An advertisement-based peer-to-peer search algorithm

Jun Wang\textsuperscript{a,}\textsuperscript{*}, Peng Gu\textsuperscript{a}, Hailong Cai\textsuperscript{b}

\textsuperscript{a} School of Electrical Engineering and Computer Science, University of Central Florida, 4000 Central Florida Blvd., Orlando, FL 32816-2450, United States

\textsuperscript{b} Google Inc., 1600 Amphitheatre Pkwy, US-MTV-42 Room 229C, Mountain View, CA 94043, United States

\begin{abstract}
Most of the existing search algorithms for unstructured peer-to-peer (P2P) systems share one common approach: the requesting node sends out a keyword search 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 overloaded when the query traffic becomes intensive or bursty.

In this paper, we present a novel content-pushing, advertisement-based Search Algorithm for unstructured Peer-to-peer systems (ASAP). An advertisement (ad) 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 storing 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 at a low level with small variations. In addition, ASAP works well under node churn.

© 2009 Elsevier Inc. All rights reserved.
\end{abstract}

1. Introduction

How to develop efficient content location schemes remains 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. This is especially true when the query contains multiple keywords that make the search procedure blind.

2. Query-based searches cannot persistently offer a high-quality service under dynamic environments. When the request workload fluctuates, the system load\textsuperscript{1} 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 incapable nodes and thus throttle the system scalability.

We found that both aforementioned limitations stem from the nature of existing query-based search approaches: a potentially excessive usage of queries, which usually leads to long search latencies 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 [19] and Beehive [30]; however, these schemes only work on structured overlays. Since most commercial P2P systems are unstructured, it appears to be a more appealing task to realize one-hop search in unstructured P2P systems. Some works try to shorten the response time by self-organizing to topic-based clusters which could introduce high overhead and are hard to implement [40,37,28].

\textsuperscript{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.
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 from 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 content indices of interest (i.e., a compact data structure for indication of contents) 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 performs the look up using 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 to an acceptable level by imposing a total budget limit used in reference [16].

In this paper, we make the following contributions:

1. We propose a novel proactive ads delivery mechanism called ASAP to facilitate content searching in unstructured P2P networks.
2. ASAP is capable of smoothing out the search load burstiness in a dynamic request rate environment.
3. We compare the performance of six search algorithms and give comprehensive results.
4. We develop a sophisticated ads delivery algorithm to achieve good search success rate and maintain a lower system load at the same time.

ASAP offers several benefits for searching for content in unstructured P2P systems. First, from the end user’s point of view, ASAP services search requests within a very short latency. From the system’s point of view, ASAP maintains a low system load with small variation, thus showing good potential in smoothing out bursty traffic. Second, unlike traditional query-based search schemes, the service quality is independent of the popularity of the shared content. Because of the proactive content pushing technique, both frequently accessed (hot) and scarcely accessed (cold) documents are treated equally. Third, ASAP works well on purely decentralized P2P systems, and does not require the presence of powerful nodes willing to act as super peers.

Comprehensive simulation results show that, compared with representative query-based approaches, ASAP improves 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.

The rest of this paper is organized as follows: In Section 2, we review existing P2P search algorithms. We present our ASAP design in detail in Section 3. The evaluation methodology is explained in Section 4 and the simulation results are presented in Section 5 along with an in-depth analysis. Section 6 discusses several implementation issues of ASAP. Finally, we make some concluding remarks in Section 7.

2. Related Work

In centralized P2P systems, like Napster, queries are sent to central servers that save and maintain complete indices of all content in the system. Although the search efficiency is good (one hop latency and constant cost), this approach suffers from reliability and scalability problems since the central servers tend to be a single point of failure and a network bottleneck. Without centralized administration, unstructured P2P systems like Gnutella construct a totally decentralized overlay. Upon making a request, query messages are created and forwarded through message flooding or random walks, either blindly or biasedly [5,15]. These schemes have proved to be effective, but have a limited search efficiency because of the blindness in searches. As an alternative, several informed search algorithms have been developed to advocate maintaining distributed content indices among neighboring peers [8]. However, the expensive maintenance overhead prevents implementation of a large number of indices. As a result, searches in these systems still have to resort to flooding or random walks before the queries reach the desired content indices. Thus, they bear similar limitations of long search latency, and high search cost brought by queries. Some recent work [6] asks each node to index a random set of files in the network in order to allow every query to have a constant probability to be successfully resolved within a fixed search space. However, the files to be indexed are selected at random and the interests of nodes are not taken into design consideration, which leads to insufficient effectiveness.

Traditional P2P search schemes, such as flooding, random walk [5,15,29], and routing indices [8] rely on queries for content location. The search performance has been continuously improved by new alternatives. In structured P2P systems [21,31,33,38], a query reaches its destination within $O(\log N)$ hops, and an object publish/removal also requires $O(\log N)$ messages of overlay maintenance. Distributed hash table (DHT) networks achieve good search performance by deterministically routing queries to specific nodes where the objects are saved. As studied by Blake et al. [1], 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 node ID, while the user at that node may have no interest in these data replicas. Consequently, queries are still needed for users’ requests, and this scheme is also subject to the problem of arbitrarily long latency and variable system load.

To further improve the search efficiency, researchers have proposed several one-hop lookup algorithms based upon DHT networks. Based on Chord, Gupta et al. [19] 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. [30] 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. [24], 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 [20] probabilistically offers $O(1)$ lookup performance by dividing the network of size $N$ 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. In Kelips, continuous background communication with a constant overhead is used to maintain the index structure with high quality, as well as guarantee quick convergence after membership changes.
While our ASAP scheme is a more aggressive scheme that aims to significantly improve the search efficiency, it introduces extra maintenance overhead by ad deliveries in the background. To sustain good search efficiency, we employ different budget based search schemes (e.g., Generalized Search Algorithm [16]) to cap the amount of network bandwidth consumed by ad delivery, and thereby the total system load is controlled at an acceptable level.

Content pushing is widely used in publish-subscribe (pub-sub) systems [43], 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 [10] employs a gossiping layer to globally replicate a membership directory and content indices. While the search performance was reported as promising, the system load tends to be high due to the global gossiping. This could limit the system scalability. It may also be noted that several works have reported good results in a similar area of P2P Information Retrieval (IR) [7,39,13]. They focus on how to implement indexing from the IR perspective.

The main idea of this paper is somewhat overlapped with the event notification in content-based publish-subscribe (pub-sub) systems [43,34,41]. A pub-sub system ensures timely delivery of published messages from publishers to all interested subscribers. Publishers and subscribers obtain services by connecting to a network of routing brokers. ASAP is different from pub-sub systems in several aspects. First, a pub-sub system typically contains nodes playing different roles such as: publishers generating events, subscribers consuming events, and a network of routing brokers for event propagation. However, each node in ASAP could be an ad issuer, an ad consumer, or a router. Second, subscribers in pub-sub systems have to widely register their interests among the routing brokers and publishers in order to guarantee in-order, gapless event delivery services. ASAP differs in that nodes determine the interestingness of an ad only when it is received. Finally, events in pub-sub are consumed by subscribers as soon as the events are received, while in ASAP ad delivery is mainly a beforehand preparation for future requests.

3. ASAP design

We present the detailed design of ASAP in this section, starting with an explanation of the design rationale in Section 3.1. Then we discuss the ad representation in Section 3.2 and explore the ad delivery and caching mechanisms in Section 3.3. The ASAP search algorithm is described in detail in Section 3.4.

3.1. 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 DHT networks, 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 real life, we call these indices advertisements. People receive a lot of ads from a variety of sources, such as the mail, TV, radio and posters. With different backgrounds and specific interests, people collect, keep, or remember some of the ads that may be useful to them. When they have a request, they find or recall the related ad, and go directly to the source to get the product or service. Clearly this is a short-cut compared to blindly going out and searching. 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 traffic in today’s P2P systems, and are likely to continuously increase with the emergence of new applications. The measurement study by Gummadi et al. [18] reports a finding of nearly 100 million transactions over a period of 200 days between May and December of 2002 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 [35]. 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 [18], and assume a number of 20 user requests during the peak time. Given an average node degree as 5 and Time To Live (TTL) set to 7, these requests may lead to an average of $20 \times (5 – 1)/24 \times 578 \approx 13$ 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 content is 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 sharing of incomplete file portions during download are not appropriate to add to ads for search. The rational is that, when people search for a file, what they want in the search results are complete files instead of incomplete file pieces. So, it does not make sense to send out ads updates for the parts newly downloaded. And since files shared in P2P systems are usually large, and takes hours to days to complete downloading, the ad delivery interval is also expected in the same range, i.e., in terms of hours or days. In addition, the existence of a large portion of free-riders in P2P systems has already been observed in some studies [36]. Furthermore, the contents in P2P systems are unlikely to be altered because of natural immutability [18].

Fourth, it is known that interest clustering is common in P2P systems [12] and it has been successfully exploited in prior work like SON [9] and SSW [25]. 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 upon 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 making 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.
3.2. Ad representation

In ASAP, an ad is comprised of four components: a node identity \( l \), 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 \((l, C, T, v)\). 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 a bloom filter [2,11]. A bloom filter is a hash-based data structure representing a set to support membership queries, and has been widely used in P2P system designs. It is an enabling technique which can index 10,000,000 items using only 12 MB of memory with a false positive rate of 1% [3,10,23,32]. 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 \( p_{min} = \left( \frac{1}{2} \right)^k = (0.6185)^\frac{k}{n} \), 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 \( m = \frac{nk}{\ln 2} = \frac{K_{\text{max}}k}{\ln 2} \), where \( K_{\text{max}} \) 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 \( \{h_1, h_2, \ldots, h_k\} \) and a pool of available filter lengths. Each node \( p \) chooses a minimum filter length that is greater than \( \left\lceil \frac{kK_{\text{max}}}{\ln 2} \right\rceil \). 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 \( \{h_1, h_2, \ldots, h_k\} \), where \( h_i = h_i \mod l(F) \).

The first approach is simple and effective. But when some peers share significantly more content, 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 \( |K_{\text{max}}| = 1000 \) and \( k = 8 \), the minimum length of a filter with the smallest false positive rate is \( m = \frac{1000 \times 8}{\ln 2} = 11,542 \) bits = 1.43 KB.

For those peers which 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 \( T(a) = \{t(d) | yd \in D_p\} \). A node \( q \) is interested in ad \( a \) if there is no-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 the field of information retrieval and out of scope of this paper.

3.3. Ad delivery and caching

Central to the ASAP design, the ad delivery and caching mechanism determines the quality of search service and efficiency of system maintenance. There are three questions to answer here. (1) When does a node issue an ad into the network? (2) How to efficiently forward ads over the network? (3) How does a node maintain the received ads in its ads cache? Since free-riders share nothing, they will never issue an ad to the network. In the rest of the paper, any node issuing an ad is not a free-rider.

3.3.1. When to issue ads

There are three scenarios when an ad is issued.

- **Node joins.** When a brand new node joins the system for the first time, it issues a full ad to network. When a node rejoins, a refresh ad is issued instead. In order to avoid undesirable ads from transient nodes that stay online very shortly, we define a minimum uptime threshold, e.g., 1 min, before a node issues an ad.

- **Content changes.** An ad is issued immediately when there are changes in a node’s content filter, e.g., by document addition or deletion. This does not incur a lot of traffic with an assumption that nodes do not change their contents very often. If the updates are minor (i.e., only a small number of bits changed in the filter), a patch ad is issued. If the changes are so significant that the length of a patch becomes large, a full ad is issued, instead.

- **Periodical delivery.** Some nodes may not have content changes for a long time. Therefore, it is necessary for a node to periodically issue a refresh ad with empty content information if no ad has been sent out from this node during the past unit of time. This way, those peers that recently joined the system also have a chance to receive ads from this node. If interested, they will contact this node for a full ad with complete content information.

Although the content information in an ad could be full, incremental, or empty, all ads contain the topics \( T \) covered by the node and a 16-bit version number \( v \). The initial version number for a brand new node can be picked randomly. Each time a full ad or a patch ad is issued, the version number is incremented by 1.

\footnote{For example, if 8% bits in a 16,384-bit content filter have changed, each bit uses a 14-bit address and these changed bits require 1311 × 14 = 18,354 bits, even longer than the whole filter. In this paper, we set this threshold to 5% for a Bloom filter of length 2 KB.}
However, a refresh ad always reuses the current version number as there is no content updates. Since the version number increments only when a node joins or content changes, it takes a long time for this number to wrap around.

Notice that both node join and content changes trigger instant ad deliveries. The periodical ad delivery only occurs when a node has not issued any ad for a period of time. As a result, if the system undergoes a lot of node churn or content changes, the number of periodical ad delivery will be small. On the contrary, if there are few dynamic activities in the system, more periodical refresh ads will be issued to allow all nodes with different ages to receive their interesting ads in a timely fashion. In this paper, we set the ad delivery period to one hour by default based on experimental studies.

3.3.2. How to forward ads step by step

In order to measure the ad interestingness, we define the relevance of an ad to a node as the percentage of the node's interests that are covered by the ad's topics. In particular, let $T(a)$ be the set of topics in ad $a$, and $I(p)$ be the set of interests of node $p$, then the relevance of ad $a$ to node $p$ is $r_{a,p} = \frac{|T(a) \cap I(p)|}{|I(p)|}$. Node $p$ is interested in ad $a$ if $r_{a,p} > 0$. Notice that if $|T(a)| \geq |I(p)|$, that is, all of node $p$'s interests are covered by ad $a$'s topics, then the relevance $r_{a,p} = 1$.

In ASAP, ads are expected to be received by the peers who are interested in the covered topics. If we treat an ad as a query and the interested peers as the query destinations, then the ad delivery procedure is isomorphic to a query process in P2P systems. Their differences are twofold. First, unlike a query by which the source node seeks for interesting contents on other peers, an ad is to inform other peers about the contents available on the source node. In other words, ad delivery in ASAP is to do content pushing in advance, although with the same ultimate objective of efficient object location for P2P systems. Second, the ad delivery latency is inferior to query delay in importance. Therefore, existing query routing and forwarding schemes can be referred to develop the ad forwarding algorithms, but with a preference for low delivery cost rather than small latency. In this paper, we study three alternative ad forwarding methods by adopting different query algorithms. Other query algorithms may also be adopted if appropriate.

- **Flooding.** In the flooding policy, the content source node sends out an ad to each of its neighbors. Upon receiving an ad, each node forwards it to all of its neighbors except for the one from which the ad is received. Like in queries, we use TTL to bound the scope within which an ad travels.

- **Random walk.** Random walk can also be used for ad forwarding. In this policy, the source node sends out an ad several times, to a randomly picked neighbor each time. Upon receiving an ad, a node just forwards it to one of its neighbors randomly, except for the one from which the ad is received. The total cost of an ad delivery can be limited by setting a total budget $M$, which is evenly distributed among all the $n$ walkers, each with a TTL equal to $M/n$.

- **GSA.** Generalized Search Algorithm is a search scheme recently proposed by Gkantsidis et al. [16]. The basic idea is to finely control the search cost by budget and appropriately distribute the budget during query propagation. When using GSA for ad forwarding, the source node initiates an ad with a given budget $M$. Upon receiving an ad, each node decrements the received budget by 1, and then splits it when forwarding the ad to its neighbors in the following way: each neighbor granted with a budget proportional to the relevance of the ad to that neighbor, except for the one from which the ad is received.

It is important to calculate the total budget for an ad delivery by random walk or GSA. Since an ad summarizes the contents on a node, different ads carry different contents and correspond to distinct sets of interested nodes. Thus the budget $M$ for delivering an ad should be dependent on the number of interested nodes of this ad. By studying a real-world eDonkey [12] file trace used in our experiments, we find that the average number of nodes interested in an ad $a$ is generally proportional to $\sqrt{|T(a)|}$, where $T(a)$ is the set of topics covered by the ad, as shown in Fig. 1. For this reason, we set the budget for delivering ad $a$ as $M_a = M_0 \times \sqrt{|T(a)|}$, where $M_0$ is the default ad delivery budget unit, defined as a system parameter.

Among the three ad forwarding algorithms, random walk shows the best potential for several reasons. First, it is simple and easy to implement. In contrast, GSA requires that each node know the interests of its neighbors, while flooding does not have an accurate control on the whole ad delivery cost. Second, by random walk, an ad can reach distant peers since the number of walkers is limited and TTL tends to be large. When used in queries, a large TTL usually leads to long response time. But in ad delivery which is not sensitive to large latency, it enables an ad to reach peers many hops away. Third, by random walk, two ads consecutively issued from the same node are likely to reach different peers because of the randomness in ad forwarding. While by flooding or GSA, two consecutive ads from the same node tend to reach the same set of peers unless the overlay undergoes significant topological changes. For these reasons, we choose random walk as the default ad forwarding scheme, although all three are included in the experiments for comparison.

3.3.3. How to maintain the local ads cache

According to the user's interests, each node maintains a local ads cache. There are two ways by which a node receives ads, either through ad requests or by receipt of ads. When a brand new node joins, its ads cache is empty until it receives interesting ads. In order to collect ads quickly, the joining node would send out ad requests to its neighbors within a short distance of $h$ hops, and try to find interesting ads from their ads repositories. After this, the ads cache is only updated when an interesting ad is received.

Upon receiving an ad $a$, a node $p$ compares its own interests with the ad topics, and directly forwards it if it is not interesting. Otherwise this ad is useful. Table 1 shows the pseudo code when a node $p$ with an ads cache $S$ receives an interesting ad $a$ with version number $v(a)$. Whether $a$ is newer than $b$ or not (newer $\forall (a,b)$ can be determined by the version numbers as $v(a) > v(b)$ or $v(a) ≤ v(b)$ and $v(a) > v(b)$) since the version number might wrap around when it reaches $2^{16}$.

The ad merge can be done by incorporating the updates in patch ad $a$ to the cached full ad $b$ and then increasing the version number by 1.

Since the ads cache size is limited, a replacement policy is needed when a node receives more interesting ads than the cache

![Fig. 1. The average number of interested nodes for an ad $T(a)$.](image-url)
Table 1
Pseudo code for a node p with ads cache $s$ to receive an interesting ad a.

receive(Full_AD:a)
{
  $s ←$ getAdSourceNode(a)
  b ← getAd($s$, $s$)
  if b = null
    insertAd(a, $s$)
  else if newerThan(a, b)
    removeAd(b, $s$)
    insertAd(a, $s$)
}

receive(Patch_AD:a)
{
  $s ←$ getAdSourceNode(a)
  b ← getAd($s$, $s$)
  if b = null
    requestFullAd($s$)
  else if newerThan(a, b)
    if $v(a)=(i(b)+1)/2^{i(b)}$ - 1
      mergeAd(a, b)
    else requestFullAd($s$)
}

receive(Refresh_AD:a)
{
  $s ←$ getAdSourceNode(a)
  b ← getAd($s$, $s$)
  if b = null
    requestFullAd($s$)
  else if newerThan(a, b)
    requestFullAd($s$)
}

Fig. 2. 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 its local ads cache and directly contacts node B for content confirmation.

can accommodate. When a replacement is needed, ASAP finds the ad item which, according to its relevance and elapsed time since last update/refresh, seems to be the least valid compared to other items in the cache. For this reason, we measure the validity of an ad $a$ as $r(a)$, where $r(a)$ is the relevance of ad $a$ to the node and $t$ is the elapsed time since the ad $a$ is updated or refreshed on this node. The validity of an ad keeps decreasing until it is updated or refreshed. When a new interesting ad is received and the ads cache is full, the ad item with the smallest validity is selected for replacement. As a result, those ads whose source nodes have departed or failed will have smaller validity and be replaced by new ads as time goes by. Another scenario to invalidate an ad item is when its source node is found dead, in which case the ad is directly removed from the cache.

3.4. 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 [27]. As for ASAP, ad delivery moves data indices (i.e., ads) towards interested nodes so that later search is executed mostly on the local node. Fig. 2 gives an illustration of the search procedures in the three categories respectively.

Given a request, a node $p$ first 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 $q$ ($q$ 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. In $A ← requestAdFromNeighbors(i, h, l(p))$, $h$ is hop number and $l(p)$ 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

3 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 this paper. Table 2 shows the ASAP search algorithm in pseudo code.

Table 2
ASAP search algorithm in pseudo-code.

```java
ASAP_search (Request: r)
// search algorithm running on node p with ads cache $s$
{ K ← getSearchTerms(r)
  for each ad $a ∈ S$
    $F ← getContentFilterFromAd(a)$
    if match(K, $F$) = true
      $S ← getAdSourceNode(a)$
      send confirmation message to node $S$
  if more responses needed
    for each neighbor i
      $A ← requestAdFromNeighbors(i, h, l(p))$
      if $A = ∅$
        return
      for each ad $a' ∈ A$
        $F' ← getContentFilterFromAd(a')$
        if match(K, $F'$) = true
          $S' ← getAdSourceNode(a')$
          send confirmation message to node $S'$
      $S ← S ∪ A$
}
```
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 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 variations under the stress of extensive requests.

4. 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.

4.1. Experimental framework

In the experiments we use an overlay network with 10,000 peers that are constructed upon the Georgia Tech Internetwork Topology Models (GT-ITM) transit-stub model [42]. 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 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 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 parameter $\alpha = -0.74$. The crawled topology is derived from a crawled Limewire network topology [26] with an average node degree of 3.35.

We choose several representative search schemes, such as flooding, random walk and generalized search algorithm (GSA) as baselines. The TTL for flooding is set to 6. For random walk, 5 walkers are used each running with $\text{TTL} = 1024$. Each query by GSA is assigned a budget of 8000, 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 walkers 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 $M_0 = 3000$.

4.2. 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 [12]. 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 [4,12]. 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 $D_{\text{all}}$. Other peers and their contents are not considered in our experiments.
2. We classify all the documents in $D_{\text{all}}$ 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. Fig. 3 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-riding, 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. Fig. 4 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 1000 node join and 1000 node departure events randomly in the trace.
5. We add a time stamp to each query event. The request interarrival time is modeled by a Poisson distribution with parameter $\lambda$ (averagely about $\lambda$ requests enter the system per second). The value of $\lambda$ is from 1 to 10 in different experiments. These request arrival rates are comparable to the statistics by Gish et al. [14].

6. When the trace is constructed, we feed it into each testing system (which is allowed sufficient time in advance for the system to distribute the ads and stabilize) replaying the queries and then collect the results.

5. Simulation results

In this section, we present the experimental results obtained from trace-driven simulations. In all the experiments except in Section 5.4, the experiment setup is the same as described in Section 4.2, and we mainly focus on two aspects: search efficiency and system load as well as its variation.

5.1. 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 Fig. 5 in terms of search success rate and Fig. 6 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 of 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 [15]. For the same reason, GSA also exhibits poor success rate, but its response time is comparable to flooding.

Fig. 7 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, does the node request more ads from its neighbors within h.

---

5 Although we use the request events from the collection of Gummadi et al. [18], we adopt higher request inter arrival rates observed by more recent works.
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 destination, thus offering both optimal search performance and minimum search cost.

5.2. System load

While significantly improving the search efficiency, ASAP introduces extra maintenance overhead by ad deliveries. To sustain good search efficiency, we need to limit the amount 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 traffic including content confirmation and ad 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. Fig. 8 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% of ad based 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 Figs. 9 and 10, respectively.

Maintaining a small load variation is critical for P2P system scalability because the system must be able to persistently sustain a high quality service. In query-based search schemes, each search request triggers a lengthy, usually multiple-hop search process. When the request rate is high, the system load tends to increase and overwhelm incapable nodes. By proactively delivering ads and intelligently controlling the ad delivery cost, ASAP smoothes out the system load and keeps it at a relatively low level.

To show the detail of the system load variations, we plot a graph of the bandwidth consumption per node per second in Fig. 11. Only a snapshot for a period of 100 s 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 a majority of 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 ad request messages are generated during an ASAP search.

5.3. Comparison of search algorithms

From Fig. 5 through 10, 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 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 from 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 terms of average system load and its variation. ASAP(FLD) in particular, incurs relatively high load and sharp variations although it provides the highest success rate. ASAP(GSA) outperforms ASAP(FLD) due to 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 ASAP scheme variants. 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 experiments since this topology is derived from a real P2P network topology.

5.4. Environmental dynamics

It is critical for P2P systems to work well when the operating environment changes. The dynamism has three types of external sources: search request rate, node churn and content changes. We study each factor in the following.

5.4.1. Search request rate

We conduct experiments to compare the system load and load variation between the baseline schemes and ASAP when the search request rate $\lambda$ changes. Since the search requests are mostly served by looking up a node’s local ads cache plus a one-hop content confirmation, the search response time and the bandwidth consumption in a search do not change significantly in the following sensitivity studies. Thus, they are not shown in the figures for the rest of the paper.

Fig. 12 demonstrates the average system load when the search request rate increases. As the search request rate rises from 1 through 10, the average system load of the baseline systems increases significantly: from 3.3 to 32.96 for flooding, from 0.96 to 9.6 for random walk, and from 1.74 to 17.58 for GSA in terms of KB per second per node. In contrast, the average system load of ASAP only increases slightly from 0.66 to 1.16. The reason is that in ASAP, only one message is generated for a request in most cases, as opposed to a large number of query messages created in the query-based baselines.

The load variation is shown in Fig. 13, which implies that as more requests are submitted in a period of time, baseline schemes will see sharper load variations along with the increasing load mean. Among the baselines, random walk shows the smallest increase in the load standard deviation, from 0.1 to 1.08. As for ASAP, however, it is interesting to notice that the system load variation narrows as the search request rate increases. This is because when more requests are being handled, a larger portion of the system load comes from the content confirmation and ad request messages, both of which only incur a limited bandwidth consumption. Thus, the system load tends to gradually stabilize. As
5.4.2. Node churn

Peers in a P2P system may come and go at any time, and this may affect the system load of ASAP since each joining peer will trigger an ad delivery as long as it stays longer than the uptime threshold. We change the number of node joins and departures in the trace and replay it to study the effects of different intensity of node churn. Then we plot the change of search success rate and system load in Fig. 14 (standard deviation is abbreviated as S.D. in this figure and Figs. 15–18). It can be seen that with more node churns in the experiment, the system load as well as its variation increase slightly. The search success rate stays over 95% as more node churn occurs in the system. This is because ASAP serves search requests mostly by looking up its local ads repository, rather than sending out queries that could be seriously affected by overlay connectivity. When a lot of peers depart simultaneously, a node can still serve a request that asks for an object on a remote peer even if there is no existing path between these two peers in the overlay.

As a comparison, traditional systems produce many query messages for each search request, while the bandwidth consumed in a period of time is unpredictable.

5.4.3. Content changes

Another kind of dynamism that leads to ad delivery is the change of content filters because of document addition or deletion. For clear presentation, we define the content variability as the ratio of the number of document changes (addition or deletion) to the number of search requests in the trace. Next we run the experiment by varying the content variability, and present the results in Fig. 15. Since the ad could be updated either by a patch or by periodical refresh, there is a large probability for the ad to be updated before the inconsistency exhibits any negative impacts on the search performance. This explains why the search success rate does not degrade significantly as the content changes more frequently.

However, as the content variability increases, the system load climbs up along with noticeable increases in the load variation.
However, even with a content variability of 0.5 (that is, more than half of the search requests are followed by a content change, such as a document addition or removal, which is unlikely to happen in real systems), the average system load is only 2.06 and its standard deviation is 1.56. It is expected that if each search request is accompanied by a document change, ASAP may not show significant improvement over query-based approaches, since each ad delivery may lead to multi-hop message propagation, similar to the query messages in the baseline systems. A straight conclusion that can be drawn is that ASAP performs better if contents are not changed often, which is the case in P2P networks as previously explained in Section 3.1.

5.5. Ad delivery and caching efficiency

Based on ad delivery and caching, the system performance of ASAP largely depends on how efficiently each ad is delivered and how well each node receives and maintains its ads cache. This corresponds to the two requirements for ASAP to work well: (1) each ad delivery can reach a majority of peers that are interested, and (2) each node is able to timely receive and maintain most ads that contain interesting contents. We explore these two aspects in the following experiments.

5.5.1. Ad delivery efficiency

For each ad delivery, we define its node coverage as the percentage of interested nodes reached by the ad out of all the interested nodes for the ad. To ensure an efficient ad delivery, the ads are expected to reach as many interested nodes as possible, that is, to obtain a high node coverage. In the previous experiments, the average node coverage for an ad delivery is 79% by the default configuration. Because of a limited budget for ad delivery, an ad may not be able to reach all interested nodes. This problem is mitigated in that the following ads from the same source node may be able to reach other interested nodes in a short time.

The ad delivery budget unit is the most important system parameter that affects an ad's node coverage. We plot the impacts of ad delivery budget unit in Fig. 16. When the budget unit increases, an ad reaches more interested nodes, resulting in higher success rate; meanwhile, the system load is higher and shows heavier variations. When the ad delivery budget unit exceeds 3000, the average node coverage and search success rate are not improved much, but the system load and variation increase a lot.

5.5.2. Ads caching efficiency

Due to ASAP design, the search performance largely depends on how well the ads cache stores interesting ads from other peers before a request is submitted. It is clear that the more frequently ads are delivered, the more possibly a node may be able to receive ads it is interested in. Next we introduce the concept of a node's cache storage and study the impact of another system parameter, the ad delivery period. The cache storage of a node $p$ is defined as $\|S_p\|_{\text{Ball}}$, where $S$ is the set of source nodes whose ads are cached on node $p$, and $S_p$ denotes the set of all the nodes whose ads have overlaps with node $p$'s interests.

As shown in Fig. 17, when the ad delivery period is longer, the average cache storage for a node decreases since the source nodes issue ads less often if their contents do not change. Consequently, the success rate is decreased. It is straightforward that a shorter delivery period leads to more ad delivery messages, and thus higher system load. An interesting observation from this figure is that the system load does not change linearly with ad delivery periods. For example, when the delivery period is longer than 1 h, the system load mean and standard deviation decrease slowly although the ad delivery period doubles or triples. The reason is that, in addition to the periodical ad delivery, a node join or document change may also lead to an immediate ad delivery. As long as there is some node churn and/or document changes (such as document addition or deletion) undergoing in the system, many nodes in ASAP are able to deliver and receive ads dynamically. The periodical ad delivery works as a supplementary scheme for all nodes to deliver and receive ads in a timely manner.

5.6. Online ads requisition

In an active P2P system with a lot of node churn and content changes, some ads may not reach all interested peers and some peers do not hold all interesting ads, as can be inferred from Figs. 16 and 17. In most cases the average ad's node coverage and node's cache storage are lower than the search success rate because of the limited ad delivery budget and period. The reason why we still obtain a good success rate is that after a miss in local cache, a peer will look for ads from its neighbors (online ads requisition), and thereby enhance its cache content. In the next experiment we vary the hop limit for ads request messages (ads request TTL in brief) upon a cache lookup failure, and collect the results shown in Fig. 18. It is expected that, if the TTL is set to 0, the search success rate becomes unacceptably low. When this TTL is greater than 1, the improvement on success rate becomes very small (less than 1%), while the system load increases quickly with heavier load variations. As the ads request TTL increases, ASAP becomes more like traditional query-based search schemes. It shows that a TTL equal to 1 strikes a good balance.

6. Implementation issues and limitations

In this section, we discuss several implementation issues in ASAP and its limitations imposed by the assumptions.

6.1. Incentives for content sharing and ad delivery

To present a P2P system architecture, we do not consider the incentives for nodes to proactively send out ads about their contents. For a commercial content delivery system with appropriate payment mechanism, however, the incentive for content sharing and ad delivery is clear and strong. Just like the ads we receive every day, the owners may have to pay for the ad delivery (within a budget), but they will earn more by content sharing with the consumers. On the other hand, consumers receive ads freely, but have to pay for data that they are interested in. With an appropriate payment scheme, nodes would like to issue ads, and our ASAP provides an ideal infrastructure for such type of commercial P2P applications.

6.2. Dependence on peer interest clustering and stability

The performance of ASAP is dependent on the extent to which the peers are clustered and stick to their interests. In the worst case that each peer's interest often changes, ASAP degrades to a Gnutella-like query broadcasting system (by means of ads request). As evidenced by existing studies, however, explicit interest clustering is common in today’s P2P systems, and many works successfully exploit this interest to develop efficient search algorithms. ASAP moves one step further to proactively push content based on users' interest. The more clustered and stable the peers' exhibited interests, the better ASAP works: ads are published more targetedly in the network while peers better identify useful ads.

6.3. Node heterogeneity exploitation

Previous studies have shown significant node heterogeneity present in P2P systems, and hierarchical structures are employed...
in several P2P systems such as Gnutella 2, and Kazaa. This hierarchy helps improve search performance to some extent, and ASAP can be applied to obtain further improvements. Since super peers are more powerful than other peers in a domain, they can handle much more ad deliveries. Clients send their ads to their super peers, where all the ads are merged into a big one for this super peer domain. When searching, clients will first send their requests to their super peers, where a large collection of ads are collected.

6.4. Others

The ASAP generates background traffic in the P2P network regardless of the querying frequency. Therefore, the ASAP could work worse than the aforementioned algorithms.

7. 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 in just one hop with a content confirmation. Experimental results show that ASAP can boost search efficiency by improving search performance and slashing search cost. In addition, ASAP smooths out the system load and keeps it at a low level, and therefore, offering better system stability than existing search algorithms. Comprehensive simulation experimental results indicate that, compared with representative query-based approaches ASAP improves the search performance by more than 62% in terms of query response time and slashes the search cost by 2 to 3 orders of magnitude in terms of bandwidth consumed in a search in representative workloads. In addition, ASAP reduces the system load by a factor of two to four, and experiences small load variations.

Acknowledgment

We would like to thank the NSF for sponsoring this work in part under grants: CNS-0646910, CNS-0646911, CCF-0621526, CCF-0811413 and the Department of Energy for also sponsoring us in part under Early Career Principal Investigator Award: DE-FG02-07ER25747.

References


[14] Adam Shaked Gish, Yuval Shavitt, Tomer Tankel, Geographical statistics and geographical queries in Gnutella0.6. Adam Shaked Gish, Yuval Shavitt, Tomer Tankel, Geographical statistics and geographical queries in Gnutella0.6.


[20] Adam Shaked Gish, Yuval Shavitt, Tomer Tankel, Geographical statistics and geographical queries in Gnutella0.6.


Jun Wang received the B.Eng. degree in computer engineering from Wuhan University (formerly Wuhan Technical University of Surveying and Mapping), the M.Eng. degree in computer engineering from the Huazhong University of Science and Technology, China, and the Ph.D. degree in computer science and engineering from the University of Cincinnati in 2002. He is a member of the faculty of the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando. His research interests include I/O architecture, file and storage systems, parallel and distributed computing, cluster and P2P computing, and performance evaluation. He has received several major US National Science Foundation research awards from Computer Systems Research and Advanced Computation Research Programs and the US Department of Energy Early Career Principal Investigator Award program. He is a member of the IEEE, the ACM, Usenix, and SNIA.

Peng Gu received the B.S. and M.S. degrees in computer science from the Huazhong University of Science and Technology, Wuhan, China, and the Ph.D. degree in computer engineering from the University of Central Florida, Orlando. He is a software engineer in the Core Operating System Division, Microsoft Corp., Redmond, Washington. His research interests include file and storage systems, parallel I/O architecture, and high performance computing.

Hailong Cai is a software engineer at Google Inc. He received his Ph.D. degree in the Computer Science and Engineering department of the University of Nebraska, Lincoln in 2006. He received his Bachelors degree from the department of Computer Science and Technology of Huazhong University of Science and Technology in 1997, and got his Masters degree from the Institute of Software, Chinese Academy of Science in 2000. His research interests include distributed storage systems and computer network systems.