a unit of time is likely to be much larger than the average. Let us consider the same network as studied in [13], and assume a number of 20 user requests during the peak time. Given an average node degree as 5 and TTL set to 7, these requests may lead to an average of $20 \times (5-1)^7/24,578 \approx 13$ query messages handled at each node per second in a Gnutella-like system. In addition to other network traffics, such an heavy workload may easily overwhelm some incapable nodes with limited network bandwidth.

Third, although peers may come and go freely, contents shared on many nodes do not change very often, if ever. In P2P systems, most contents are shared as the nodes enter the network. And they usually do not further share the documents downloaded from other peers, while some sharing may be imposed during the downloading. This can be safely conjectured by the existence of a large portion of free-riders in the system [25]. In addition, the contents in P2P systems are unlikely to be altered because of natural immutability [13].

Fourth, it is known that interest clustering is common in P2P systems [10] and it has been successfully exploited in prior work like SON [7] and SSW [18]. It is expected that many peers share common interests and the variety of each peer's interests is limited. Furthermore, most peers are unlikely to change their interests very often assuming a peer only corresponds to one user. In fact, node interest clustering and stability properties are two important assumptions based on which ASAP is designed.

While the search requests continuously increase and fluctuate, the contents are relatively stable as long as the system has warmed up. This motivates us to design a cooperative system in which peers proactively distribute and cache content indices. A modest investment on the indices distribution and preparation is well amortized to the service of a large number of user requests.

In an "optimal" approach, we may assume a system in which every node maintains a copy of complete content indices. If the indices are always up-to-date, all searches can be answered in 1 hop by local lookups. In practice, however, the index maintenance overhead in such a system would be prohibitively expensive, in terms of both network bandwidth and storage space on each node. Therefore, to develop a practical content-pushing, ads-based search algorithm, the biggest challenge is how to make the ad preparation and distribution efficient such that the index maintenance cost is kept reasonably low, and effective so that most local lookups get a hit. ASAP addresses these issues by appropriately conducting ad representation, issuing, forwarding, updating and refreshing.

B. Ad representation

In ASAP, an ad is comprised of four components: a node identity I, a piece of content information denoted by C, a set of topics T covered by the node, and a version number v. Thus an ad a is denoted as a tuple (I, C, T, v). The node identity can be the IP address along with a machine name, or a user account in case of dynamic IP address. With regard to content information, ASAP predefines three types of ads: *full ad* with complete indices of a peer's contents, *patch ad* with incremental index changes since the last update, and *refresh ad* with empty content information. The version number is a 16-bit integer used for consistently merging index changes. More details on this can be found in the next section.

The content information in a full ad summarizes all the contents shared on a node by using Bloom filter [2], [9]. Bloom filter is a hash-based data structure representing a set to support membership queries, and has been widely used in P2P system designs [3], [8]. The membership test returns false positives with a predictable probability but never returns false negatives. Assume D_p is the set of documents shared on node p, and $K_p = \{kw \in d_i | d_i \in D_p\}$ is the set of keywords that appear in any document in D_p, where kw is a keyword that appears in document d_i. The content filter of node p is initialized by hashing all the keys in K_p and setting the corresponding bits. Free-riders have a null content filter, thus having nothing to advertise.

Given a set of predefined hash functions, we can obtain the minimum probability of false positive as $p_{min} = (\frac{1}{2})^k = (0.6185)^{\frac{m}{n}}$, where k is the number of hash functions, m is the filter length and n is the set size. For example, with $k = 8$, the smallest false positive rate is 0.39%, and it demands 11.54 bits per element. This minimum probability of false positive imposes a requirement on the ratio of filter length to the set cardinality, that is, the average number of bits per element. In unstructured P2P systems, peers share contents at their own will and thus have different keyword set sizes. As a result, the desired false positive rate requires a different minimum length of content filter for each peer. There are two approaches to address this issue. One solution is to use the Bloom filters with fixed length for all peers, which is determined as $m = \frac{nk}{\ln 2} = \frac{|K_{max}|k}{\ln 2}$, where K_{max} is the largest keyword set among all the peers. The other approach is to use variable filter lengths. Suppose all nodes agree on a set of universal hash functions $\{h_1, h_2, ..., h_k\}$ and a pool of available filter lengths. Each node p chooses a minimum filter length that is greater than $\frac{|K_p|k}{\ln 2}$. When mapping or querying an item on a filter F with length $l(F)$, we can use a set of hash functions ranging from 0 to $l(F)-1$, for example, by defining them as $\{h'_1, h'_2, ..., h'_k\}$, where $h'_i = h_i \ mod \ l(F)$.

The first approach is simple and effective. But when some peers share much more contents, the filter length may have to be increased unless some load migration mechanism is used to prevent the maximum keyword set from growing. On the other hand, the variable filter length releases the constraint on the maximum keyword set and utilizes the space more efficiently. However, it complicates the system design in other aspects. For example, a node may have to compute the filter multiple times using different lengths for a search request. In this paper we choose Bloom filters with fixed size for two reasons. *First*, it is simple since only one set of hash functions are used everywhere. *Second*, it suffices for current P2P applications since the sizes of the keyword sets are not arbitrarily large [3]. With $|K_{max}| = 1,000$ and $k = 8$, the minimum length of a filter with the smallest false positive rate is $m = \frac{1,000 \times 8}{\ln 2} = 11,542$ bits $= 1.43 \ KB$.

For those peers who share few files and keywords, we use a compressed representation of the filter as a collection of 2-tuples (i, x), which means that the i^{th} bit is set for x times. Only the first number in each tuple is transmitted over the network. Similarly, an ad patch for content filter changes is implemented by a list of changed bit locations in the filter.

To determine the topics of an ad, we predefine a universal set U of all possible topics in the system, and apply classifications to the contents. We assume each document d belongs to a topic $t(d) \in U$, and each node p has a set of interests $I(p) \subseteq U$. For example, in a music file sharing network like Napster, music files are classified into tens of topics, such as pop, country and jazz. A node may be interested in pop and jazz but indifferent in any other types. Therefore, the topics of an ad a (no matter which type this ad is) from node p is denoted as $T(a) = \{t(d) | \forall d \in D_p\}$. A node q is interested in ad a if there is non-empty intersection between $T(a)$ and $I(q)$, where $I(q)$ is the set of q's interests. The document classification technique is matured in information retrieval field and out of scope of this paper.

C. ASAP search algorithm

In general view, search by flooding drives queries towards data, and DHT-based search moves both queries and data, causing them to meet at a rendezvous in the network [20]. As for ASAP, ad delivery moves data indices (*i.e.*, ads) towards interested nodes so that a later search is executed mostly on the local node. Figure 1 gives an illustration of the search procedures in the three categories respectively.

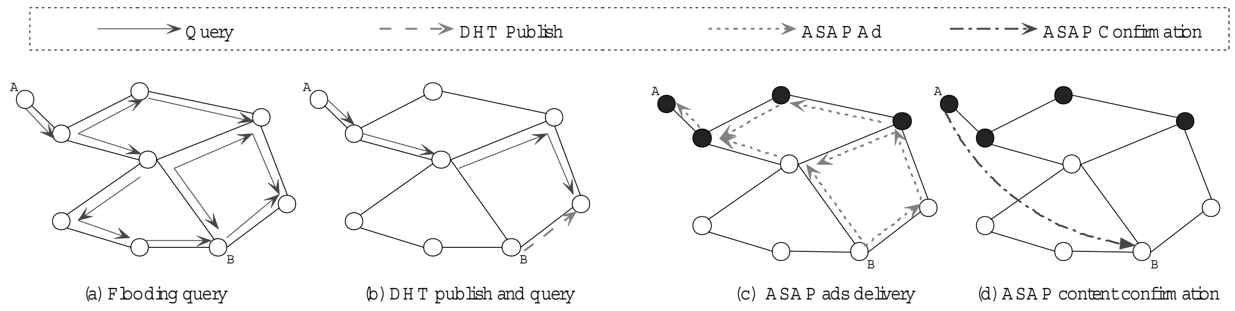

Fig. 1. The search procedure of a request from node A for a document shared by node B. The object publishing in (b) and ad delivery in (c) are done before the search begins. Shaded nodes in (c) and (d) are interested in the ad issued by node B. In ASAP, node A looks up local ads cache and directly contacts node B for content confirmation.

TABLE I
ASAP SEARCH ALGORITHM IN PSEUDO-CODE.

```
ASAP_search (Request: r)
//search algorithm running on node $p$ with ads cache \$
{
    $K \leftarrow getSearchTerms(r)$
    **for** each ad $a \in \$$
        $F \leftarrow getContentFilterFromAd(a)$
        **if** $match(K, F) =$ **true**
            $S \leftarrow getAdSourceNode(a)$
            send confirmation message to node $S$
    **if** more responses needed
        **for** each neighbor $i$
            $A \leftarrow requestAdFromNeighbors(i, h, I(p))$
        **if** $A = \emptyset$ **return**
        **for** each ad $a' \in A$
            $F' \leftarrow getContentFilterFromAd(a')$
            **if** $match(K, F') =$ **true**
                $S' \leftarrow getAdSourceNode(a')$
                send confirmation message to node $S'$
        $\$ \leftarrow \$ \cup A$
}
```

Given a request, a node p firstly looks up its local ads repository, and tries to find matching ads that contain the search terms (an ad is considered a match if the Bloom filter returns true for all the query terms). A match by an ad a_q (q is the source of this ad) indicates that node q has the requested objects. However, this may not be sufficient for a search in some cases. For example, node q may have multiple documents, each containing one or multiple of the search terms, while none of them have all of them. On the other hand, node p may expect documents to contain all or most of the terms. In addition, the ads are represented by Bloom filters in which false positives may occur, and the ad source node may be offline at the request time. For these reasons, node p needs to send the request to node q for content confirmation, and after a positive match the search is completed with the cost of one hop communication [2]. Table I shows the ASAP search algorithm in pseudo code. In $A \leftarrow requestAdFromNeighbors(i, h, I(p))$, h is hop number and $I(p)$ is the set of node p's interests.

With the help of ads cache lookup, most search requests are expected to be answered in one hop. However, if no match is found, or more responses are needed, then node p sends out ads request messages to its neighbors within a hop distance of h. These neighbors reply to p with their cached ads that contain topics overlapping with p's interests. In order to control the network bandwidth consumption, we limit the ads request scope by setting the distance h to a small value, e.g., 1 by default. After this, the search is repeated by looking up the replied ads for more possible hits. In essence, this is the same ads requesting process as the one when a brand new node joins. If a node stays offline for a long time and then rejoins, the ads in its cache could be mostly out of date. This ads request method enhances the chances to serve requests from these nodes.

For most requests from nodes that maintain many ads, only one hop communication is needed for a search, delivering an optimal search performance. In the mean time, the search cost only includes content confirmation messages, and only the initiating and destination nodes are involved in the search process. By moving ads towards their potential consumers beforehand, ASAP trades the ad preparation and distribution cost for a high search efficiency, and for most search requests, offers one-hop search performance with modest search cost. Moreover, ASAP bounds the system load at a low level and maintains small variances under the stress of extensive requests.

IV. Evaluation Methodology

We develop a trace-driven simulator to evaluate the performance of ASAP compared with several representative unstructured search algorithms. We describe our experimental methodology in this section and present the simulation results and analysis in the next section.

A. Experimental framework

In the experiments we use an overlay network with 10,000 peers that are constructed upon the GT-ITM transit-stub model [26]. This model constructs a hierarchical Internet network with 51,984 physical nodes randomly distributed in an Euclidean coordinate space. We set up 9 transit domains, with each containing 16 transit nodes on the average. Each transit node has 9 stub domains attached. Each stub domain has an average of 40 stub nodes. Nine transit domains at the top level are fully connected, forming a complete graph. Every two transit or stub nodes in a single transit or stub domain are connected with a probability of 0.6 or 0.4 respectively. There

[2]It is possible to piggyback download requests along with these confirmation messages in implementation, such that a download may begin just after the confirmation. However, we are only concerned with the search process that is the focus of our paper.

is no connection between any two stub nodes in different stub domains. The network latency is set according to the following rules: 50 ms for inter transit domain links; 20 ms for links between two transit nodes in a transit domain; 5 ms for links from a transit node to a stub node; 2 ms for links between two stub nodes in a stub domain. Out of these 51,984 physical nodes we randomly select 10,000 P2P nodes and construct the logical topology. Notice that only some physical nodes participate in the P2P system but all of them contribute to the network latency.

Three logical topologies are used in our experiments: *random*, *powerlaw* and *crawled*. In the random topology connections are randomly created with an average node degree of 5. The node degrees in the powerlaw topology have the same average but follow a powerlaw distribution with $\alpha = -0.74$. The crawled topology is derived from a crawled Limewire network topology [19] with an average node degree 3.35.

We choose several representative search schemes, such as flooding, random walk and generalized search algorithm (GSA) as baselines [3]. The TTL for flooding is set to 6. For random walk, 5 walkers are used each running with $TTL = 1024$. Each query by GSA is assigned a budget of 8,000, which limits the total number of messages during a search process. By adopting different ad forwarding algorithms (flooding, random walk or GSA), we develop and examine three ASAP schemes: ASAP(FLD), ASAP(RW) and ASAP(GSA), respectively. Ad flooding in ASAP(FLD) also sets TTL equal to 6, and 5 walker are used in ASAP(RW). For ASAP(RW) and ASAP(GSA), the total budget for one ad delivery can be determined by the number of topics in the ad and a budget unit $M_0 = 3000$.

B. Trace preparation

Since there is no real-world trace publicly accessible that contains query and download history information needed in our experiments, we carefully rebuild such a trace by processing a content distribution trace of an eDonkey system obtained from [10]. The eDonkey trace, probed during the first week of November 2003, contains the names of 923,000 files shared among 37,000 peers. More analysis of this trace, such as file popularity distribution, can be seen in [4], [10]. This trace contains a snapshot of the system while we need a query trace. We conduct the following preprocessing to construct such a synthetic but reasonable query trace.

1) We randomly select 10,000 peers out of the 37,000 nodes observed in the content distribution trace. All documents shared on these peers are collected to form a universal content set D_{all}. Other peers and their contents are not considered in our experiments.
2) We classify all the documents in D_{all} into 14 categories according to their content semantics with an assumption that each document belongs to a single class. File content semantics are deduced from its name and extension. Figure 2 shows the number of nodes whose shared contents fall in each of the semantic classes.
3) These semantic classes also define the universal set of peer interests and ad topics. If a peer is not a free-rider, the set of its interests contains all the semantic classes of its contents. This set also comprises the topics of ads from this peer. The interests of free-riding nodes are assigned randomly. Figure 3 shows the distribution of the node interests, that is, the number of nodes with each interest.
4) With an assumption that a peer only asks for interesting documents, we create a synthetic trace containing 30,000 search requests, 10% of which are followed by a content change, such as a document addition or removal. The network dynamics are emulated by inserting 1,000 node join and 1,000 node departure events randomly in the trace.
5) We add a time stamp to each query event. The request inter-arrival time is modeled by a Poisson distribution with $\lambda = 8$ (Averagely about 8 requests enter the system per second).
6) When the trace is constructed, we feed it into each testing system, replaying the queries and collect the results.

[3]Hierarchical scheme with superpeers is not used as a baseline since it requires a different system architecture and the presence and willingness of powerful nodes to act as super peers. Moreover, ASAP can work well on hierarchical systems in which only super peers are responsible for ad representation, delivery, caching and processing.

V. Simulation Results

In this section, we present the experimental results obtained from trace-driven simulations. In all the experiments, we mainly focus on two aspects: search efficiency and system load as well as its variation.

A. Search efficiency

We compare the search efficiency of all search algorithms by measuring their search performance and cost when replaying the trace in each of the three overlay topologies. Search performance metrics include success rate and response time. The success rate is defined as the percentage of search requests that obtain at least one result. Notice that all the search requests are created such that there is at least one matching document existing in the system at the request time. The response time is averaged among all successful search requests. Since the processing time at a node is negligible compared to the network delay, we ignore the queuing delay and Bloom filter computation overhead when calculating the average response time. The search cost is measured in the average bandwidth consumed in a search process.

The search performance results are shown in Figure 4 in terms of search success rate and Figure 5 concerning the average response time. Compared to the baseline search schemes, ASAP consistently obtains both high success rate and low response time in all experiments. ASAP is able to offer a low response time about 62% to 78% shorter than that of flooding and GSA search algorithms. Among the three ASAP schemes, ASAP(FLD) shows the best performance since it delivers ads more broadly and extensively than the other two.

We can see that, in all three overlay topologies, ASAP achieves satisfactory success rate and very short response time, although the success rates are not always the best for powerlaw and crawled topology. By studying the eDonkey content distribution trace, we find that the average number of copies per document is around 1.28 and 89% files only have one copy in the whole network. This partially explains why the random walk scheme shows poor success rate and long response time, as it usually requires a high document replication ratio [11]. For the same reason, GSA also exhibits poor success rate, but its response time is comparable to flooding.

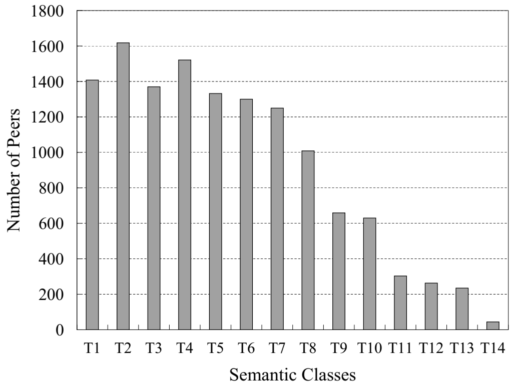

Fig. 2. The distribution of 14 semantic classes among the 10,000 peers used in our experiments.

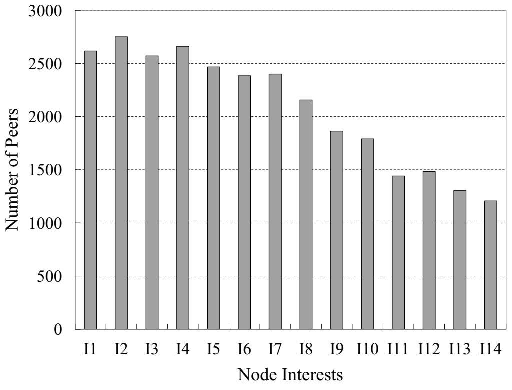

Fig. 3. The distribution of 14 node interests among the 10,000 peers used in our experiments.

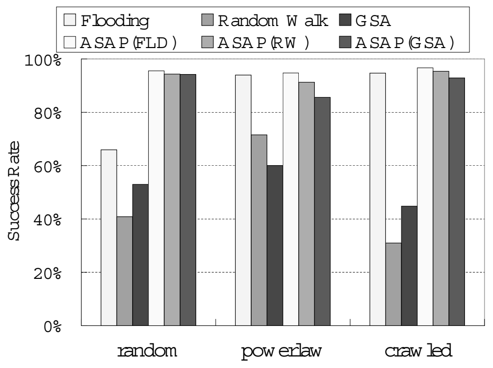

Fig. 4. Comparison of search performance in terms of search success rate.

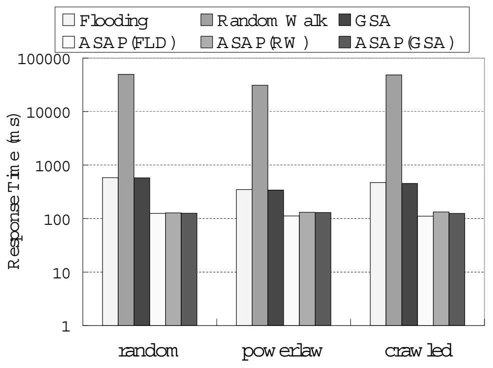

Fig. 5. Comparison of search performance in terms of search response time.

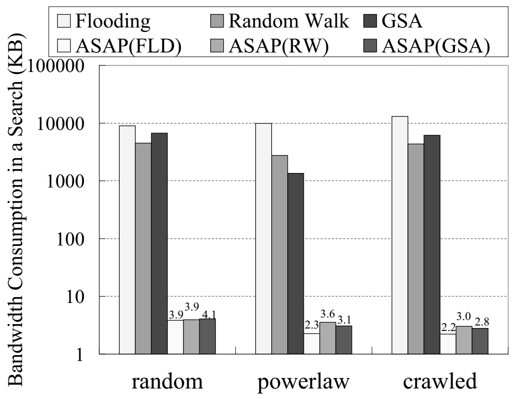

Fig. 6. Comparison of search cost in terms of bandwidth consumed in a search.

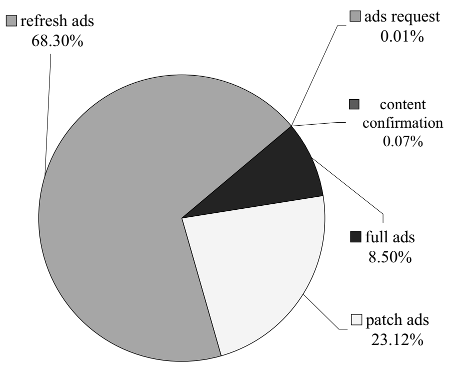

Fig. 7. Breakdown of ASAP(RW) system load in consumed bandwidth.

Figure 6 depicts the comparison of search cost in bandwidth consumption for all algorithms in the three overlay topologies. Notice that the search cost includes both content confirmation and ads request messages in ASAP, while in the baselines it refers to query messages only. The figure shows that ASAP satisfies most searches while consuming a small amount of bandwidth since only a few messages are generated in a search process. Compared with baseline algorithms, ASAP drastically reduces the search cost by 2 to 3 orders of magnitude. The significant improvement on search efficiency stems from the the beforehand ad preparation and distribution.

By maintaining a substantial amount of ads, a node is able to resolve most search requests by looking up local ads cache (and one message is needed for content conformation). Only if this fails, the node requests more ads from its neighbors within h hops. In order to obtain a satisfactory success rate, baseline search schemes have to generate multiple query messages and touch many nodes in the network. In ASAP, however, the ads are able to guide a request directly to its destinations, thus offering both optimal search performance and minimum search cost.

B. System load

While significantly improving the search efficiency, ASAP introduces extra maintenance overhead by ad deliveries. To sustain a good search efficiency, we need to limit the amount

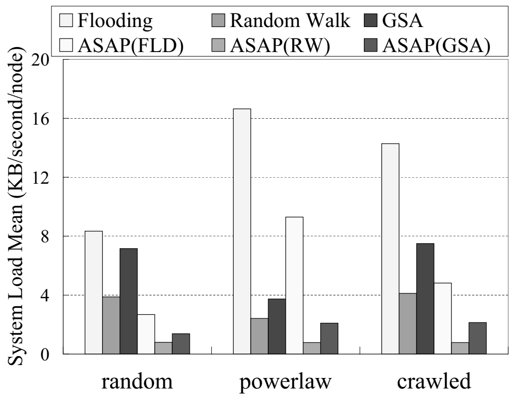

Fig. 8. Comparison of the average system load in bandwidth consumption.

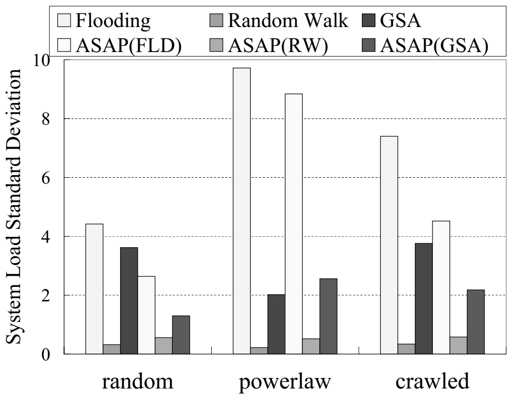

Fig. 9. Comparison of the system load variation (standard deviation).

of network bandwidth consumed by ad delivery, so that the total system load is controlled at an acceptable level. In the baseline systems, we count all the query messages as the system load, while in ASAP, all ad delivery messages are counted in addition to the search-related traffics including content confirmation and ads request messages. To compare the system load of ASAP against baseline schemes, we collect the total amount of bandwidth consumption and the number of live peers in the system at each second, and calculate the system load in terms of bandwidth consumption per node per second. Figure 7 shows the breakdown of the ASAP(RW) system load during the experiment. It is notable that the size of a full ad is larger than a query message because a full ad contains the Bloom filter, usually with a large size. However, after the system warms up, patch or refresh ads dominate since most ads are triggered by content updates or periodical deliveries. We can see that around 91% ads system load is from patch ads or refresh ads and full ads contribute 8.5%. The average system load and its standard deviation for each scheme (in each of the three topologies) are shown in Figure 8 and Figure 9, respectively.

To show the detail of the system load variations, we plot a graph of the bandwidth consumption per node per second as in Figure 10. Only a snapshot for a period of 100 seconds is presented for clarity. From this figure we can make two important observations. *First*, existing search algorithms lead to high system load while in ASAP(RW), the load is much lower. Compared with random walk which has the lowest system load among baselines, ASAP(RW) further reduces it by more than 81%. *Second*, the system load of ASAP scheme changes little while the baseline algorithms except random walk exhibit severe load fluctuations. At the peaks, the system load of flooding reaches more than 32 KB per node per second, but ASAP keeps it lower than 0.8 KB at most time. The minor vibrations in ASAP system load possibly result from a high search request rate at that time, since a content confirmation and/or a few ads request messages are generated during an ASAP search.

C. Comparison of search algorithms

From Figure 4 through 9, we compare the baselines and ASAP schemes systematically, and draw the following conclusions.

1) Flooding obtains good search success rate and fair response time, but consumes too much bandwidth due to many query messages generated in a search process. As a result, the system load is quite high and varies with a large variation.
2) Random walk controls the search cost within a prescribed limitation. Thus the system load is low and exhibits the smallest variation. However, the quality of search service is poor due to low success rate and long response time.
3) In random and crawled topology, GSA answers more queries successfully than random walk while consuming more bandwidth by query messages. It beats random walk in response time comparable to that of flooding. However, the system load also goes higher and experiences heavier vibrations.
4) In each overlay topology, all ASAP schemes demonstrate good success rate, very short response time and modest search cost. They differ mostly in the average system load and its variation. Particularly, ASAP(FLD) incurs relatively high load and sharp variations although it provides the highest success rate. ASAP(GSA) outperforms ASAP(FLD) by low system load and small load variation. The results show that ASAP(RW) maintains the lowest system load and variation, indicating the best choice among the three alternative ASAP schemes. For this reason, we choose ASAP(RW) as the default ASAP scheme in the following experiments.

We present results using only the crawled topology in the following since this topology is derived from a real P2P network topology.

VI. Conclusion

In this paper we propose ASAP, a new search algorithm for unstructured P2P systems. Nodes in ASAP proactively deliver content indices to interested peers, and each node caches and maintains a set of interesting ads. Given a search request, a node firstly looks up its local ads cache, trying to resolve it locally and answer it by just one hop content confirmation. Experimental results prove that ASAP can boost search efficiency by improving search performance and slashing search cost. In addition, ASAP smoothes out the system load and keeps it at a low level, and therefore, offering better system stability than existing search algorithms.

Acknowledgement

This work is supported in part by the US National Science Foundation under grants CNS-0646910, CNS-0646911,

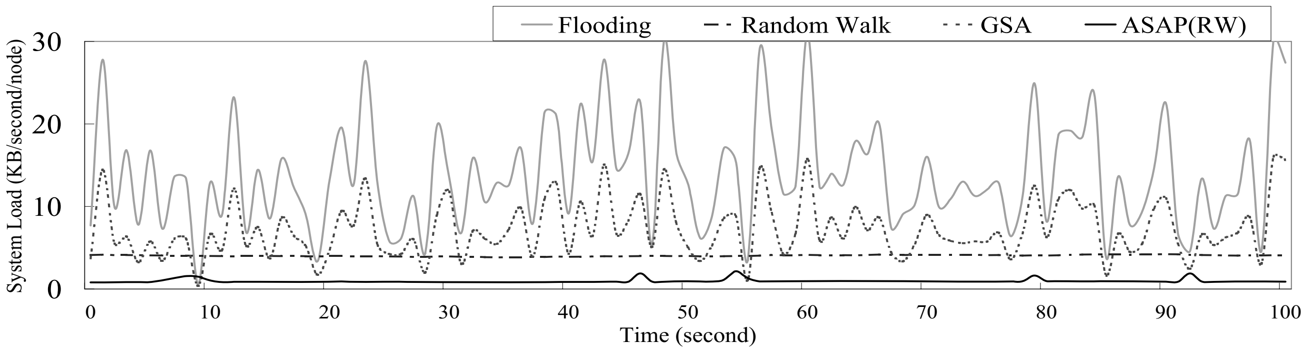

Fig. 10. Comparison of real time system load in the average bandwidth consumption per node per second when replaying the trace. To clearly show the load variation, we only plot the graph for 100 seconds.

CCF0621526 and CCF-0429995, and the US Department of Energy Early Career Principal Investigator Award DE-FG02-07ER25747.

References

[1] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 1–6, Lihue, Hawaii, USA, May 2003.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM, 13(7)*, pages 422–426, 1970.

[3] Hailong Cai and Jun Wang. Foreseer: A novel, locality-aware peer-to-peer system architecture for keyword searches. In *Proceedings of International Middleware Conference (Middleware 2004)*, pages 38–58, Toronto, Ontario, Canada, Oct. 2004.

[4] Miguel Castro, Manuel Costa, , and Antony Rowstron. Debunking some myths about structured and unstructured overlays. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, May 2005.

[5] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of ACM SIGCOMM'03*, pages 407–418, Karlsruhe, Germany, August 2003.

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

[7] Arturo Crespo and Hector Garcia-Molina. Semantic overlay networks for P2P systems. Technical report, Stanford University, 2003.

[8] Francisco Matias Cuenca-Acuna, Christopher Peery, Richard P. Martin, and Thu D. Nguyen. PlanetP: Using gossiping to build content addressable peer-to-peer information sharing communities. In *Proceedings of Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, pages 236–249. IEEE Press, June 2003.

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[10] F. Le Fessant, S. Handurukande, A.-M. Kermarrec, and L. Massoulie. Clustering in peer-to-peer file sharing workloads. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, San Diego, USA, February 2004.

[11] Christos Gkantsidis, Milena Mihail, and AMin Saberi. Random walks in peer-to-peer networks. In *Proceedings of IEEE INFOCOM'04*, pages 120–130, Hong Kang, March 2004.

[12] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Hybrid search schemes for unstructured peer-to-peer networks. In *Proceedings of IEEE INFOCOM'05*, Miami, FL, March 2005.

[13] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP-19)*, pages 314–329, Bolton Landing, NY, October 2003.

[14] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *Proceedings of the 9th IEEE Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 7–12, Lihue, Hawaii, USA, May 2003.

[15] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable p2p DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[16] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, Washington, USA, March 2003.

[17] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd symposium on networked systems design and implementation (NSDI'05)*, Boston, MA, USA, May 2005.

[18] Mei Li, Wang-Chien Lee, and Anand Sivasubramaniam. Semantic small world: An overlay network for peer-to-peer search. In *Proceedings of 12th IEEE International Conference on (ICNP'04)*, pages 228–238, Berlin, Germany, October 05 - 08, 2004.

[19] Limewire. http://www.limewire.org.

[20] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph Hellerstein. The case for a hybrid p2p search infrastructure. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, San Diego, USA, February 2004.

[21] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of 16th ACM International Conference on Supercomputing(ICS'02)*, pages 84–95, New York, NY, June 2002.

[22] Venugopalan Ramasubramanian and Emin Gn Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st symposium on networked systems design and implementation (NSDI'04)*, pages 99–112, San Francisco, CA, USA, March 2004.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001*, pages 161–172, San Diego, CA, August 2001.

[24] Stefan Saroiu, Krishna Gummadi, Richard Dunn, Steve Gribble, and Henry Levy. An analysis of internet content delivery systems. In *Proceedings of The Fifth USENIX symposium on Operating System Design and Implementation (OSDI)*, pages 315–328, Dec. 2002.

[25] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, San Jones, CA, Jan. 2002.

[26] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *Proceedings of the IEEE Conference on Computer Communication*, pages 594–602, San Francisco, CA, Mar. 1996.

[27] Yuanyuan Zhao, Daniel C. Sturman, and Sumeer Bhola. Subscription propagation in highly-available publish/subscribe middleware. In *Proceedings of International Middleware Conference (Middleware 2004)*, pages 274–293, Toronto, Ontario, Canada, Oct. 2004.