Foreseer: A Novel, Locality-Aware Peer-to-peer System Architecture for Keyword Searches ⋆

Hailong Cai, Jun Wang

Computer Science and Engineering, University of Nebraska-Lincoln
{hcai, wang}@cse.unl.edu

Abstract. Peer-to-peer (P2P) systems are becoming increasingly popular and complex, serving millions of users today. However, the design of current unstructured P2P systems does not take full advantage of rich locality properties present in P2P system workloads, thus possibly resulting in inefficient searches or poor system scalability. In this paper, we propose a novel locality-aware P2P system architecture called Foreseer, which explicitly exploits *geographical* locality and *temporal* locality by constructing a *neighbor* overlay and a *friend* overlay respectively. Each peer in Foreseer maintains a small number of neighbors and friends along with their content filters used as distributed indices. By combining the advantages of distributed indices and utilization of two-dimensional localities, the Foreseer search scheme satisfies more than 99% of keyword search queries and realizes very high search performance, with a low maintenance cost. In addition, query messages rarely touch free-riders, and therefore avoid most meaningless messages inherent in unstructured P2P systems. Our simulation results show that, compared with current unstructured P2P systems, Foreseer boosts search efficiency while adding only modest maintenance costs.
Keywords: geographical locality, temporal locality, Foreseer, Bloom filter

1 Introduction

Unstructured peer-to-peer (P2P) system becomes one of the most popular Internet applications at present, mainly resulting from its good support for file lookup and sharing. Its system architecture is categorized as either centralized or distributed. Centralized system architectures like Napster [1] require a central index server, which limits the system scalability and incurs a single point of failure [2]. Without centralized administration, recent systems like Gnutella [3] construct a totally decentralized overlay (at the application level) on top of the Internet infrastructure. Search schemes in these systems can be *blind* or *informed*. Blind search schemes are based on message flooding and thus suffer poor system scalability. To address this problem, some researchers have proposed random walks [4, 5] as well as several improved versions of this scheme, such as Directed BFS [6], GIA [7] and Interest-based shortcuts (IBS) [8]. However, these schemes are still blind to some extent because they lack any indexing information. As a result, they cannot prevent a peer from repeatedly trying multiple walks due to previous walk failures or meaningless walks toward free-riding peers. This is because the sender does not know what contents, if at all (a free-rider shares nothing), are shared on the receiver before the query is actually transmitted. A straightforward solution to

⋆ This research is supported in part by a University of Nebraska Lincoln Layman Fund.

resolve the blindness without using a centralized index server is to maintain distributed indices among peers, which are used in the informed search technique that trades off distributed indices storage management for search performance. Intelligent BFS [9], APS [10], Local Indices (LI) [6] and Routing Indices [11] are examples of this class. In order to intelligently direct searches and obtain an acceptable hit rate, however, the indices to be maintained would be extraordinarily large, and hence the overhead involved in indices' update may become prohibitively expensive, thus partially offsetting the benefits of the indices themselves.

Previous studies [12, 13] have shown that current P2P systems have both *geographical* locality and *temporal* locality. For node A, its geographically nearby node B exhibits geographical locality if it is likely to offer service to node A in the near future. This is because the objects on node A's neighbors are more likely to be reached by queries from A than those objects located on distant nodes from A. Similarly, node C, which has successfully served requests from node A in the past, exhibits temporal locality if it is likely to be able to offer further service to node A in the near future. The main reason why current search schemes cannot simultaneously realize high search efficiency and good scalability is that they do not or not fully exploit such an inherent two-dimensional locality in P2P systems. For example, an interest-based shortcut approach [8] attempts to exploit the temporal locality, but only to improve the blind flooding. LI [6] introduces a simple way to implement indices in unstructured P2P systems but does not consider any kind of localities. These search schemes introduce inefficient blind traffic in P2P network or require very expensive maintenance to manage distributed indices, especially in support of keyword searches, and therefore seriously compromise the search performance and system scalability. A good design of a P2P system that supports keyword searches must organize distributed indices more efficiently and exploit two-dimensional locality awareness intentionally.

In this paper, we attempt to design a novel unstructured P2P system architecture for keyword searches called *Foreseer*. Our design fully exploits two-dimensional localities. *First*, unlike Gnutella-like systems that simply organize live peers into an overlay with small-world property [1], Foreseer constructs two orthogonal overlays on top of the Internet infrastructure: a neighbor overlay based on geographical locality, and a friend overlay based on temporal locality which also has small-world properties. The neighbor overlay is built with network proximity while the friend overlay is maintained according to the online query activities. Each peer maintains a small number of links to its neighbors and friends that serve its future queries more efficiently. *Second*, we use Bloom filters [15] to compactly represent the contents shared on each peer and distribute the content filters, so that each peer saves copies of the content filters of its neighbors and friends as well as their IP addresses. *Third*, due to the native locality properties, one peer's neighbors and friends provide a much better chance of serving its query requests than other "strange" peers do. Therefore, Foreseer employs a locality-aware search scheme to answer queries more efficiently. The search is performed in two phases on each involved node on the route path: 1) *local matching* to resolve the query (*i.e.,* find the node that seems to have the requested objects) on behalf of its neighbors

[1] A network exhibiting the small-world property is one with high level of clustering and a small characteristic path length. For more details, please refer to [14].

and friends using their content filters, and 2) *selective dispatching* to forward the query to the destination peer if it has been resolved, or otherwise to the node's friends or neighbors. Foreseer is not only able to answer queries within few hops, but also reduces a lot of redundant flooding messages and skips most free-riders. In addition, using a Bloom filter index minimizes the maintenance overhead, making it even more suitable for a highly dynamic environment. Trace-driven simulation results show that Foreseer can boost the search performance by up to 70% and significantly reduce the search cost by up to 90%, compared with state-of-the-art P2P systems.

2 Related Work

We review several representative search schemes in Gnutella-like decentralized P2P system architectures in this section.

2.1 Blind searches

Gnutella does not scale well because of its message flooding scheme. One way to improve its scalability is to reduce the number of redundant messages by forwarding queries only to a subset of neighbors. The neighbors are either randomly picked or selectively chosen based on their capability of answering a query. Lv *et al.* [5] suggest a *random walk* scheme, in which a query is forwarded to a randomly chosen neighbor at each step until there are sufficient responses. Adamic *et al.* [4] recommend that the search algorithm bias its walks toward high-degree nodes. *GIA*, designed by Chawathe *et al.* [7], exploits the heterogeneity of the network and employs a search protocol that biases walks toward high-capacity nodes. Although these approaches are effective in reducing the number of flooding messages, the system performance is compromised. The search may require multiple walks due to previous walk failures or meaningless walks toward free-riders. By estimating neighbors' potential capabilities according to their past performance, a *Directed BFS* approach [6] selects neighbors that have either produced or forwarded many quality results in the past. This approach intuitively reuses the paths that were effective in previous searches. However, if any peer on a path departs, the path is lost. Sripanidkulchai *et al.* [8] exploit temporal locality to employ interest-based *shortcuts*. These shortcuts are generated and updated after each successful query, and are used to serve future requests. The authors claim that the destination peer who hosts requested objects can be found in just one hop for many queries. Unfortunately, this approach may delay other queries that cannot be satisfied by the shortcuts. This is due to the fact that the peer has to contact all the peers marked as shortcuts before it sends the query to its neighbors.

2.2 Informed searches

Considering the benefits of indices for object location, another way to improve Gnutella-like systems' performance is to build distributed indices. *Intelligent BFS* [9] maintains query-neighbor tuples on each peer. These tuples map classes of queries to neighbors who have answered most of the queries that are related. This technique tries to reuse paths that were used for previous queries of the same class. Unfortunately, this technique cannot be easily adapted to object deletion and node departures. In addition, its search accuracy highly depends on the assumption that nodes specialize in certain documents. In *APS* [10], each node keeps a local index of the relative probability for each object it requests per neighbor. This approach saves bandwidth but may suffer long delays if the walks fail. *Local Indices*, proposed by Yang *et al.* [6], suggests each node's index files are stored at all nodes within a certain radius r, and queries are answered on behalf of all of them. If r is small, however, the indices cannot satisfy many queries; whereas if r is big, the indices' update will be very expensive. In *Routing Indices* [11], each node stores an approximate number of documents from every category that can be retrieved through each outgoing link. This technique can be efficient for searches, but requires too much flooding effort for the indices to be created and updated. Therefore it may not work well in a highly dynamic environment.

3 Foreseer Design and Implementation

Foreseer comprises three components at different layers, as shown in Figure 1. The neighbor and friend overlays are built on top of the Internet infrastructure by exploiting geographical and temporal localities, respectively. The indices implemented by Bloom filters are distributed according to the relationships between peers within the two overlays. By directing searches along the overlay links and resolving queries by the distributed indices, the searching module provides high performance for keyword searches with a low maintenance cost.

The rationale behind Foreseer comes from our daily life. Everyone has neighbors who live nearby and friends who live further away. Neighbors and friends constitute one's social relationships. One gets to know his neighbors upon settlement, and makes new friends when interacting or doing business with someone else. Suppose each person has a business card and knows about others only through their business cards. When someone has a new business request, he looks at his the business cards of his friends and neighbors first. If these cards imply that a friend and/or neighbor can help, he immediately contacts that person. If none of them can help, he passes the request to his friends and/or neighbors who, in turn, seek help from their friends and neighbors.

We present the detailed design of Foreseer in this section. First, we explain the creation of peers' content filters represented by Bloom filters. Then we show how to explicitly exploit both geographical and temporal locality and construct the neighbor and friend overlays. In Section 3.3, we present the Foreseer search algorithms. The system maintenance cost is detailed in Section 3.4.

3.1 Bloom filters as content filter

In Foreseer, the business card refers to a peer's content filter derived by computing the Bloom filter [15] on all its shared contents. A Bloom filter is a hash-based data structure for representing a set to support membership queries. The membership test returns false positives with a predictable probability, but it never returns false negatives. With an optimal choice of hash functions, we can obtain the minimum probability of false positive as $(\frac{1}{2})^k$, or $(0.6185)^{\frac{m}{n}}$, where k is the number of hash functions used, m is the number of bits in the filter and n is the number of elements in the set.

In this paper, we use Counting Bloom filters proposed by Fan *at al.* [16] to summarize the contents shared with each peer. Assume D_p is the set of documents shared on node p, and $K_p = \{kw_i \in d_j | d_j \in D_p\}$ is the set of keywords that appear in any

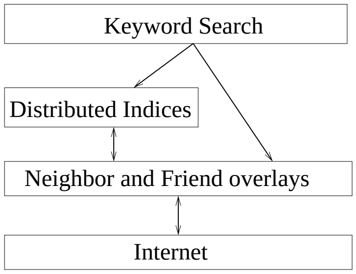

Fig. 1. System architecture of Foreseer built on top of the Internet infrastructure.

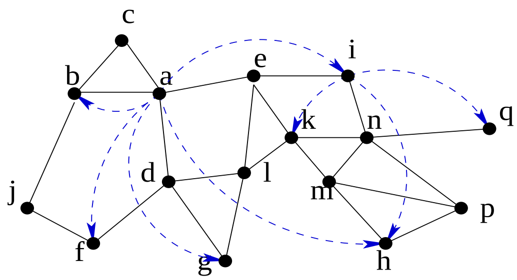

Fig. 2. Illustration of the neighbor overlay and friend overlay. For clarity, this figure only shows the friend links from node *a* and *i*.

documents in D_p. The kw_i is a keyword that appears in document d_j. The content filter of node p, denoted by $\mathcal{F}_{content\_p}$, is initialized by hashing all the keywords in K_p and setting the corresponding bits to 1. Free-riders have a null content filter, which can be easily recognized by other peers. If the number of hash functions in use is fixed, the cardinality of the maximum keywords set K_{max} determines the space requirement for the filter with the least false positive rate as $m = \frac{nk}{\ln 2} = \frac{|K_{max}|k}{\ln 2}$. We believe that the size of the maximum keyword set will not be arbitrarily large for several reasons. *First*, the number of shared files on most peers is limited. Saroiu *et al.* [17]'s measurement studies on Gnutella indicate that about 75% of the clients share no more than 100 files, and only 7% of the peers share more than 1000 files. The results in [12] show that 68% of 37,000 peers share no files at all (free-riders), and most of the remaining clients share relatively few (between 10 and 100) files. *Second*, the documents on the same peer tend to share common topics. The overlap of semantics among documents on one peer reduces the number of unique keywords to be mapped to the content filter. *Third*, according to [12, 13], most files shared in current P2P systems are multimedia streams, where only a few unique keywords can be derived from one document. Even with $|K_{max}| = 10,000$ and $k = 8$, the length of the filter with least false positive rate is $m = \frac{10,000 \times 8}{\ln 2} = 114,416$ bits $= 14.4KB$. When transmitted over the network, this filter can be packed into several IP packets. However, as the network size increases, some peers may share so many files that the cardinality of their keyword sets become greater than current K_{max}. This problem can be solved in two ways. If only a few peers begin to share a large number of files, a workload migration mechanism can be used to move some of these shared files to their neighbors and/or friends. If a lot of peers want to share a large number of files, we can deliberately increase the length of the Bloom filters according to the size of the maximum keyword set K_{max}. 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 (location of a 1 in the filter) is transmitted over the network.

To facilitate keyword queries, each peer keeps an inverted file created from its keyword set by information retrieval techniques. In addition, a counter is locally maintained for each bit in its content filter to record the current number of objects mapped to this bit by any of the k hash functions. Notice that this counter does not need to be transmitted over the network since it is only used in local file insertion and deletion.

3.2 Two-dimensional, locality-aware overlay construction

Two orthogonal overlays are constructed in this system: 1) the neighbor overlay, which captures geographical locality, and 2) the friend overlay, which captures temporal locality. In addition to maintaining its own local content filter, each node p saves copies of the content filters of the peers in both its neighbors list $N(p)$ and friends list $F(p)$. If a peer becomes a neighbor and a friend at the same time, it is allowed to act as both a neighbor and a friend. To limit the number of filters one peer maintains, we restrict the size of N and F as follows: for node p, $n_{min} \leq |N(p)| \leq n_{max}$, $f_{min} \leq |F(p)| \leq f_{max}$, where n_{min}, f_{min} and n_{max}, f_{max} are the lower bound and upper bound for the number of neighbors and friends respectively. Figure 2 illustrates the two overlays in a simple network where $N(a) = \{b, c, d, e\}$ and $F(a) = \{b, f, g, h, i\}$. Node a maintains a copy of the content filter for each node among $N(a) \cup F(a)$. Together with its own local filter, node a maintains nine content filters in total, which greatly strengthens its ability to serve future queries.

Finding and maintaining neighbors. The bidirectional neighbor overlay is constructed with network proximity, so that only peers that are physically proximate can become each other's neighbors. Network latency is used as a simple metric to measure the physical distance between peers because it reflects the performance directly seen by end hosts and can be easily measured in an end-to-end, non-intrusive manner. As in Gnutella, a new node joins the system by contacting well-known bootstrapping nodes, and builds up its neighbors according to the replies. To ensure network proximity, a new peer, upon receiving replies from peers who can accept more neighbors, will always choose peers with lower latencies as neighbors. The content filters are transmitted along with the replies so that the new peer can initialize its neighbors list quickly. If the number of its neighbors is smaller than the lower bound, the node issues a PING_NEIGHBORS message to its current neighbors. The current neighbors, in turn, propagate this message to their neighbors. Upon receiving this message, peers with less than the maximum number of neighbors reply positively along with their content filters. This process repeats until the new arrival peer has a minimum number of neighbors. When a peer makes a planned departure, it notifies its neighbors so that they can remove it from their neighbors list and discard its content filter. In case of a node failure, the node does not have a chance to notify other peers. However, its neighbors will realize this when trying to contact it later, and make updates accordingly.

Geographical locality implies that an object near the querying peer is more likely to be reached than distant objects, thus minimizing network latency and bandwidth consumption. The construction of the neighbor overlay ensures that each peer keeps a list of its nearby peers, and resolves the query locally if the requested object can be found on any of its nearby peers.

Making and refreshing friends. The friend overlay is constructed as a directed graph independent of the physical network topology. Unlike bidirectional friendships in real life, the friend relationship in this paper is designed to be unidirectional. The unidirection of the friend relationship does not affect the small-world property of the subgraph induced by the friend links, as we will prove in our experiments. Each peer knows a number of friends. Each peer may also be a friend of other peers, who are called *back* *friends*. In addition to the neighbors and friends lists, each peer also maintains a list (including IP address) of its back friends denoted as F^{-1}, but without any content information. Tracking a reverse direction of the friends relationship, the list is used to notify those back friends peers of its content filter update when necessary. In Figure 2, $\{a, i\} \subseteq F^{-1}(h)$, and any filter update on node h would cause node a and i to take corresponding action: updating filter copies of node h accordingly.

It is obvious that a peer who has ever answered a request from node p should be a candidate of p's friends, according to temporal locality principles. However, when a brand new peer issues its first query, it has no friends to consult. To mitigate this problem, we recommend an active "friends making" stage for the new node as soon as it builds up its neighbors list. To find potential friends, new node p sends out a PING_FRIENDS message to its neighbors, who in turn forward this message to their friends. Upon receiving this request, peer q checks to see if it can be accepted ($|F^{-1}(q)| < f^{-1}_{max}$, where f^{-1}_{max} is the maximum number of a peer's back friends) or not. Those peers who can accept this "friends making" request will reply to p along with their content filters. Based on these replies, p can fill out its initial friends list by selecting those peers who have more 1's in their content filters, because the documents shared on these peers contain more keywords. Since free-riders have nothing to share, no friend link will point to them (*i.e.,* they have an empty F^{-1} list). Therefore, they will never see PING_FRIENDS messages except as a neighbor at the first step. Updating the friend overlay due to node departure/failure can be done in a similar manner as the neighbor overlay.

Node p's friends are ordered and replaced in an LRU manner as new information is learned. After each download, p has a chance to refine its friends list. If the serving peer is already one of p's friends, this peer comes to the top of the list because it is the most recently used. If the serving peer is not on p's friends list, and $|F(p)| < f_{max}$, this peer becomes a new friend of p with the highest priority. However, if $|F(p)| = f_{max}$, p has to remove a least recently used friend and insert the new friend as the most recently used. When an old friend is replaced, p sends a message to that node so that it is removed from that node's back friends list. For partial searches that result in multiple responses, we can use a counter to record the number of results each friend returns for previous X queries, and find a victim that returns the fewest number of results when a newly recognized friend asks for a replacement. Accordingly, the counters are updated when a query is answered.

To attain a better performance, we can speed up the node join procedure by employing a caching scheme. Before a node departs the system, the addresses of its neighbors and friends are saved on its local disk. When the node rejoins the system, it tries to contact its old neighbors and friends directly and asks for their current content filters. Recent research results [13, 18] show that the node departure-and-rejoin pattern is a common feature of current P2P systems. Not only is the join process simplified in this way, but the workload of bootstrapping nodes is also reduced.

3.3 Two-dimensional, locality-aware search algorithm in Foreseer

Algorithm design. The keywords extracted from documents work as metadata to be mapped to the content filters in our design, although the extraction method is beyond the scope of this paper. Existing systems use either *local* or *global* [19] indexing to retrieve or place a document's metadata. As described above, Foreseer uses local indexing so that multi-term queries are as easy to be processed as single-term queries. Foreseer avoids the inefficiency of local indexing by orienting the queries intelligently instead of flooding every node in the system.

The main process of object location is simple. When initiating a new query request or receiving an unresolved query message that contains one or more terms $kw_1, kw_2, ...kw_r$, node p runs a search algorithm that consists of two phases: *local matching* and *selective dispatching*. In the local matching phase, node p computes the query filter $\mathcal{F}_{query}$ by mapping all the query terms, and compares it with the content filter $\mathcal{F}_{content\_q}$ for each node $q \in N(p) \cup F(p)$ by the logical "AND" operation. If $\mathcal{F}_{query} \wedge \mathcal{F}_{content\_q} = \mathcal{F}_{query}$, then there is a match, indicating that node q seems to have the document containing all the keywords with a high probability. Otherwise, none of p's neighbors or friends has the requested document. This matching is conducted on node p locally and requires no network bandwidth. The query message is then selectively forwarded based on the results of the first phase. If there is a match, *i.e.*, the query has been resolved and is likely to be answered by one of p's neighbors or friends q, the message is flagged as *resolved* and forwarded to q, which looks up its local inverted file for the document that matches the query. If a false positive occurs, however, the query is flagged as *unresolved* and returned to node p. In either situation, whether the local matching fails or a match turns out to be a false positive, the query message is forwarded according to the system searching policy $P_1 = \{F^{h_1} N^{h_2}\}$. Let h be the current hop count of the query message. In P_1 policy, given $h < h_1$, the query is forwarded to its friends. If $h_1 \leq h < h_1 + h_2$, the query will be forwarded to its neighbors. The query stops traveling when $h = h_1 + h_2$. Because of the locality properties in P2P system workloads, a large number of queries are resolved locally (when $h = 0$) at the nodes who issue the requests. For other queries that need to travel more hops, Foreseer runs an intelligent, light flooding procedure: the query messages are forwarded along the friend links for up to h_1 hops and then along the neighbor links for up to h_2 hops until they are successfully resolved and answered by some nodes in the system, or otherwise fail with an exception.

We search along the friend links before the neighbor links based on several intuitive reasons. *First*, suppose $q \in F(p)$ and $r \in F(q)$, files shared on node r, which tend to interest node q, may also interest node p because p is likely to download more files from q in the near future according to temporal locality principles. On the other hand, geographical locality only ensures that objects *near* node p are more likely to be reached than *distant* objects, but does not mean that node p's neighbors have a better chance of answering the query. This implies that peers reached through friend links have a better chance of answering the query than peers reached through neighbor links. *Second*, the construction of the friend overlay implies that the friend links point to peers who share many objects and never refer to free-riders. These peers have a better chance of answering the query than other peers sharing few or no files. *Third*, the small-world property of the friend overlay ensures that by following friend links, the query quickly scatters over a large network diameter and reaches distant peers in few hops. Because of its construction, however, the friend overlay may consist of disconnected subgraphs. To ensure a high success rate for searches, Foreseer propagates the query along neighbor links after h_1 hops in the friend overlay. At this stage, free-riders may serve as an intermediate router for the query messages.

We also develop and examine other possible search policies that can be employed in our two-dimensional overlays. By directing the query along neighbor links first and then friend links, we have a policy $P_2 = \{N^{h_1}F^{h_2}\}$. However, this policy suffers a low success rate since the neighbor overlay is built with network proximity. Therefore, going through neighbor links first cannot reach distant peers who may have the requested document. We could also forward the query messages to neighbors and friends simultaneously, as denoted by policy $P_3 = \{F^{h_1}//N^{h_2}\}$. However, propagating through neighbor links at the beginning does not incur much benefit since only peers within a local area will be touched. For the same reason, a more complicated policy like $P_4 = \{N^{h_1}F^{h_2}N^{h_3}\}$ does not attain a better result. Other policies like $P_5 = \{F^{h_1}N^{h_2}F^{h_3}\}$ do not work better than P_1 since propagating the query along the friend links again after it has traveled along the neighbor links does not capture the temporal locality. Our experiments also prove that P_1 attains the best outcome among all the policies mentioned above. Therefore, we adopt P_1 as our default search policy in all experiments without explicit specification.

In short, the key ideas (major procedures) in our search scheme are three-fold: 1) The peer who issues the query tries to resolve the query instantly by local matching. If successful, only one more message will be involved in this search. 2) If 1) fails, the query will be selectively forwarded along the friend links and then the neighbor links, and will never touch free-riders until traveling along the neighbor links. 3) As soon as the local matching is successful at some nodes, the query is resolved and only one more message is needed for success confirmation in case of no occurrence of false positives.

Algorithm efficiency. For any peer p issuing a query, if the requested object can be found on any peer $p_0 \in N(p) \cup F(p)$, this query can be resolved locally and reach the destination peer within only a single hop with a high probability. Such kind of queries are resolved with $O(1)$ complexity. In other cases, the query needs to be spread out as is examined here. We use $F^i(p)$ to denote the set of peers with i hops distance along friend links from node p, and let $F^0(p) = \{p\}$. Formally,

$$F^i(p) = \{q | r_0 = p, r_i = q, r_{k+1} \in F(r_k), 0 \leq k \leq i-1\}$$

If the requested object resides on any peer

$$p_i \in N(f^i(p)) \cup F(f^i(p)), \forall f^i(p) \in F^i(p), 0 \leq i \leq h_1$$

it can be resolved in i hops and reach the destination peer in $i+1$ hops with a high probability. At each hop, the query touches some new nodes along the friend overlay, and checks the content filters of their neighbors and friends. Table 1 shows the number of peers touched, the number of peers foreseen, and the number of messages produced at each hop, in which f and n denote the average number of one peer's friends and neighbors respectively. We do not consider revisited peers for simplicity.

If the query fails in the friend overlay, it spreads by following the neighbor links. Similarly, we use $N^j(f^{h_1}(p))$ to denote the set of peers with j hops distance along the neighbor links from node $f^{h_1}(p)$, where $f^{h_1}(p) \in F^{h_1}(p)$. Formally,

$$N^j(f^{h_1}(p)) = \{q | r_0 = f^{h_1}(p) \in F^{h_1}(p), r_j = q, r_{k+1} \in N(r_k), 0 \leq k \leq j-1\}$$

Table 1. The search efficiency in the friend overlay.

Hop	Nodes touched	Nodes foreseen	Messages produced
0	1	$f+n$	0
1	f	$(f+n)f$	f
...	...	...	...
i	f^i	$(f+n)f^i$	f^i
...	...	...	...
h_1	f^{h_1}	$(f+n)f^{h_1}$	f^{h_1}

Table 2. The search efficiency in the neighbor overlay.

Hop	Nodes touched	Nodes foreseen	Messages produced
0	1	$(f+n)f^{h_1}$	0
1	n	$(f+n)nf^{h_1}$	n
...	...	...	...
j	n^j	$(f+n)n^j f^{h_1}$	n^j
...	...	...	...
h_2	n^{h_2}	$(f+n)n^{h_2} f^{h_1}$	n^{h_2}

If the requested object resides on any peer

$$p\prime_j \in N(n^j(f^{h_1}(p))) \cup F(n^j(f^{h_1}(p))), \forall n^j(f^{h_1}(p)) \in N^j(f^{h_1}(p)), 1 \leq j \leq h_2$$

it can be resolved in h_1+j hops and reach the destination peer in h_1+j+1 hops with a high probability. Table 2 shows the the number of peers touched, the number of peers foreseen, and the number of messages produced at each hop along the neighbor links.

As long as the shortest distance between the query source peer and the pre-destination peer that has a successful local matching is not longer than h_1 hops in the friend overlay plus h_2 hops in the neighbor overlay, this query is satisfied by our algorithm. Similar to other flooding approaches, a randomly generated identifier is assigned to each query message and saved on passing peers for a short while so that the same message is not handled by the same peer again. Our experiments suggest that 5 hops in the friend overlay ($h_1 = 5$) and 1 hop ($h_2 = 1$) in the neighbor overlay suffice for more than 99.9% of queries. As shown in Figure 2, where $N(a) = \{b, c, d, e\}$ and $F(a) = \{b, f, g, h, i\}$, a query from node a for objects shared on these nodes can be resolved locally and reach the destination peer in one hop. Since $j \in N(b), l \in N(g), m \in N(h), p \in N(h), n \in N(i)$, and $k \in F(i), q \in F(i)$, a query for objects shared on any of these peers can be resolved in one hop by one of a's friends, and reach the destination in two hops. Compared to other blind search algorithms, Foreseer conducts searches more aggressively with the help of distributed content filters by checking $(f + n)$ times more nodes at each hop. Compared to current distributed indexed approaches, Foreseer directs searches more intelligently by well exploiting temporal and geographical locality properties, namely following the friend links and then neighbor links accordingly.

Skipping free-riders. Previous studies have shown that a large portion of participating nodes are free-riders. Although any node can issue queries to the system, only non-free-riders can contribute objects and are helpful in answering queries. The friend overlay ensures that free-riders cannot be friends of any peer, and thus a query will never touch them when traveling along the friend links for h_1 hops. If the query is not yet resolved, Foreseer continues the search in the neighbor overlay and may touch some free-riders. By giving search priority to the friend overlay, Foreseer reduces a lot of meaningless messages that are unavoidable in many blind search schemes.

Effects of false positives. When a false positive occurs, the peer that resolves the query receives a negative reply from the "matching but false" peer, and the query is further forwarded. However, this only incurs some extra workload with a very low probability, and does not affect the correctness of the search algorithm. To reduce the side effects of false positives, when a peer finds that more than one neighbor or friend may have the requested document, it forwards the query to two or more of them. The probability of obtaining multiple false positives at the same time is very slim. For partial keyword searches that require multiple responses, the query can be forwarded to all of its neighbors and friends that seem to have the requested documents.

3.4 System maintenance

Object publishing and removal. When a peer is going to share new files, this information should be quickly made visible to its neighbors and back friends peers. To do this, the peer extracts keywords from new documents, and selects new keywords from all extracted keywords, if any, to map to its content filter. Any change in the filter is recorded and sent in an update message to all peers in its lists N and F^{-1}. Only a small amount of location information (less than k bits per new keyword contained in the file) that reflects the changed bits is transmitted over the network, and thus the involved network traffic is minimized. At the same time, the inverted file and counters associated with corresponding bits are also updated locally on the peer. Upon receiving an update message, peers make necessary changes on their filter copies of the sender node. The process of removing a document is similar to this object publishing procedure. Compared to current indexed search schemes or DHTs, only a small amount of location information specifying the changed bits in the content filter is transmitted when publishing or removing a document, thus making Foreseer perform an efficient maintenance job in a highly dynamic environment.

An additional benefit from this instant publishing is that popularly requested files are advertised widely and quickly. As a result, they will have more and more copies available in the network as more peers conduct such downloading and publishing processes. Therefore, queries for this kind of files can be served more efficiently and the query hot spot may not occur at all.

Node join and departure. When a new node joins the system, it needs to set up its neighbor and friend relations as described in Section 3.2. When a node departs, it notifies all its neighbors and back friends. A total of $(|N| + |F^{-1}|)$ small messages are involved per update if necessary. The nodes that receive this notification message simply remove this node from their neighbors or friends lists, along with the corresponding content filter copies. In the meantime, the departing node caches its neighbors and friends on its local disk. When it rejoins the system, it first tries to contact its old neighbors and friends to build up its initial relations quickly. When a node fails unexpectedly, it has no chance to notify other nodes of its absence. But when live nodes try to contact it, they would find this node has already departed. Since each node maintains multiple neighbors and friends, random node failures do not affect the overall system performance. A more aggressive approach would be adoption of PING-PONG messages to proactively check the live status of each peer's neighbors and friends.

4 Experimental Methodology

We develop a trace-driven simulator to evaluate the performance of Foreseer compared with other baseline P2P systems. We describe the experimental methodology in this section and present the simulation results and our analysis in the next section.

4.1 Set up experiments

To evaluate Foreseer's performance compared with other popular search schemes, we carefully choose several representative systems, such as Gnutella, Interest-based Shortcuts (IBS) [8], and Local Indices (LI) [6] as baselines. For the LI scheme, where each node maintains an index over the data of neighbors within r hops, we choose two practical configurations, LI-1 with $r = 1$ and LI-2 with $r = 2$ without loss of generality. LI-1 maintains fewer indices than LI-2, but its indices are not able to satisfy enough queries. LI-2 maintains more indices but has a high maintenance cost. We configure each baseline system according to its default configuration to guarantee a fair comparison. For systems without indices such as Gnutella and IBS, we set $TTL = 7$. For LI-1, LI-2 and Foreseer, which maintain indices on each node, we set $TTL = 6$. As suggested by Yang *et al.* [6], a policy $P = \{0, 3, 6\}$ is adopted in both LI-1 and LI-2 to gain good performance. This policy suggests that the query is processed by nodes at depth 0, 3 and 6, while nodes at other depths simply forward the query to the next depth. In IBS, each node maintains at most 10 shortcuts as specified by the authors [8]. In Foreseer, unless explicitly specified, we have $2 \leq |N(p)| \leq 10$, $4 \leq |F(p)| \leq 8$, and $|F^{-1}(p)| \leq 20$ for any node p. The default search policy in Foreseer is $P_1 = \{F^5 N^1\}$.

In order to best simulate the system performance, we choose the Transit-Stub model [20] to emulate a physical network topology for all testing systems. 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, on the average, 16 transit nodes. 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 stub nodes in different stub domains. The network latency is set according to the following rules: 100 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. We randomly choose peers from these 51,984 nodes to construct the testing P2P systems in our experiments. Notice that only some of the physical nodes participate in the P2P system while all nodes contribute to the network latency for messages passing by.

For baseline systems that require a Gnutella-like network overlay, we apply the crawled Gnutella network topology data downloaded from the Limewire website [21], and then set up the logical network connections. Foreseer needs to build its own neighbor and friend overlays that are different from the baseline systems. To construct the neighbor overlays with network proximity in Foreseer, we find nearby nodes and create neighbor links for a new peer according to the network latency between this peer and other present peers that may accept more neighbors. This process repeats until each peer has a sufficient number of neighbors. The initialization and online evolution of the friend overlay are described in Section 3.2. In each run of the trace replaying, we randomly select 5% node departures and 5% node failures on the fly to emulate dynamic activities in P2P systems.

Trace preparation. Because there is no real-world publicly accessible trace that contains the keyword query and download history information required in our experiments, we carefully rebuild a trace that contains original query terms associated with each event by preprocessing a content distribution information trace of an eDonkey [22] system obtained from [12]. The eDonkey trace contains the names of 923,000 files shared among 37,000 peers, and was probed during the first week of November 2003. To restore the keyword query trace, we do two preliminary jobs (calculating keyword weights and restoring a download trace) and then transform the download trace into the keyword query trace with keyword weight information. To do the first job, we conduct a simple lexical analysis and extract keywords from each document by converting its file name into a stream of words. We then calculate the total number of occurrence per keyword, which indicates the weight of the keyword among the entire document set. For the second job, we process the same eDonkey trace to restore a download trace, where each event consists of a peer that issues a query, a peer that answers the query, and the document that is transferred. During the restore process of the download trace, we assume that only one copy of each document is shared in the system before any query and download, and the content distribution of the eDonkey trace reflects the system status after the completion of all queries and downloads. This assumption is only used to derive a reasonable downloading trace, and does not disallow a file to have multiple copies in the system during the trace replay. When the two jobs are finally completed, we transform the download trace into a keyword query trace used in our experiments. During the transform process, we add query terms that have relatively high weights out of the requested file in each event, since these terms are more effective to improve search precision than those with light weights. The number of query terms is controlled within a limit so that most of the queries involve 1 to 4 keywords. The maximum number of terms for each query is limited by 10, as suggested by Reynolds *et al.* [23]. When the keyword query trace is restored, we feed it into each testing system, replay the queries, and collect the results.

Metrics in use. We measure the search efficiency using the following metrics:

– *Success rate*: the percentage of successfully resolved (*i.e.,* can be satisfied) queries among all submitted queries.
– *Response time*: the average response time to find the first matching document. Since the processing time at a node is negligible compared to the network delay, we ignore the queuing latency and Bloom filter computation times.
– *Relative distance*: the distance actually traveled by consecutive nodes encountered along the route normalized to the shortest distance between the source and the destination node, as defined in [24]. This metric tells the search cost, in terms of distance traveled in the proximity space, and indicates how well the system exploits the geographical locality, while the average hops only count the number of P2P nodes along the route.
– *Messages produced*: the average number of messages produced while searching for an object that matches a query.

Table 3. Performance of different search policies of Foreseer.

Policy	Success rate	Average hops	Average response time	Relative distance	# of query messages	# of node touched	# of free-riders touched
$P_1 : \{F^3N^3\}$	93.64%	3.01	426	3.91	194	164	56
$P_1 : \{F^4N^2\}$	99.7%	2.95	453	4.22	234	175	18
$P_1 : \{F^5N^1\}$	99.9%	2.97	459	4.29	262	177	2
$P_1 : \{F^6N^0\}$	99.34%	2.95	457	4.24	250	169	0
$P_2 : \{N^4F^4\}$	28.3%	5.64	482	4.36	181	163	5
$P_3 : \{F^4//N^2\}$	94.87%	2.77	402	3.70	191	150	7
$P_3 : \{F^4//N^3\}$	98.25%	2.75	382	3.42	210	170	28
$P_3 : \{F^4//N^4\}$	99.29%	2.74	374	3.36	228	187	42
$P_4 : \{N^2F^2N^2\}$	96.46%	4.14	442	4.01	228	181	18
$P_5 : \{F^2N^2F^2\}$	98.1 %	3.35	403	3.65	208	174	47

– *Nodes touched*: the average number of nodes touched by the query messages during the search.
– *Free-riders touched*: the average number of free-riders touched by the query messages during the search.

The first three metrics demonstrate how well a system conducts searches for a given query (search performance), while the last three metrics indicate the bandwidth consumption involved in a query (search cost) that can be used to indirectly justify the system scalability. The success rate is the most important factor in choosing searching policies. Besides these metrics, we also compare the indices' maintenance overhead involved in both LI schemes and Foreseer because their work in updating indices for object addition and deletion impacts the entire system performance and its scalability.

5 Experimental Results

5.1 Search policy

We conduct experiments to find an optimal search policy of Foreseer by running 20,000 queries on a 10,000-peer network and comparing the performance of each policy as shown in Table 3. On the average, most policies in class P_1 and P_3 show better performance than other policies, because temporal locality is fully exploited by traveling along the friend links in the early stages. The table also indicates that P_1:$\{F^4N^2\}$, P_1:$\{F^5N^1\}$, P_1:$\{F^6N^0\}$ and P_3:$\{F^4//N^4\}$ achieve a success rate higher than 99%. Among these policies, P_3:$\{F^4//N^4\}$ achieves the best search performance in terms of average hops per query, average response time per query and relative distance per query, but still touches a lot of free-riders in a search. Due to its high success rate and good performance (although not best), we use P_1:$\{F^5N^1\}$ as our default policy when running our experiments.

In order to demonstrate the different functions of neighbor and friend links at each hop, we collected the number of queries resolved by friend links and by neighbor links respectively at hop $h = 0, 1, ..., 6$. Notice that $h = 0$ indicates a successful local matching at the peer who issues the query. If the query is resolved at node p, and the destination

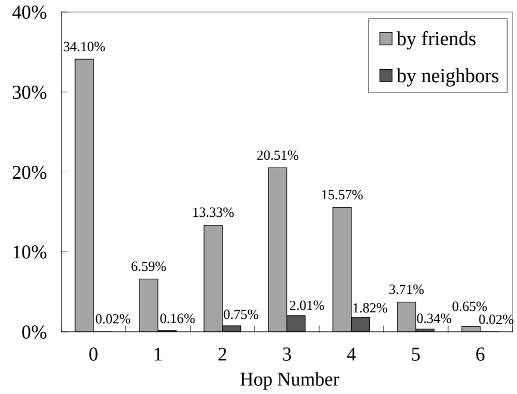

Fig. 3. Distribution of queries served by friend links and neighbor links at each hop, with policy $P1 : \{F^5N^1\}$.

peer is both a neighbor and a friend of p, this is considered as a contribution of both links. The results are shown in Figure 3, which illustrates that more than 34% queries are resolved locally (h=0) due to temporal locality in the workloads. When $h > 0$, the number of nodes touched at each hop dominates in answering the queries. As the hop number increases and the search touches more peers in the friend overlay, both the friend and neighbor links serve an increasing number of queries until the hop number reaches 3. When the hop number exceeds 3, however, the number of served queries by either type of links decreases since most of the queries are already satisfied. This figure also shows that for each hop number, the friend links serve many more queries than the neighbor links. One reason for this is that the friend links established by temporal locality are more likely to serve future requests, while the neighbor links constructed with geographical locality only help reduce the search cost. Another reason is that each peer maintains up to 8 friends while the average number of neighbors is only around 2.43, which implies that there are many more friend links existing in the system than neighbor links. Although it seems that the neighbor links do not contribute much to the search performance, they play a critical role in Foreseer. They increase the success rate by connecting isolated nodes without many friend relationships, and reduce the search cost in relative distance by network proximity, as shown in Table 3.

5.2 Search efficiency

In this section, we compare the search efficiency of Foreseer against the baseline systems in terms of search performance and search cost. By running the query traces, we found that, with the configurations mentioned above, all the baseline systems have an average success rate around 98%, while our Foreseer achieves an even higher success rate of 99.9%. Because of the uncontrolled data placement and finite TTL for query messages, no current search algorithms in unstructured P2P systems can guarantee a 100% success rate. However, only a small percentage (less than 0.06%) queries may fail in Foreseer, which is quite satisfactory for most users.

Search performance. The average response time and relative distance of each algorithm to find the first matching document are shown in Figure 4 and Figure 5, respec-

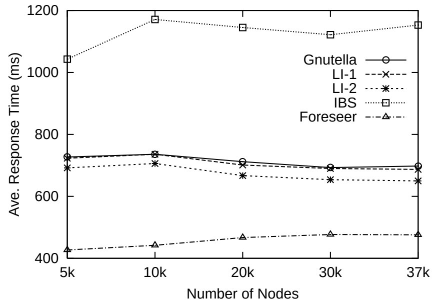

Fig. 4. Comparison of search performance in average response time.

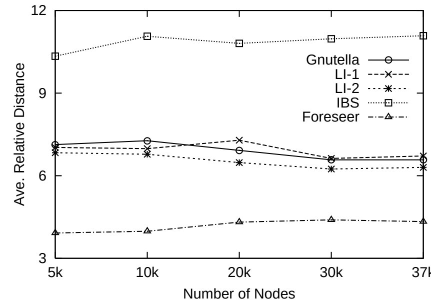

Fig. 5. Comparison of search performance in relative distance.

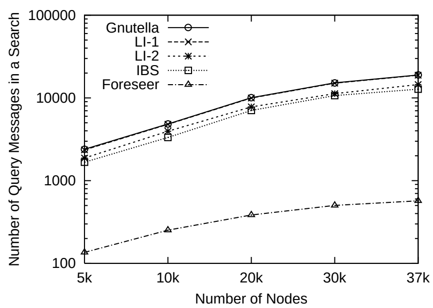

Fig. 6. Comparison of search cost in the number of messages produced in a search.

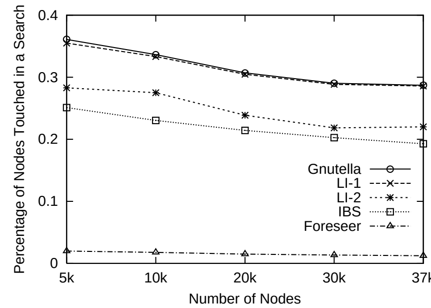

Fig. 7. Comparison of search cost in the percentage of nodes touched in a search.

tively. By exploiting temporal locality, both IBS and Foreseer can answer a lot of queries (around 34% in our experiments) within just one hop. Compared to IBS, however, Foreseer reduces the average response time and relative distance by up to 70%. It is clear that IBS obtains such one-hop successes with the cost of a longer response time and a farther relative distance. In IBS, a peer first contacts all of its shortcuts to see if they can answer the query. If no shortcut peer has the requested object, the search is already delayed before the peer floods the query to its neighbors. In contrast, Foreseer does not need the flooding algorithm as a backup. Compared to other baseline systems like Gnutella and Local Indices, Foreseer reduces the average response time by up to 40% and the average relative distance by up to 45%. The benefit stems from Foreseer's ability to exploit both temporal and geographical locality at the same time to propagate queries. Following the friend links enables Foreseer to reach the destination peer within few hops. Furthermore, if the local matching indicates a neighbor seems to have the object, the peer forwards the query to that neighbor, which is physically nearby according to network proximity. However, the neighbor in other systems only indicates a logical connection and may point to a distant node.

Search cost. Gnutella has poor system scalability because its blind flooding results in a large number of redundant messages and touches too many unrelated peers during the object searches. Other baseline systems also require a lot of messages if the query is not satisfied by the shortcuts (in IBS) or the local indices (in LI schemes). We collected

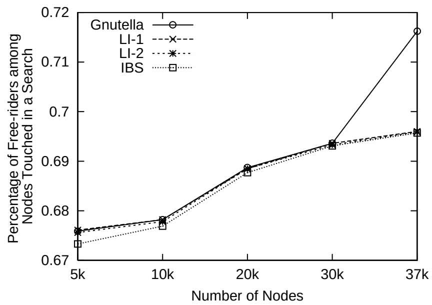

Fig. 8. Comparison of search cost in the percentage of free-riders among the touched nodes in a search.

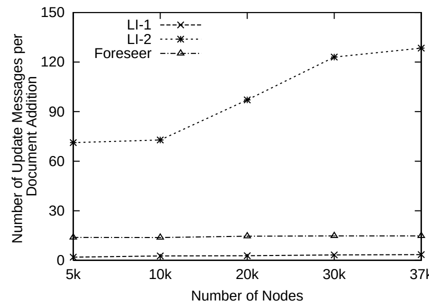

Fig. 9. Comparison of indices maintenance cost in the number of update messages produced for this purpose.

the total number of query messages and touched nodes in all the queries, and computed the average number of messages produced and the average percentage of touched nodes in a search, as shown in Figure 6 and Figure 7 respectively. Compared with IBS, which shows the best results among the baseline systems, Foreseer can reduce both the number of messages and the percentage of touched nodes by more than *90%*. In our experiments, Foreseer only touches less than 2% live nodes for each query on average. From the two figures, we can see that both IBS and LI-2 show capabilities of reducing the number of redundant messages and touched nodes. IBS achieves this improvement by exploiting temporal locality, while LI-2 by maintaining abundant index information. By combining their advantages, Foreseer improves the search performance and simultaneously slashes the search cost.

One of the valuable features of our friend overlay is that no free-riders are pointed by any friend links since they share nothing and cannot serve any query. Therefore, the search will never touch free-riders while propagating along friend links. If the query is not satisfied in the friend overlay, then it will touch free-riders along the neighbor links. In other systems, however, even a peer knows that some of its neighbors are free-riders (by looking at the indices as in LI schemes), it still sends the query to them when fanning out the query. We conducted experiments and calculated the percentage of free-riders among nodes touched in a search. Figure 8 depicts the results we obtained when running queries on the baseline systems. The result of Foreseer, not shown in the figure, is less than 1% in the experiments.

5.3 Maintenance costs

When a query is answered, the peer who issued that query has a new document to be published to other peers (we assume this is a requirement). We compared the number of messages used to update indices in LI schemes and Foreseer, as shown in Figure 9. It is clear that LI-1 only needs to send several update messages after a query on the average, because only a small number of peers need to be reached for indices update. But for LI-2, since each peer stores the indices of files shared on all the nodes within radius $r = 2$, an index update will result in a large number of update messages. With an average of 13 update messages after each query, Foreseer pays a modest cost for its good search performance as seen in the previous sections. Furthermore, by using Bloom filters, the

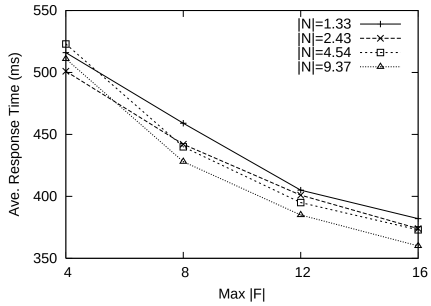

Fig. 10. Sensitivity of search performance to peer's number of neighbors and friends, in terms of response time.

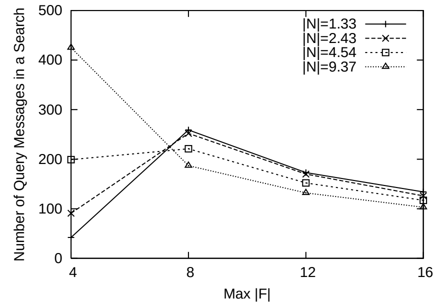

Fig. 11. Sensitivity of search cost to peer's number of neighbors and friends, in the number of query messages produced.

update messages are quite small and do not consume much network bandwidth. For an object addition, a peer only needs to transmit the locations of changed bits in its content filter. Suppose $T = 100$ unique keywords can be extracted from this document and $k = 8$, $m = 8KB$ for the Bloom filter. Each changed bit requires $B = 2Bytes$ to specify its location in the filter. The information to be sent is limited by $L \leq T \times k \times B = 1.6KB$ $bits$, which can be easily packed in few IP packets.

5.4 Scheme optimization

We studied Foreseer's sensitivity to the number of neighbors and friends by running 20,000 queries on a 10,000-peer network with various configuration parameters. Since peers keep making new friends after their queries until they have the maximum number of friends, the upper bound of friends (Max $|F|$) indicates the number of friends each node maintains in the system. On the other hand, a peer may have a lower bound number of neighbors and will not look for new neighbors until some of its current neighbors depart. We collected the number of neighbors for each node and computed the average value as $|N| = 1.33$ for $n_{min} = 1$ and $n_{max} = 5$, $|N| = 2.43$ for $n_{min} = 2$ and $n_{max} = 10$, $|N| = 4.54$ for $n_{min} = 4$ and $n_{max} = 20$, $|N| = 9.37$ for $n_{min} = 8$ and $n_{max} = 40$. Due to space limitation, we only present the results in terms of response time for these configurations as the evaluation of search performance, as shown in Figure 10. The results for other metrics follow the same trend. Similarly, the number of query messages produced in a search is plotted in Figure 11 to show the search cost with various configurations. We noticed that when $Max|F| = 4$, a large portion of queries failed because the query could only reach a small number of nodes due to the upper bound for the number of friends. When $Max|F| > 4$, as shown in the two figures, the search performance keeps increasing, and the number of query messages produced keeps decreasing as more neighbors and more friends are allowed on each peer. This is straightforward since a peer can have the content filters of more peers and outgoing links if allowed to maintain more neighbors and friends. However, a larger number of neighbors and friends results in more indices update workloads, as shown in Figure 12.

Figure 13 shows the number of free-riders touched in each query when varying the number of neighbors and friends in our experiments. It is notable that when $Max|F| = 8$, a query, on the average, touches less than 23 free-riders, indicating that some of

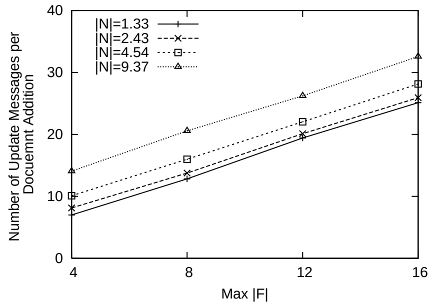

Fig. 12. Sensitivity of indices maintenance cost to peer's number of neighbors and friends, in the number of update messages.

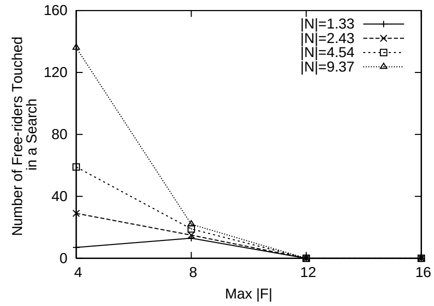

Fig. 13. The number of free-riders touched in a query with varying number of neighbors and friends.

the queries are not resolved until they propagate along the neighbor links. However, when $Max|F| \geq 12$, the query will never touch the free-riding peers, which implies that all the queries have been resolved within the friend overlay. This also indicates that the number of friends is not necessarily too large, considering the maintenance cost of updating a peer's content filters.

6 Conclusions and Future Work

In this paper, we propose a new P2P system architecture called Foreseer, which constructs two orthogonal overlays based on geographical and temporal localities and maintains distributed indices for objects shared on peers' neighbors and friends. By selectively directing searches along the friend links and neighbor links, Foreseer achieves a high search efficiency with a modest maintenance overhead. We conduct a comprehensive set of trace-driven simulations and perform an in-depth analysis of the results. Our experiments show that Foreseer can boost the search performance by up to 70%, with regard to response time and relative distance, and slash the search cost by up to 90% in terms of the number of query messages produced and nodes touched, compared with state-of-the-art P2P systems. In future work, we will study the system performance of Foreseer when different searching policies are employed and/or standard information retrieval traces are fed. In addition, we would like to measure its benefits for partial keyword search applications with multiple responses required.

References

1. Napster: (http://www.napster.com)
2. Milojicic, D.S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z.: Peer-to-peer computing. Technical Report HPL-2002-57, Hewlett Packard Laboratories (2002)
3. Gnutella: (http://www.gnutella.wego.com)
4. Adamic, L.A., Lukose, R.M., Puniyani, A.R., Huberman, B.A.: Search in power-law networks. Technical report, Physical Review E, vol.64(4), 046135 (2001)
5. Lv, C., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and replication in unstructured peer-to-peer networks. In: Proceedings of 16th ACM International Conference on Supercomputing(ICS'02),New York, NY. (2002)

6. Yang, B., Garcia-Molina, H.: Efficient search in peer-to-peer networks. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria,. (2002)
7. Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., Shenker, S.: Making gnutella-like p2p systems scalable. In: ACM SIGCOMM'03, Karlsruhe, Germany. (2003) 407–418
8. Sripanidkulchai, K., Maggs, B., Zhang, H.: Efficient content location using interest-based locality in peer-topeer systems. In: IEEE INFOCOM'03, San Francisco, USA. (2003)
9. Kalogeraki, V., Gunopulos, D., Zeinalipour-Yazti, D.: A local search mechanism for peer-to-peer networks. In: Proceedings of the 11th international conference on Information and knowledge management, (CIKM'02), McLean, Virginia, USA. (2002) 300–307
10. Tsoumakos, D., Roussopoulos, N.: Adaptive probabilistic search (aps) for peer-to-peer networks. In: Proceedings of the 3rd IEEE International Conference on P2P Computing. (2003)
11. Crespo, A., Garcia-Molina, H.: Routing indices for peer-to-peer systems. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria,. (2002)
12. Fessant, F.L., Handurukande, S., Kermarrec, A.M., Massoulie, L.: Clustering in peer-to-peer file sharing workloads. In: Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS), San Diego, USA. (2004)
13. Gummadi, K.P., Dunn, R.J., Saroiu, S., Gribble, S.D., Levy, H.M., Zahorjan, J.: Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: the 19th ACM Symposium on Operating Systems Principles (SOSP-19), Bolton Landing, NY. (2003) 314–329
14. Watts, D.: Small worlds : the dynamics of networks between order and randomness. Princeton University Press, Princeton (1999)
15. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. In: Communications of the ACM, 13(7). (1970) 422–426
16. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking **8** (2000) 281–293
17. Saroiu, S., Gummadi, P.K., Gribble, S.D.: A measurement study of peer-to-peer file sharing systems. In: Proceedings of Multimedia Computing and Networking (MMCN), San Jones, CA. (2002)
18. Sen, S., Wang, J.: Analyzing peer-to-peer traffic across large networks. In: Proceedings of the second ACM SIGCOMM Workshop on Internet measurment, Marseille, France. (2002) 137–150
19. Tomasic, A., Garcia-Molina, H.: Query processing and inverted indices in shared-nothing document information retrieval systems. VLDB J. **2** (1993) 243–275
20. Zegura, E.W., Calvert, K.L., Bhattacharjee, S.: How to model an internetwork. In: Proceedings of the IEEE Conference on Computer Communication, San Francisco, CA. (1996) 594–602
21. Limewire: (http://www.limewire.org)
22. Edonkey: (http://www.edonkey2000.net/)
23. Reynolds, P., Vahdat, A.: Efficient peer-to-peer keyword searching. In: Proceedings of International Middleware Conference. (2003) 21–40
24. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany. (2001) 329–350