Probabilistic File Indexing and Searching in Unstructured Peer-to-Peer Networks

An-Hsun Cheng
Department of Information Management
National Taiwan University
Taipei, Taiwan
r90052@im.ntu.edu.tw

Yuh-Jzer Joung
Department of Information Management
National Taiwan University
Taipei, Taiwan
joung@ccms.ntu.edu.tw

Abstract

We propose a simple, practical, yet powerful index scheme to enhance search in unstructured P2P networks. The index scheme uses a data structure "Bloom Filters" to index files shared at each node, and then let nodes gossip to one another to exchange their Bloom filters. In effect, each node indexes a random set of files in the network, thereby allowing every query to have a constant probability to be successfully resolved within a fixed search space. The experimental results show that our approach can improve the search in Gnutella by an order of magnitude.

1 Introduction

Thanks to the advance of computing and network technology, Peer-to-Peer (P2P) is becoming a popular way for file sharing. A legend of such P2P applications is Napster, which at one time attracted millions of users to share MP3 files concurrently [5]. Since then, many popular P2P networks have been built on top of the Internet, and an enormous number of files can now be directly accessed and downloaded by a simple mouse click.

Depending on how file indexing is implemented, the architecture of P2P file sharing systems can be divided into the following categories: *centralized*, *hybrid*, and *fully distributed*. In centralized systems, a dedicated, centralized server is employed to provide indexing functions for the files to be shared. A user can query the server to find a specific file and its hosting peer, and then connects to the peer to access the file. Centralized P2P systems, in general, are easier to build, but are difficult to scale and vulnerable to attacks and failures.

In hybrid systems, the index scheme in the centralized systems is decentralized and distributed into a number of servers. For example, in KaZaa some 'supernodes' are selected to act as index servers. Ordinary nodes connect to the supernodes, and transmit the information of the files they wish to share to the supernodes. All queries from ordinary nodes are transmitted to the supernodes, and routed through supernodes. Once the desired file is located, download is done directly between the querying node and the source node. The success of these hybrid systems relies on the stability and the performance of the servers and on the cooperation among them. Still, the servers could become the target to paralyze the network.

In a fully distributed P2P system, no separate server is used. All the participating peers must cooperate together to provide the index service. To do so, each participating peer establishes a connection with some other peers, creating an overlay network over which query messages can be forwarded and answered. Depending on the structure of the network, fully distributed P2P systems can be further divided into two types: *structured* and *unstructured*.

In the structured category, the topology of the overlay resembles some data structure, e.g., tree, grid, ring, hypercube, mesh, etc. Each peer is mapped to a node in the data structure by hashing certain strings, such as IP. The data structure also determines for each node its neighboring nodes to which it logically connects. The collection of this neighboring nodes information then determines how a message routes from a given source to a destination. Similarly, file indexing is provided by hashing a file to a node in the data structure. The node is responsible for the file so that all queries to the file will be directed to the node. So, searching for a file becomes simply a message routing on the overlay from the querying node to the node handling the file. Because such systems are effectively maintaining a hash table for the mapping between objects and nodes, they are also commonly referred to as *Distributed Hash Tables (DHTs)*. Examples include CAN [10], Chord [17], Tapestry [20], and Pastry [14].

Structured P2P systems have several advantages. For example, query messages in Chord and Pastry are guaranteed to reach destinations in $O(\log n)$ steps, where n is the total number of nodes in the overlay. The storage requirement for routing per node is also $O(\log n)$. Moreover, because of the nature of hash functions, the indexing load can be uniformly distributed to all peers in the network. However, nodes are closely coupled in a structured P2P system. Maintaining the routing table at each node is usually not an easy task, especially when nodes may join and leave the network frequently. Although current research has shown that it is possible to achieve high performance and reliability with low maintenance cost even in an adversarial condition [11], it is still unclear whether structured P2P architectures can be successfully deployed over the Internet.

On the other hand, peers are loosely coupled in an unstructured network. The (logical) link between two nodes is

0-7803-8430-X/04/$20.00 ©2004 IEEE

established in a more casual way, usually when one knows the existence of the other. Because little cost has been paid to maintain the network (other than connectivity), unstructured P2P networks are very resilient to external and internal failures, and to frequent joins and leaves of peers. Several successful systems have been deployed over the Internet, e.g., Gnutella and Freenet [4]. However, because no structure is assumed in the topology, a node has virtually no knowledge about which node may resolve the query. So searching a file is more or less an exhaustive process, from some initiating node to the entire network, usually in a *breadth-first-search* style (e.g., broadcast in Gnutella) or in a *depth-first-search* manner (e.g., the sequential search in Freenet. In practice, some TTL (Time-To-Live) is set to limit the search space, so as not to flood the network. The search methods, although inefficient and unscalable, are very simple and easy to implement. They are actually quite effective when queried files are popular and one copy is nearby. But search cannot be guaranteed in bounded TTL, and search for rare files is extremely difficult. Several techniques and heuristics have been proposed to reduce search space in the broadcast and to increase the performance [19, 9, 6, 15, 1], but none of them is effective in finding files that are far away from the querying node.

In this paper we propose a probabilistic file index and search scheme for unstructured P2P networks. We aim to preserve the simplicity and robustness of the networks, while offering a file index scheme that satisfies the following properties:

- If the desired file exists somewhere in the network, then our search algorithm is able to locate it with high probability, even though there might be only one copy of the file in the entire network.
- The index scheme is simple and easy to maintain, and is also quite resilient to failures.
- The response time of each query is fast, while the memory and bandwidth consumption are kept at a moderate level, thereby allowing the system to scale.
- The load of each node is balanced.
- The index scheme facilitates keyword search.

The rest of the paper is organized as followed. Section 2 presents our system model and the main data structure. Section 3 presents our probabilistic index scheme, and Section 4 presents search algorithms for the scheme. Simulation results of the index and search scheme is presented in Section 5. Finally, Section 6 concludes and provides some directions for future work.

2 Preliminaries

In this section we present our system model, the problem, and the main data structure used in our index scheme.

2.1 System Model

We can model an unstructured P2P overlay as an undirected graph $G = (N, E)$, where N is a set of nodes, and E is a set of edges. Each node represents a peer participating in this network. An edge between two nodes means that the two nodes are neighbor to each other in the overlay. The neighboring node information is maintained by each node u in its neighbor table NT_u.

Every node has the ability to communicate directly with any other node in the overlay, just like that a host can communicate directly with any other host in the Internet by forwarding packets to a proper IP address. However, in order for a node u to be able to communicate with another node v, u must know the existence of v and its contact information (just like that in the Internet, u must know the IP of v). A node always knows the existence of its neighbors (and being able to communicate with them), but it may know more.

We assume that the overlay network is maintained by the underlying P2P protocol. Our probabilistic file indexing and searching is built on top of this overlay. Because nodes may join and leave the network dynamically, the topology may change over time, possibly resulting in partitions. However, recent research [13] has shown that the number of connected components is relatively small: the largest connected component always includes more than 95% of the active nodes. Hence, we shall assume that the entire network is connected in the remaining sections.

Each node in the overlay is home to a set of files that are available to other nodes. The *file locating problem* then is that, given a query containing some keywords, returns a set of nodes that have files matching these keywords.

2.2 Bloom Filters

Bloom filters [3] are compact data structures for a probabilistic representation of a set of objects. A Bloom filter is a bit array B of length m with k independent hash functions, which map elements of the set to an integer in $[0, m)$. To construct a filter for a set, each element of the set is hashed with k functions, and the bits corresponding to the hashed results are set. Therefore, if a_i is an element of the set, in the resulting Bloom filter B, all bits corresponding to the hashed results of a_i is 1. To determine whether an element e is in the set, e is hashed with the same k functions. If all the corresponding bits of the hashed results in the Bloom filter are set, the represented set *may* contain e. If any one of the bits is not set, the set must not contain the element. Bloom filters exhibit a *false positive* phenomenon, meaning that they may return a truth value to a query even thought the queried element is not in the set. Let m be the length of a Bloom filter, n be the size of the represented set, and k be the number of hash functions used. Then, the probability of false positive f can be calculated as follows,

$$f = (1-p)^k$$

where $p \approx e^{-kn/m}$.

Bloom filters are quite useful in supporting keyword search [7]. Recall that in our system model each node shares some files to others. We shall assume that each file is summarized by some keywords. The collection of keywords of all the files shared by a node v is then represented by a Bloom filter B_v. So by first querying the filter, one can see if a queried file could be at the node. An actual search to the

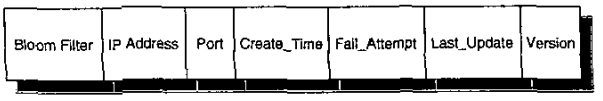


Figure 1. Data format of a node's Bloom filter.

node's directory is performed only if the query is positively answered. To allow copies of this filter to be distributed to other nodes, each copy is attached with some tags: *IP*, *Port*, *Create_Time*, *Fail_Attempt*, *Last_Update*, *Version* (see Figure 1). IP address and access port allow other nodes to contact with the node when a query is performed outside the node. The other tags are used to maintain a fresh version of the Bloom filter. They will be explained in Section 3.2. In the rest of the paper, by Bloom filters we mean the filers of the format in Figure 1.

3 A Probabilistic File Index Scheme

Recall from Section 2.2 that each node u has a Bloom filter B_u summarizing the files it shares. If B_u is replicated and distributed to other nodes in the network, then every node v that has B_u can answer queries about whether u might have a particular file one is looking for. If the answer is yes, v can ask u to perform an actual check to its local directory to see if u does have the file.

In general, the replication allows a query to u's file to be answered by any node having a copy of B_u. So if every node has a copy of B_u, then file search can be efficiently performed in the system. However, for the system to scale, only a limited number of replicas of B_u can be distributed. In practice, the number of replicas should allow a query initiating from a node to be answered within a reasonable search space. Here, the search space refers to the set of nodes to be visited for the query. It corresponds directly to the set of (replicas of) Bloom filters to be searched during the query resolving process.

However, the distribution of a node's Bloom filters may not be so uniform to let the search space starting from a node v to contain a copy of a particular B_u. As a result, a query to u's files will not be resolved at v. To solve this problem, we let nodes dynamically exchange their Bloom filters so that the search space of each node is also dynamically changing. This then allows every query from a node to have some probability to be successfully resolved, regardless of the time the query is issued. By properly setting the system parameters, we can tune the probability to be reasonably high.

3.1 Construction and Maintenance of the Index Scheme

The index scheme at each node consists of four parts: initialization, rejoin, leave, and gossip.

Initialization

This process is performed when a node u joins the network for the first time. It creates a Bloom filter B_u of the format in Figure 1 to summarize the files it shares. Then, it replicates d copies of the filter and stores them in an index table IT_u it maintains at its site.

Leave

Before a node u leaves the network, it must distribute all of the entries in its index table to its neighbors (presumably in a uniform and balanced style). This ensures that the Bloom filters in its index table are still available while the node is offline.

Rejoin

When a node rejoins the network, it asks its neighbors to distribute some entries in their index tables to the node to fill up its index table.

Gossip

The kernel of the index scheme is a gossip protocol for exchanging index table entries between two nodes, so as to disseminate node information. The protocol is performed at each node u in every δ time units as follows:

1. Node u uniformly selects an entry from its index table IT_u with probability p, or selects an entry from its neighbor table NT_u with probability $(1-p)$, where p is a system wide parameter. We say that an entry in IT_u *represents* v if the entry is a Bloom filter B_v of v. Similarly, an entry in NT_u *represents* v if the entry describes node v.

   Note that in the above step if the selected entry represents u itself, then u must repeat the step until an entry representing another node is obtained.

2. Let v be the node represented by the selected entry. Node u informs v that they are going to exchange Bloom filters in their index tables.

3. Node u then randomly selects $\lfloor s * |IT_u| \rfloor$ entries from IT_u (where $|IT_u|$ denotes the size of IT_u), and similarly node v also randomly selects $\lfloor s * |IT_v| \rfloor$ entries from IT_v. The variable s, $0 < s < 1$, is again a system parameter.

4. Both u and v send the selected entries to the other to exchange their information.

3.2 Remarks

There are several issues left to be addressed in the index scheme, including the prohibitive setting of p, obsolete information, load balancing, information update, and node failure.

Prohibitive Setting of p

We note that the system parameter p in the gossip protocol cannot be set to 1. To see this, we can use a directed graph to model the relationship among nodes according to their index tables: there is an edge from node u to v if u holds a copy of v's Bloom filter in IT_u. If p is set to 1, then a node u can exchange Bloom filters with v only if (u, v) is an edge in the graph. The exchange might remove (u, v) from the graph if u sends v its only copy of B_v. This then might break any link from u to v, resulting in a partition in which u and v belong to different blocks. Since u can only exchange Bloom filters with nodes in the same block, once a partition occurs, there is no way for the partition to heal. Moreover, the partition might get worse (resulting in more blocks), and eventually drive the system into the initial state in which every node has only its own Bloom filters!

Setting p to be less than 1 allows u to exchange Bloom filters with its neighbors in the overlay. Recall that the overlay is connected. So the exchange allows u to obtain information from other blocks should partitions occur, and therefore to heal the partitions.

Obsolete Information

In a P2P network, nodes may fail or go offline permanently. So their Bloom filters need to be removed from the system. If a node u discovers that another node v is not available (when u attempts to contact with v for some information), u increases the counter *Fail_Attempt* (see Figure 1) in its copy of B_v by one. If the counter exceeds some constant, u can simply discard B_v from its index table. On the other hand, u resets the counter to zero if it has successfully contacted with v.

Load balancing

The departure of a node u may affect the load of other nodes because u will distribute all the entries in its index table to its neighbors. Moreover, after u leaves the network, some nodes will detect its leaving, and then delete u's Bloom filters from their index tables. So u's departure causes some nodes to increase their index tables, while some others to decrease. However, our gossip protocol has the ability to balance the load. This is because in each gossip operation, the node that has more entries will send more to the other, thereby decreasing its index table while increasing that of the other.

Information Update

The files shared by a node u may change over time, and therefore u's Bloom filters need to be updated across the network. To do so, the tag *Create_Time* in a Bloom filter B_u records the time at node u the Bloom filter is created. Then, we let nodes holding a copy of B_u periodically check with u the status of their copies using the tag *Create_Time* in the copies, and to update their copies if needed. The timestamp *Last_Update* in a Bloom filter B_u records the latest time at which the status of the copy is checked. It is the local time of the node (say v) at which the copy resides. Node v checks with u if its local clock exceeds the timestamp by some threshold t_{update}. Moreover, when v sends the copy to another node w in the gossip protocol, w needs to translate the timestamp in the copy (which was set according to v's clock) to w's clock.

Node Failure

In the above index scheme we assumed that nodes will distribute their index tables to their neighbors when they leave the network. This process cannot be guaranteed if nodes fail during the operation. As a result, the Bloom filters in their index tables will be lost. Another situation that causes a node's Bloom filter to be lost is when other nodes attempt to contact with the node while the node is offline. After a number of fail attempts they may discard their copies of the node's Bloom filter. The tag *Version* in a Bloom filter is used to resolve this problem. The tag consists of two values: $\langle$*replica_no*, *generation*$\rangle$. When a node u creates d copies of B_u, it numbers these copies in the *replica_no* field. For each copy number, u also writes into *generation* an integer recording the number of times u has generated B_u.

Recall that every node that has a copy of B_u will check with u for the status of u in every t_{update} units. So u can determine if a particular copy is lost if no node has checked the copy with u for a certain time. If this is the case, u generates in its index table a new copy of B_u with the same *replica_no* as that of the lost one. However, u increments *generation* in the new copy. This new value allows u to tell which copy of the same *replica_no* is new should the old copy still exist in the network (the owner of the old copy fails to check with u its status due to, say, some network problems). When a stale copy is detected, u can simply ask the possessing node to discard the copy.

4 Search Algorithms

In this section we discuss how search can be performed on the above probabilistic file index scheme. We begin with a basic search algorithm that resolves a query with some high probability. The probability can be increased to approach to one by performing multiple instances of the basic search. Two such algorithms are presented at the end of this section.

4.1 Basic Search Algorithm

The search algorithm is almost the same as in Gnutella. A node initiating a query broadcasts the query to its neighbors with nonzero TTL. A query contains keywords such as "NBA", "Jordan", "2003", etc. Nodes which receive the query message check the TTL to determine whether or not to broadcast the query message. (If they have already received and processed the messages before, then they simply discard the messages.) If the TTL is less than one, they drop the messages; otherwise, they decrease the TTL by one and broadcast the queries to their neighbors.

By examining the Bloom filters in their index tables, the receiving nodes can know whether some particular node might have the desired files. Suppose node u finds that v

might have the files. Node u then forwards the query message to v with TTL set to -1. Note that when a node receives a query, if the TTL is no less than zero, both the local directory[1] and the Bloom filters in its index table will be examined. Otherwise (TTL is -1), only local directory will be checked. If some matched files are found there, the reply message goes through the same path, but in the reverse direction, back to the querying node as the way query replies are routed in Gnutella (or goes directly to the querying node if one wishes).

4.2 Analysis of the Basic Search Algorithm

We analyze the probability that a query will be resolved in the basic search algorithm. Suppose that Bloom filters of the nodes are randomly and uniformly distributed over the overlay network. Let n be the number of nodes in the network. Assume that each query message is sent with TTL h. Let $N(u,h)$ denote the set of nodes that are within h hops from node u. Suppose that u initiates a query for a file that exists at node v. Let A be the event that the query is resolved, and let $\bar{A}$ denote the event otherwise.

There are n^d ways to distribute the d replicas of v's Bloom filter B_v to the n nodes. The event $\bar{A}$ occurs if none of the nodes in $N(u,h)$ currently holds a replica of B_v while the query is being resolved. There are $(n-|N(u,h)|)^d$ ways to distribute the d replicas of B_v so that none of the nodes in $N(u,h)$ gets a copy. Therefore, the probability that event $\bar{A}$ occurs is:

$$Pr(\bar{A}) = \frac{(n - |N(u,h)|)^d}{n^d} = (1 - \frac{|N(u,h)|}{n})^d$$

Then,

$$Pr(A) = 1 - (1 - \frac{|N(u,h)|}{n})^d \quad (1)$$

In practice, $|N(u,h)|$ is relatively small compared to n. So $Pr(A) \approx 1 - (1 - \frac{d*|N(u,h)|}{n}) = \frac{d*|N(u,h)|}{n}$. Hence, if $|N(u,h)|$ is fixed, d grows linearly with n in order to retain equal success probability. Table 1 gives a numerical illustration of $Pr(A)$ for some values of n, d, and $|N(u,h)|$. One can see that when B_u is distributed to 0.4% of the population of the network, the probability that a query can be successfully resolved is about 0.8 if the query is broadcast to 0.4% of the nodes (which, as we shall see in the simulation, is within TTL = 4 in a typical network).

4.3 Multiple-Query Algorithms

If one wishes to increase the success probability of a query without increasing either d (the number of replicas of a Bloom filter) or $|N(u,h)|$ (the search space), then multiple queries can be performed. There are two alternatives:

Repeated Queries: If a query fails to return an answer, the user can wait for a period of time and issues the same query again. Due to the gossip protocol, if the time between the two queries is long enough, the second query will have an equal probability to be resolved as the first one (because Bloom filter replicas have been randomly re-distributed over the network). If the second one also fails, the a third query can be issued, and so on. If the desired file exists in the network, then the probability that the queries continue to fail approaches to zero. For example, consider the setting of $d = 400$ and $|N(u,h)| = 400$ in Table 1. The probability that three continuous queries all fail to return a file is $(1 - 0.7988)^3 \approx 0.0081$. That is, 99.18% of the queries will be successfully resolved in three continuous queries.

The problem, of course, is how long a retry should wait? We will answer this question in Section 5.

Surrogate Queries: If time is critical, then, instead of waiting for a retry at the same node, one can ask a randomly selected node to execute the retry. The node can be selected from the index table of the original querying node. The motivation behind this approach is that the search space of the randomly selected node should be independent of the search space of the original node. Therefore, the second query will have an equal probability to be resolved as the first one; and similarly for the third, the fourth, etc.

$n = 100000$				
$\lvert N(u,h)\rvert$ \ d	50	100	200	400
200	9.52%	18.14%	32.99%	55.10%
400	18.16%	33.02%	55.14%	79.88%
800	33.07%	55.21%	79.93%	95.97%

Table 1. The probability of a successful query under various n, d, and $|N(u,h)|$.

5 Simulations

There are several important parameters left to be addressed in the index scheme and in the gossip protocol. Determination of their appropriate values are nontrivial and crucial to the performance of the query algorithms on the index scheme. In this section we conduct some experiments to address these issues. The simulation programs are written in Java.

5.1 Network Topology

According to [13], unstructured P2P networks like Gnutella tend to exhibit power-law distribution. Therefore, we use BRITE [18] topology generator to construct an overlay network where the distribution of the number of links incident to a node follows the power-law distribution. The network consists of 89000 nodes with average two links per node. We believe that a network of this scale is typical in real unstructured networks. All simulations presented in the section were conducted on the network. Moreover, unless

[1] When searching the local directory, the node's own Bloom filter can be used to filter out unnecessary disk access.

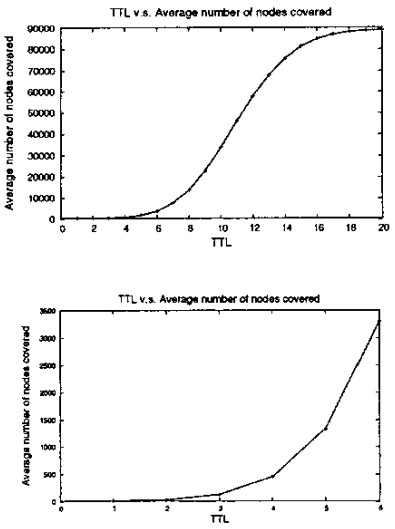


Figure 2. Search space vs. TTL.

System parameter	Meaning
n	The size of the network
δ	The time between two gossip operations
d	The number of copies a node distributes its Bloom filter to the network
p	The probability to select an entry from the index table (while $(1 - p)$ being the probability to select an entry from the neighbor table)
s	The percentage of entries in an index table to be exchanged in each gossip

Table 2. System Parameters

stated otherwise, nodes do not leave or join the network during the simulation.

To see the average number of nodes covered within a certain TTL, we randomly sample 2000 nodes from the network, and calculated the average number of nodes covered within each TTL. The result is plotted in Figure 2.

5.2 Gossip Performance

Because our index scheme relies on the ability of the gossip protocol to disseminate information, it is important to understand the performance of the gossip protocol under various configurations. The gossip protocol defines several system parameters. For ease of reference, they are summarized in Table 2.

5.2.1 System Initialization

In this experiment, we study the time the gossip protocol takes to make an initial network stable. Here, 'stable' means that Bloom filters of all nodes have been uniformly disseminated over the network. In general, if the network is stable, then the expected search size for a node to find a copy of another node's Bloom filter should be n/d.

We start a network of 89000 new nodes, each of which shares a unique file. Then, we let nodes gossip to each other to disseminate their Bloom filters. We count how many iterations of gossip are required to make the network stable. After every 10 gossip operations per node, we generate several queries to search files. By measuring the mean search size of the queries, we can see if the network is stable. The results are shown in Figure 3(a).

From Figure 3(a), one can see that d does not affect the converging time much. Our gossip protocol needs about 2500 gossip operations per node to make the network stable. If δ is 20 minutes, it takes about 35 days to converge. Fortunately, the bootstrapping process only needs to be executed once.

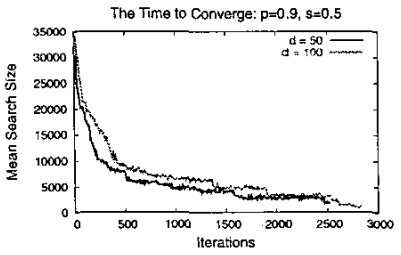
Still, there is much room for improvement. But, first, let's see why the gossip protocol takes such a long time to converge. We observe that we can envisage an *index graph* induced by the index tables as follows: there is an edge (u, v) if u holds a B_v in its index table. Then, setting p close to 1 means that the gossip protocol tends to use the index graph to exchange Bloom filters, rather than using the underlying overlay network. Note that initially every node has nothing but its own Bloom filters in its index table. So, at the beginning, the index graph consists of 89000 disconnected components, each of which is a singleton. Due to the protocol, each node in its first gossip operation must find a neighbor in the overlay to exchange Bloom filters (because each node must select a node other than itself to exchange information). This then extends the singleton of the node to a component of two. When a component is not a singleton, the high value of p means that each node in the component tends to select a node in the same component, rather than a neighboring node from the underlying overlay, to exchange information. Clearly, the former has no help to extend the component, while the later has, as the neighbor (w.r.t. the overlay) may be in a different component (w.r.t. the index graph). So, when p is large, the chance to merge two connected components is low. Hence, it takes a long time to merge all the initial singleton components.

To justify our argument, we also measure the converging time for other values of p: 0.1 and 0.5. The results are shown in Figure 3(b). One can see that small p significantly improves the performance, as now a node tends to use the underlying overlay network to exchange information. So from this experiment we see that p should be set small during the system initialization phase.

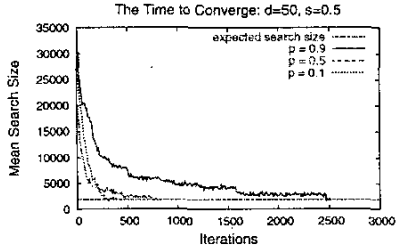
5.2.2 Propagation Time of New Information

In this experiment, we study the performance of our gossip protocol by measuring the interval from the time a new node joins a stable network to the time its Bloom filters have been uniformly disseminated over the network. This measurement indicates how long some remote users can utilize files brought into the network by a new member.

We start a network of 88800 nodes and let them gossip to each other until the network becomes stable. Then, 200 new members join the network simultaneously, and each of them has a unique file to be shared. We count how many gossips are required until the network becomes stable again. To do so, we randomly choose some nodes and generate queries to search files owned by the new members, and then calculate the mean search size for those queries. If the network is



(a) When $p = 0.9$, $s = 0.5$, and $d = 50$ and 100.



(b) When $p = 0.1, 0.5, 0.9$, $s = 0.5$, and $d = 50$.

Figure 3. The converging time of an initial network.

stable, the mean search size should be close to the expected search size n/d.

In the first part of the experiment we study the performance with respect to various p and s, given a fixed $d = 50$. Figure 4 presents the results of the simulation. One can see that, with appropriate configuration, our gossip protocol is quite fast to disseminate new information. For example, Figure 4(b) shows that if we set $p = 0.9$ and $s = 0.5$, our gossip protocol needs only 6 iterations to stabilize the network. From Figure 4(a)-(c) we can see that a large p results in better performance than a small p does. To see this, recall that a large p means that nodes tend to use the index graph induced by the index tables to exchange Bloom filters. Note that the topology of the index graph varies after each gossip operation. In contrast, the underlying overlay network is static (as no node joins or leaves). So, to each node, any node selected from its index table to exchange Bloom filters tends to be a random node in the underlying overlay network; while any node selected from its neighbor table is always its neighbor in the overlay. When the network is stable, the index graph should be connected. As a result, information disseminates quickly in the index graph than in the overlay. So, when a new node joins a stable network, the first gossip operation by the node will connect the node to the existing index graph. Thereafter, its Bloom filters will be disseminated to the network quickly if we have a large p setting. The results in the simulation indicates that setting p in between 0.5 and 0.9 is recommended.

From Figure 4(d)-(f) we see that setting s to 0.5 yields better performance than setting s to 0.1 or 0.9. The reason is as follows. The time to stabilize the network after a new node joins depends on the time to disseminate the d Bloom

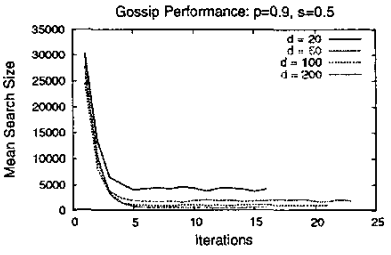


Figure 5. Gossip performance when $s = 0.5$, $p = 0.9$, and $d = 20, 50, 100, 200$.

filters of the new member to the network. Recall that initially all the d Bloom filters are in the new member's index table. If s is set to 0.5, after the first gossip operation the d entries is divided into two equal parts, one hold by the new member itself, the other by an existing node. Each of these two parts tends to be partitioned into a set of $1/4d$ size in the next gossip by its holder, and so on. On the other hand, if s is set to 0.9 or 0.1, the d entries of the new member tend to be partitioned into two unbalanced sets in each gossip operation, with one significantly larger than the other. So, it takes longer time to divide the original d-element set into pieces to be hold by different nodes.

Since setting $p = 0.9$ and $s = 0.5$ can yield fast propagation time in a stabilized network, we will use this configuration to our gossip protocol in the following experiments.

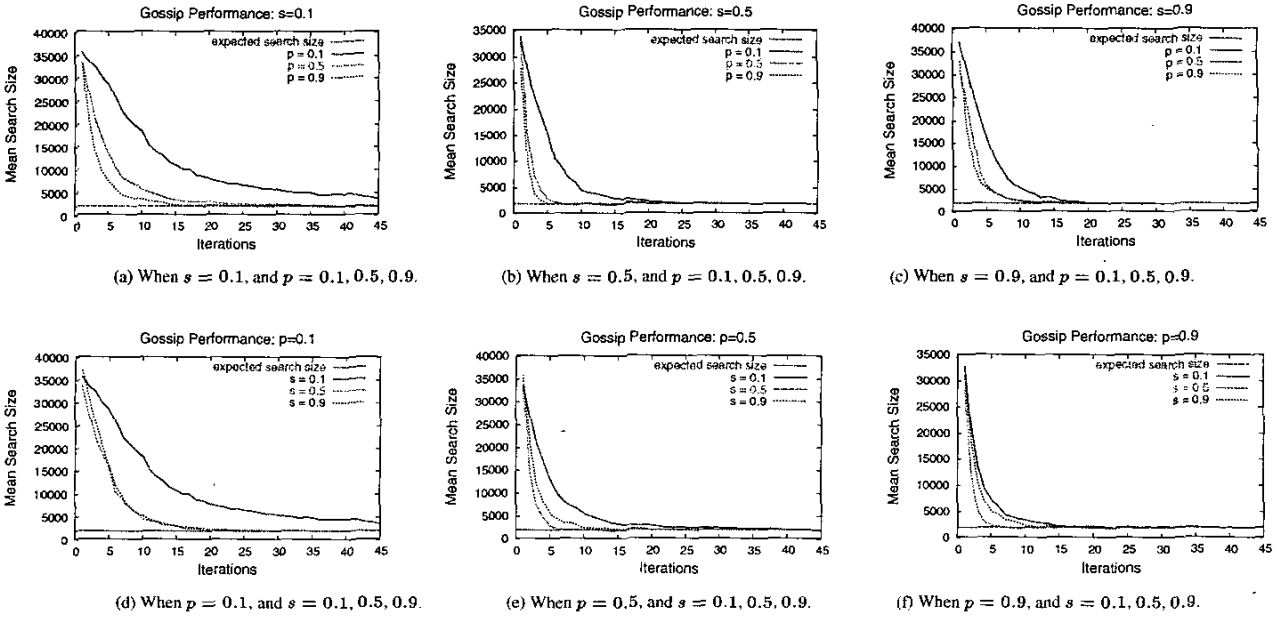
In the second part of this experiment, we study the performance of the gossip protocol with respect to $d = 20, 50, 100, 200$, given that $p = 0.9$ and $s = 0.5$. The results are shown in Figure 5. It takes about 5, 6, 7, and 7 iterations of gossip to make the network stable when d is 20, 50, 100, and 200, respectively. One can see that different settings of d do have some impact on the gossip protocol. This is because the larger d is, the longer time the gossip protocol needs to partition the d Bloom filters of a new member into pieces. However, the impact is not very significant (in logarithmic growth).

5.3 Query Performance

In this section we study search performance in our probabilistic index scheme. In particular, we compare it with Gnutella, which is equivalent to our system without the index scheme.

5.3.1 File Distribution

The experiments in this section need a large amount of real data to configure our simulator, such as the placement of files and the number of files shared per node. However, this kind of data are not publicly available. We instead use the information obtained from the Web proxy logs of Boeing [8] as our simulation data. In Web proxy logs, the field 'user ID' denotes a client and the field 'hostname' represents a WWW server the client requests. So for each entry in the Boeing logs, we extract the two fields and treat 'user ID' as a node,



(a) When $s = 0.1$, and $p = 0.1, 0.5, 0.9$.
(b) When $s = 0.5$, and $p = 0.1, 0.5, 0.9$.
(c) When $s = 0.9$, and $p = 0.1, 0.5, 0.9$.
(d) When $p = 0.1$, and $s = 0.1, 0.5, 0.9$.
(e) When $p = 0.5$, and $s = 0.1, 0.5, 0.9$.
(f) When $p = 0.9$, and $s = 0.1, 0.5, 0.9$.

Figure 4. Gossip performance with respect to various s and p for a fixed $d = 50$.

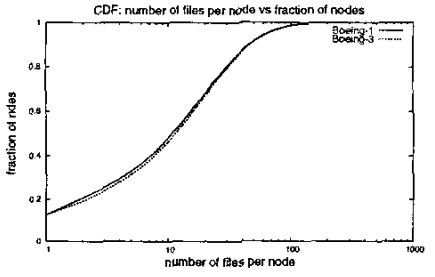


Figure 6. Cumulative density function of the number of files shared per peer. The distributions of Boeing logs in different days are almost the same, so only two of them are shown.

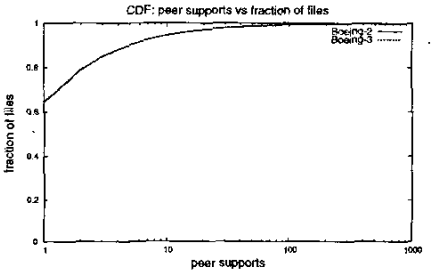


Figure 7. Cumulative density function of peer support.

and 'hostname' as a file. The distribution of the number of files shared per node of this mapping, depicted in Figure 6, is similar to the one shown in [16]. So we believe that the extracted information can represent a typical file distribution in real P2P networks.

Some researches [2, 13] have shown that free riders do exist in many P2P networks. Hence, besides assigning nodes to share files, we also add some nodes that do not share any file into the network. According to [13], they found that about 25% of peers are free riders. The network in our simulation contains 89000 nodes, and about 66000 of them share files to reflect the free-rider phenomenon.

Figure 7 shows the cumulative counts of the fraction of files vs. peer support. The *peer support* of a file is the number of nodes owning this file in the network. To the best of our knowledge, the distribution of peer support has not been analyzed in the literature. Therefore, we cannot verify the validity of our simulation data set. It only serves as a statistic reference. In Figure 7, we observe that 90% of the files belong to rare files (peer support less than 10), and only 1.5% of the files are popular (peer support larger than 100).

In [13], it said that: "While most nodes share few files and maintain only a couple of connections, a small group provides more than half of the information shared, while another, distinct, group provides most of the essential connectivity." In addition to the above remark, there is also a figure in [13] showing the correlation between the number of connections a node maintains and the amount of files a node shares. In the experiment, we simply map the data layer onto the overlay network randomly, and make sure that the

correlation between the number of files a node shares and the number of links a node maintains is similar to the figure in [13].

5.3.2 Performance of Basic Search

To study the performance of the basic search algorithm, we consider four cases: $d = 50, 100, 200$, and 400 (where $p = 0.9$ and $s = 0.5$). We also study the performance of the search algorithm without the index scheme, which is essentially a Gnutella network.

Because the size of the Boeing data set is too large, we evaluate the performance based on a sample of queries. Each query consists of two parts: a "Node ID" and a "File ID". "Node ID" is the initiating node of the query, and "File ID" represents the queried file. In this experiment, we only search files withpeer support 1, 10, and 100. We generate 2000 queries for each peer support, with a total of 8000 queries in the simulation. For each query, we count the number of hops required to locate the first copy of a desired file. The results of the simulation are shown in Figure 8.

From the figure we can see that our probabilistic index scheme improves search results significantly over that in Gnutella. For example, in Figure 8(a), by setting the TTL of each query to 5, 70%, 57%, 45%, 33% of the queries are answered for $d = 400, 200, 100, 50$, respectively. In contrast, only 1% of the queries are successfully resolved in Gnutella.

5.3.3 Performance of Repeated Queries

In Section 4.3 we presented a "Repeated Queries" algorithm that allows a user to wait for some period of time before reissuing the same query at the same node. Here we study the time one should be waiting.

We start a network of 89000 nodes, each of which shares a unique file. The system parameter d is set to 100. We let nodes gossip to each other until the network is stable. Then, we generate queries to search files, with search space equal to 400 nodes per query (about TTL= 4). We can use Equation (1) to calculate the success probability of each query, which is 0.3626 in the setting.

For each unsuccessful query, we reissue the same query after k iterations of gossip per node, and record whether the reissue is successfully resolved or not. The percentage of the successful queries among the second attempts can then be calculated to see if it is close to the success probability. The experiment is conducted for k ranging from 1 to 9, and for each possible value of k, 95000 samples were collected. The success probability under different setting of k is then calculated and plotted in Figure 9.

One can see that after 4 or 5 iterations of gossip, the success probability is close to 0.3626. We offer an explanation for this: In each iteration of gossip system wide, each node will execute two gossip operations on average (one initiated by the node itself, and the other by its neighbor). Recall that s is set to 0.5, which means that half of a node's index table will be refreshed in each single gossip operation. Then, about 75% of the index entries stored at each node will be refreshed in an iteration of gossip. After 4 iterations of gossip system wide, about 99.6% of entries at each node are renewed. Therefore, the index entries stored at each node

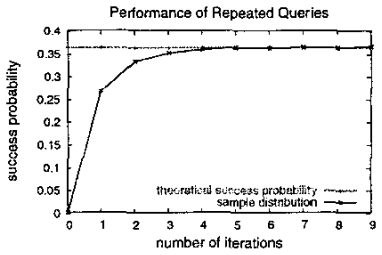


Figure 9. The success probability of repeated queries under different settings of k.

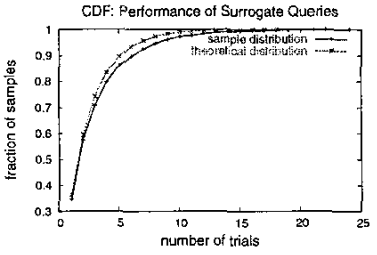


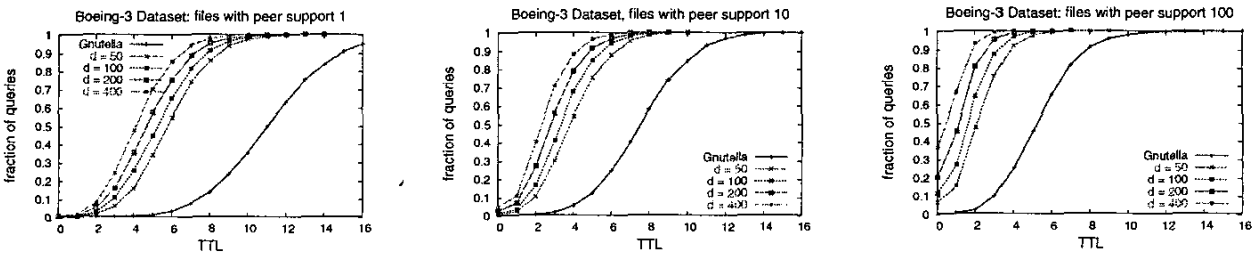
Figure 10. Cumulative number of trials in the "Surrogate Queries" algorithm.

before 4 iterations of gossip are almost independent of the entries stored at the same node after 4 iterations of gossip. Hence, the success probability should be close to 0.3626 after 4 iterations of gossip.

5.3.4 Performance of Surrogate Queries

In this experiment we study how many queries should be reissued in the "Surrogate Queries" algorithm. The setting is similar to the previous one. We start a network of 89000 nodes, each of which shares a unique file. The parameter d is set to 100. We let nodes gossip to each other until the network is stable. Then we generate queries to search files, with search size equal to 400 per query. In theory, the success probability of each query is 0.3626. So the expected number of trials is the reciprocal of the success probability, which is 2.75. For each unsuccessful query, we continue to issue queries using the "Surrogate Queries" algorithm until a desired file is located. We count how many additional queries are required to locate the desired file after the first unsuccessful query. There are a total of 4900 samples collected in this simulation. Each sample is a number recording the number of additional queries issued to locate a desired file. The result of simulation is shown in Figure 10.

The sample mean is 3.103, and the standard deviation of the sample mean is 0.0405. From Figure 10 and the statistic data, one can see that "Surrogate Queries" performs slightly worse than the theoretical estimate. The reason is because the search area among each query might overlap. However,



(a) Peer support equal to 1.

(b) Peer support equal to 10.

(c) Peer support equal to 100.

Figure 8. Basic search performance.

the performance is not too far from the theoretical estimate. Since "Surrogate Queries" can save a lot of time, it is still a feasible alternative to improve search results.

6 Concluding Remarks

In the paper we have assumed that the computing power and the network bandwidth per node are the same. In real networks, some nodes do have much resource than the others. For these nodes, we can enlarge their index tables to give them more responsibility in the index scheme. Moreover, experiences have shown that a query might have higher chance to be resolved by a node that shares many files than by a node that shares few. So we may also let the number of Bloom filters a node inserts into the network be proportional to the number of files the node share. Therefore, query messages will have higher chance to be forwarded to nodes that have many files than nodes that have fewer.

Note that security is treated as a separate issue in the paper. Because peers are quite autonomous, all peer-to-peer networks are vulnerable to various security attacks, e.g., a peer responds with a 'poisoned' file, or blocks queries and fakes search results. All these issues are certainly very complicated and are still an active and ongoing research in the field.

Finally, although in the paper we use Gnutella as our primary example, we believe that our approach can be directly applied to many other unstructured P2P networks without too much modification.

References

[1] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman. Search in power-law networks. *Physical Review E*, 64(4), 2001.

[2] E. Adar and B. A. Huberman. Free riding on gnutella. *First Monday*, 5(10), Oct. 2000.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. Int'l Workshop on Design Issues in Anonymity and Unobservability, LNCS 2009*, pages 46–66, 2001.

[5] CNN.COM. http://www.cnn.com/2001/tech/internet/06/28napster.usage/.

[6] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proc. of CIKM*, pages 300–307. 2002.

[7] A. V. Patrick Reynolds. Efficient peer-to-peer keyword searching. In Middleware 2003, *LNCS 2672*, pages 21–40, 2003.

[8] Boeing proxy logs. http://www.web-caching.com/traces-logs.html.

[9] M. K. Ramanathan, V. Kalogeraki, and J. Pruyne. Finding good peers in peer-to-peer networks. In *Proc. IPDPS*, 2002.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. SIGCOMM*, pages 161–172. 2001.

[11] A. R. Ratul Mahajan, Miguel Castro. Controlling the cost of reliability in peer-to-peer overlays. In Proc. IPTPS, pages 21–32, 2003.

[12] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. INFOCOM*, pages 1248 -1257, volume 3, 2002.

[13] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proc. First Int'l Conference on Peer-to-peer Computing*, pages 99–100, 2001.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. Middleware, LNCS 2218*, pages 329–350, 2001.

[15] K. Sripanidkulchai, Bruce, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *Proc. INFOCOM 2003*, pages 2166–2176, volume 3, 2003.

[16] P. K. G. Stefan Saroiu and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. MMCN '02*, pages 407–418, 2002.

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM*, pages 149–160. 2001.

[18] Boston University. Brite.

[19] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *Proc. ICDCS*, 2002.

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