Posting Protocol for Improved Keyword Search Success in Peer-to-Peer File Sharing Systems

Lenitra M. Clay, Mostafa H. Ammar, Ellen W. Zegura, Russell J. Clark

Georgia Institute of Technology, Atlanta, GA, USA

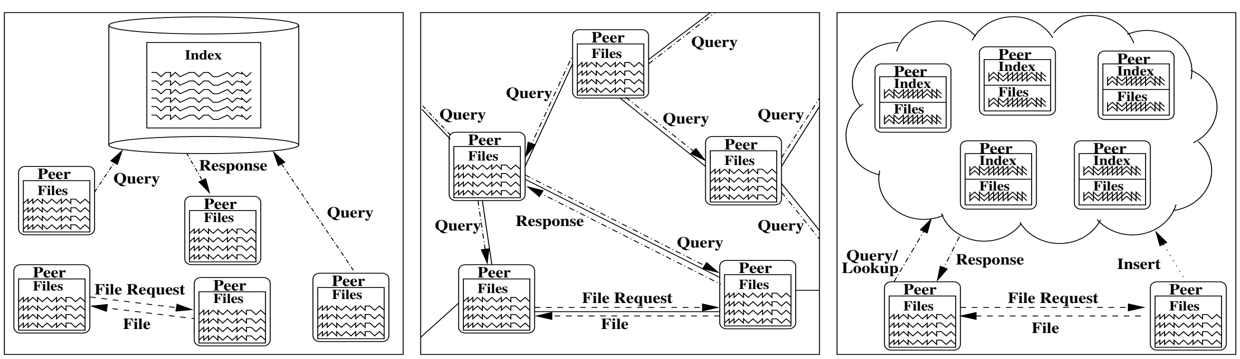
ABSTRACT

Decentralized peer-to-peer (P2P) file sharing systems, where peers query each other for content, are the most dominant in today's Internet. In the unstructured decentralized P2P systems, there is no direct connection between content location and system topology. Searches in such systems are typically broadcast within a limited region of the network and thus may not receive a response if the content is not within that region. Structured decentralized P2P systems provide a connection between content location and the system topology. There, queries can be directed to a peer who can respond definitively. Research in distributed computing has examined the problem of matching a client process to a desired server process. An approach to solving this distributed match-making problem is to have the server "post" or replicate information to other nodes in the system. We modify this approach to be used in the decentralized P2P file sharing environment. In this paper we propose a "posting" protocol to improve the success of searches in the decentralized P2P systems. By having peers replicate keyword information to other peers the search success rate can be increased. We evaluate different posting policies and compare the results for searching with and without posting.

Keywords: Peer-to-Peer Media Systems, Replication, Keyword Searches, Content Distribution

1. INTRODUCTION

There are two models for P2P file sharing over the Internet: centralized and decentralized (See Fig. 1). These models



(a) Centralized (b) Unstructured Decentralized (c) Structured Decentralized

Figure 1. Indexing Models for P2P File Sharing Systems

differ in how the search for content is performed. With the centralized model (Fig. 1(a)), as used with Napster,[1] a client contacts a central server(s) with the search request. The central server(s) is responsible for maintaining an index of all files available in the system and the location of the files. When a search request is received, the central server(s) responds with a list of available files and the addresses of the peers storing those files. The client can then request the desired file directly from one of the peers storing the file. The second model for P2P file sharing is a decentralized one where peers query each other for content. There are two approaches indexing in these decentralized systems: unstructured and structured.[2] In the unstructured decentralized systems (Fig. 1(b)), as seen with Gnutella[3] there is no tie between content location and the topology. Connections between peers are unstructured and each peer maintains only information about the files they share. In order to find a desired file, a search request is broadcast by a client to its neighbors. These neighbors send the request to their neighbors and so on. If the search request reaches a peer who has the desired file that peer sends a response back

Email: {lenitra,ammar,ewz,rjc}@cc.gatech.edu. Address: College of Computing, 801 Atlantic Drive, Atlanta, GA 30332-0280

towards the requester. As with the centralized approach, the file is downloaded directly from the peer storing the file. A Time-to-Live (TTL) field is used to scope the search and prevent the request from traveling through the entire P2P network. An advantage of this decentralized approach is that there is no single point of failure. However this leads to the need for the search requests to be flooded through the Gnutella network. In the structured decentralized systems, distributed hash-table functionality is used to direct queries and distribute the file index (Fig. 1(c)). This approach can be seen in systems such as CAN, Chord, Pastry and Tapestry.[4–7] With this approach, peers are responsible for maintaining a part of the file index in the form of (key, value) pairs. Peers insert (key, value) pairs corresponding to the files they are willing to share. Then queries are directed to the responsible peer and the file is downloaded directly from the peer storing the file.

Typically, searches in P2P file sharing systems are keyword searches and fall into two categories: 1) direct and 2) browsing. With direct searches the peers know exactly what file they desire and only need to locate that exact file at another peer. With browsing searches, a peer may not know the name of a particular file at the time the search request is issued but wishes to browse the P2P system for files relating to the specified keywords. Both the centralized and decentralized models for P2P file sharing have scalability issues related to performing browsing keyword searches. With Napster the central server(s) needs to maintain a large amount of information about all of the files in the system. So, the peer can easily browse the system as long as the central server(s) can handle the requests. Gnutella's scalability issue lies with the flooding of the search requests which may not find the desired data due to the TTL search limit.Finally, the hash-based indexing schemes were designed for direct searches. In such schemes, when keywords are used as keys, certain keywords may be more popular than others so that a few peers become responsible for maintaining a large part of the system file index.

Searching in decentralized P2P systems is similar to the distributed match-making problem addressed in the distributed computing environment. There a client process needs to be matched to a server process providing the desired service. Algorithms described in Ref. 8 require the server process to "post" the fact that it is providing a service to other nodes in the system, and the clients can then query nodes to find the address of the desired server process. However, in such an environment, the services are uniquely named while in the P2P systems, files with the same name may contain different content and a peer may not know the exact name of the file desired. Thus we modify this match-making approach for use with decentralized P2P systems.

In this paper we propose posting of keyword information as a method to reduce search traffic and improve the success rate of keyword searches, both browsing and direct, in the Gnutella model of decentralized P2P file sharing. In such a system the peers are dynamic and heterogeneous. where peers arbitrarily form connections to other peers. Our service model is one where the peers perform browsing keyword searches in order to find content. The content itself is not searchable and keywords are used to find files with file names matching the specified keywords. Our posting protocol requires that peers in the network periodically replicate or "post" information about files they are willing to share to other peers in the system. Peers cache this post information for a specified period of time. Then when a query for a file reaches a peer it searches for the keyword(s) as part of the file name of its stored files and also searches the cached post information it is maintaining. If a match is found a response is sent back to the querying peer. Strengths of our posting protocol are:

1. Trading memory for bandwidth. By caching the post information peers can query less, reducing query traffic.
2. Reducing the search range. Replicating the keyword information allows peers to find matches using a smaller TTL.
3. The overhead is small. By using keyword information the posting protocol requires less memory when compared to caching entire files or queries and their responses.
4. Peers remain in control. The posting protocol allows the peers to decide whether or not to cache the post information.

The main weakness of posting keyword information is the reception of stale information. A querying peer may receive a post result in response to a query and not be able to contact the posting peer, the peer whose post information matched the query. However, with the Gnutella network, a querying peer may receive a query response from another peer and not be able to contact the responding peer to perform the download. Since the querying peer received the response as a direct result of the query, the chances of the responding peer leaving before the download request is received is lower but still a possibility. Another weakness of our posting protocol is that even though the cached post information matched a query, the posting peer may not have files matching the query. This will depend on how the post keywords are selected and the amount of keywords included in a post message. For example, a query for "beatles", "yesterday", and "mp3" can match a post cache entry but the files stored at the posting peer could be "beatles.mp3" and "that_was_yesterday.mp3".

The remainder of this paper is organized as follows. The next section discusses related work in the area of P2P file sharing systems. Section 3 discusses features of the unstructured decentralized P2P file sharing system, as seen in Gnutella, that motivate our posting mechanism. Section 4 provides an overview of our proposed posting protocol. Section 5 contains an evaluation of the protocol and Sect. 6 provides concluding remarks.

2. RELATED WORK

Recently there have been proposals for the implementation of a scalable P2P lookup or indexing service. The Chord[4] system proposes to solve the problem of efficient location of data items. It provides a scalable protocol for lookup in P2P systems by using consistent hashing to map a key onto a node. So by associating a key with each data item, data location can be implemented on top of Chord. The CAN[5] work also proposes the use of a hash table to provide a scalable indexing mechanism. The content-addressable network (CAN) is a distributed, Internet-scale, hash table. The basic operations performed on a CAN are insertion, lookup and deletion of (key, value) pairs. The Pastry[6] system provides an object location and routing scheme for decentralized P2P systems. Tapestry[7] provides a location and routing infrastructure for location-independent routing of messages. Like Pastry it routes based on address prefixes.

These hash-based systems were designed to address the problem of scalable indexing and not browsing keyword searches where the exact file name is not known. The Chord work states that the key for a file could be derived from the desired keywords but this requires the owner of the file to specify the keywords and insert the file using each as the key. The owner would also be responsible for updating each of these references. A user searching for the file using any subset of the owner specified keywords would be able to find the file by using each keyword as the key as well. However, due to the structured indexing, peers that are responsible for storing the popular keywords will need to maintain a potentially large set of (key, value) pairs associated with the large number of files and their copies that contain the popular keywords. So when keywords are used as keys in these structured decentralized systems, a potentially large number of keys must be redistributed amongst the remaining peers in the system when peers join and leave. This creates more of a problem when peers storing popular keys frequently join and leave the system.

Thus to overcome this searching limitation we propose the use of a "posting" or replication technique to improve the success and reduce the traffic of searches in the unstructured decentralized P2P systems. By having peers replicate keyword information to other peers the search region can be reduced without reducing the success rate of searches. We evaluate different posting policies and compare the results for searching with and without posting.

3. UNSTRUCTURED DECENTRALIZED P2P FILE SHARING SYSTEMS

We focus on the unstructured decentralized model as seen in the Gnutella network, a popular decentralized P2P file sharing system. Gnutella peers form an overlay network consisting of TCP connections between the peers. To join the Gnutella network a user must connect to a peer currently in the network. Once a connection is established the peer can communicate with other peers using Gnutella protocol[9] messages. A *ping* message is used to discover other peers on the network and contains no payload. *Pong* messages are sent in response to a ping and specify the responding peer's address and port, number of files shared and number of kilobytes shared. Peers use ping and pong messages to maintain the overlay network by forming connections with the peers discovered. Searching in the Gnutella network is done by sending a *query* message containing search keywords to neighbors. The query message is broadcast to all peers within the number of hops specified by the TTL in the message. Peers receiving a query search the files they are sharing and if a match is found a *query response* message is generated. Query Response messages travel the reverse path of the query message to which it is responding. The query response message contains information about the peer responding and the files matching the query. Each Gnutella application runs an HTTP server and file downloads are performed using a direct HTTP connection between the serving peer and the requester.

The characteristics of the Gnutella network should be considered when creating a new searching mechanism. Our posting mechanism is motivated by two interesting features of Gnutella: 1) heterogeneous peers and 2) query popularity.

3.1. Heterogeneous peers

A measurement study of P2P file sharing systems[10] details several characteristics of the peers participating in such systems. The work found that peers are heterogeneous with respect to connection speeds, latencies, lifetimes, and data shared. The downstream (to the peer) bottleneck bandwidth is on average higher than the upstream (from the peer) bottleneck bandwidth of a peer. Thus peers are usually able to download files at a faster rate than the rate at which they can serve files to others. High-latency is experienced by a large fraction of the connections due to the arbitrary construction of neighbor connections. The lifetime of most sessions is very short with a median duration of 60 minutes. The work in Ref. 10 found that the percentage of files shared by a bandwidth class was roughly the same as the percentage of peers in that bandwidth class. The work concludes that different responsibilities should be placed on different hosts depending on the heterogeneous characteristics of peers. Therefore our posting protocol must take the heterogeneity of peers into account. For example, the lifetime of the peers should be considered when determining the amount of time to cache post information. Also, connection speeds, latencies, and data shared affects a peer's ability to serve files and should considered when selecting amongst the returned query results from different peers.

3.2. Query popularity

A study of the characteristics of queries in the Gnutella network was performed in Ref. 11. Traces of queries and results routed through a modified Gnutella client were maintained. Queries were ranked in order of popularity with 1 being the most popular query. The work showed that the most popular queries in the Gnutella network (those with rank 1 to 100 in the collected traces) are equally popular, and less popular items (with rank 100 to 100,000^{+}) follow a Zipf-like distribution, where the probability of seeing a query for the i^{th} most popular query is proportional to $\frac{1}{i^{\alpha}}$. Simulations using three different caching timeouts were performed. The simulation results in Ref. 11 showed that caching queries and replies significantly reduced the traffic seen on the Gnutella network. The traffic was reduced by 3.7 times when the highest cache hit rate was observed. The results found that a larger cache interval reduce the traffic significantly but this can lead to stale information being cached. The work concludes that caching is only a short-term solution to scaling the Gnutella due to memory requirements of caching. In the simulations, the amount of memory used for caching ranged from 195kB to 4.3MB depending on the caching interval. As the Gnutella network grows, more memory will be needed in order to achieve good caching performance. The results from this work show that caching is useful but that the size of the cache limits the use of caching as a long-term solution to reducing the query traffic in the Gnutella network. Thus having our protocol post only keywords reduces the size of the cache required when compared to caching entire queries and responses.

4. POSTING PROTOCOL

In this section we describe our posting protocol. First, posting approaches are given followed by a discussion of state transitions and a description of protocol functions. As discussed in Sect. 2, the hash-based indexing systems provide structured decentralized indexing by making peers responsible for storage of part of the index and directing client queries to the peer responsible for that part of the index. With the Gnutella approach to decentralized P2P file sharing, the peers can be thought of as maintaining an index of the files they share and it is necessary for searches to be broadcast in order to locate the peer with the desired file. Finding the desired file in the decentralized P2P file sharing system is similar to the distributed match-making problem addressed by Mullender and Vitányi.[8]

The distributed match-making problem involves a distributed computing environment where a client process needs to be matched with the address of a server process providing the service the client process desires. The algorithms in Ref. 8 require the server process to post the fact that it is providing a service to a collection of nodes in the network, P. These nodes cache the information for future use. Then, to find a server process providing the desired service, a client process will send a request for the address of the server process to another set of nodes in the network, Q. Nodes in both of these sets, $P \cap Q$, will respond to the client process by providing the address of the server process.

The current Gnutella network is equivalent to a network where the servers post to themselves and clients query everyone. However, with the Gnutella network peers are searching for files and the addresses of peers providing the files. The exact file name may not be known so that a query response containing a long list of files matching the keywords may not contain the file the client desired. Also, files with the same name may contain different content. In Ref. 8, the services are uniquely named so there is no ambiguity about what service the client process desires. Server processes providing the service will return the same result to the client process regardless of which server process is contacted. Thus the match-making method in Ref. 8 will need to be modified for use in the Gnutella network.

By posting information about files shared to other peers in the network the success rate of searches in the Gnutella network can be improved. Queries in the Gnutella are scoped by the TTL specified and do not reach all the peers in the network. Posting information to peers may allow queries to find the post information in a situation where the poster is beyond the TTL scope of the searcher. Figure 2 shows a set of peers in a region of the Gnutella network. Peer A wishes to find the Beatles song titled Yesterday. The peer sends a query with a TTL of 3 and the keywords “beatles” and “yesterday”. Peers J and K are both sharing a “Beatles - Yesterday.mp3” file and are the only peers sharing files matching the specified

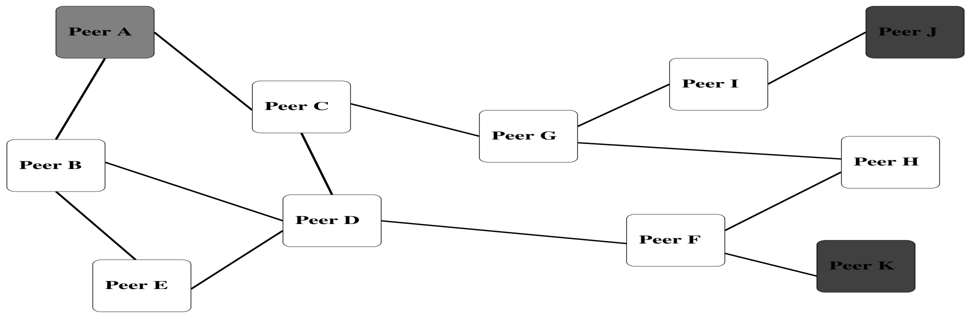


Figure 2. Posting Example

keywords. Peers B-I receive the query and check their list of shared files. In this situation peer A's query fails. Now if peers J and K were to post the keywords "beatles" and "yesterday" and any one of the other peers (A-I) cached the information, A's query would receive a response. Then A could contact J or K directly to determine if they are indeed storing the desired file. Note that if A were to receive and cache the post from J or K, it could have contacted J and/or K directly before scoped flooding the query to neighbors.

The above example illustrates how posting can improve search success. The improvement in search success is likely to lead to a reduction in search traffic as fewer searches are re-issued due to failure finding the desired files. The search TTL may also be reduced as matches can be found in a smaller region as a result of cached posts. Our posting protocol will work by having peers periodically post keyword information about the files they are willing to share. We chose keyword information instead of the filenames to reduce the size of the post information in messages and the cache size at peers. This post information can be appended to the Gnutella query protocol messages as will be discussed later. The post information will need to have a lifetime associated with it since peers are not likely to be always on the Gnutella network. Unlike the algorithms in Ref. 8, the peers receiving post information will decide whether or not to cache it.

4.1. Posting approaches

The next sections describe approaches that can be used to implement our proposed posting protocol. The performance of these approaches will be evaluated in Sect. 5.

4.1.1. Post information

As mentioned above the post information consists of keywords that make up the files a peer shares. However keyword information about all of the files being shared by a peer need not be posted. Some possibilities for the type of keyword information to post are listed below.

1. **Frequently Served** - Keywords from files frequently sent in response to download requests.
2. **Recently Served** - Keywords from files recently sent in response to download requests.
3. **Served** - Keywords from files sent in response to download requests.
4. **Frequently Responded** - Keywords from files frequently sent in response to query requests.
5. **Recently Responded** - Keywords from files recently sent in response to query requests.
6. **Responded** - Keywords from files sent in response to query requests.
7. **Frequently Queried** - Frequently seen keywords from queries to which the peer responded.
8. **Recently Queried** - Recently seen keywords from queries to which the peer responded.
9. **Queried** - Keywords from queries to which the peer responded.

Depending on the type of information to be posted, a posting peer will need to maintain statistics on the files or query words that have been served or responded. The post message must contain enough data for a peer receiving it to be able to contact the posting peer. Thus, in addition to the keywords being posted, the post information contains fields similar to those of the Gnutella query response (See Ref. 9 for Gnutella protocol details). Specifically, the post information also includes the posting peer's port number, IP address and connection speed. The post also contains a timestamp value which marks the maximum amount of time the post is to be cached and a length indicating the size of the keywords included. As discussed in Sect. 3.1 the connection speed of a peer should be considered when deciding which peer to contact. Providing a timestamp to indicate the length of time to cache the post reduces the amount of stale post information being cached.

4.1.2. Post messages

Instead of adding new post messages to the Gnutella protocol for the sending of the post information we decided to include the post as part of Gnutella protocol messages. Thus the post information could be piggybacked along with the Gnutella queries, query responses or the file download.

1. **Queries** - Including the post information with queries will distribute the information to more peers since these messages are currently flooded in the Gnutella network. However, there is no guarantee that the peers receiving this information will desire to cache the post.
2. **Query Responses** - Query responses travel the reverse path towards the querier. By posting along with query responses fewer peers are reached than by posting with queries. A peer receiving the query response may be more interested in using the post information since it has already initiated a query that contained keywords associated with files stored by the poster.
3. **File Download** - By adding post information to the file download the poster provides the downloader with information about other files the poster is sharing. This is similar to the online store advertisements “You’ve searched for this but you might also want these items”.

We decided to piggyback the post information with Gnutella queries to allow the post information to reach more peers. For peers who do not query but are sharing files, empty queries with the appended post information will be sent. The posting rate determines how often the posting keywords are extracted from the post information being maintained by the peer. Note that peers that are not sharing any files will not have any information to post.

4.1.3. Post caching

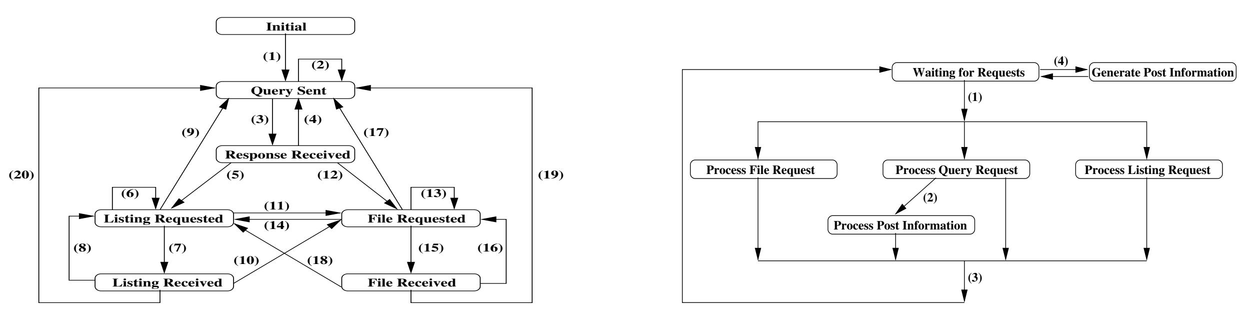
Peers will need to determine the size of their cache as well as what keyword information should be cached. Possibilities for deciding what types of post information to cache include:

1. **Like** - Cache the posts with keywords like those currently being posted by the receiving peer.
2. **Diff** - Cache the posts with keywords that are different from those currently being posted by the receiving peer.
3. **Prob** - Probabilistically determine whether or not to cache the posts. For example, the receiving peer can randomly choose a number uniformly distributed between 0 and 1 and if it is less than acceptance probability it will cache the post.
4. **All** - Cache all the posts received without regard to the type of keywords they contain.

Policies we consider for cache replacement are: Least Recently Used (LRU), Least Frequently Used (LFU) and First-in First-out (FIFO).

4.2. State transition

We describe the state transitions for the posting protocol in this section. Although peers can act as both clients and servers, we will discuss the state transitions for the basic protocol in terms of client and server processes. The client process is only concerned with searching the P2P file sharing system while the server process responds to requests and generates posts. Figure 3(a) shows the client process state transition diagram. The client starts in an *Initial* state and moves to the *Query*



(a) Client Process State Transition

(b) Server Process State Transition

Figure 3. State Transitions

Sent state once a query has been initiated (1). The client will wait for a response and if a response is received (3) will move to the *Response Received* state. If no response is received the client may issue another query (2) and remain in the *Query Sent* state. From the *Response Received* state the client can either: issue a new query (4), request a listing of files (5) or request a particular file for download (12).

When the client receives a response indicating that the query keywords matched posted keywords, only the cached post information is returned. The names of files stored at the posting peer are not provided. Thus, the client will need to request a listing of files matching the keywords directly from the posting peer. If a listing is received (7) the client transitions to the *Listing Received* state. Otherwise, the client may issue a: listing request (6), query request (9), or file download request from the non-post query responses (11). From the *Listing Received* state the client can either: request another listing (8), request a file from the listing or query responses (10), or issue another query (20).

The client will transition from the *Response Received* state to the *File Requested* state when a file request is issued (12). If the file requested is received (15) the client transitions to the *File Received* state. Otherwise the client may issue a: file download request (13), query (17) or listing request from the post query responses (14). From the *File Received* state the client can either: request another file (16), request a listing (18), or issue another query (19). The client process exits whenever the program is terminated by the user which can occur during any state.

Figure 3(b) displays the server process state transition diagram. Here the states are more general and will be described in more detail in the following section. The server process begins in the *Waiting for Requests* state. When a request is received (1) the server transitions to either the: *Process File Request*, *Process Query Request* or *Process Listing Request* states. Once the request has been processed (3) the server returns to the *Waiting for Requests* state. Periodic timers are used to determine when to update post information. When the timer expires (4) the server process transitions from the *Waiting for Requests* state to the *Generate Post Information* state. Depending on the timing of the last post a new post message may also be generated. Our posting information is piggybacked along with queries sent by a peer. From the *Process Query Request* state if post information is present the server transitions (2) to the *Process Post Information* state.

4.3. Protocol functions

The posting protocol requires the addition of seven new functions for P2P file searching. These functions are *generate-post*, *process-post*, *send-post*, *request-listing*, *process-listing*, *check-cache* and *update-post-info*. Each of these seven functions are described below.

1. **generate-post:** A peer performs the generate-post function while in the *Generate Post Information* state.
   (a) The post timer expires (4) indicating that it is time to update the post information.
   (b) If the last post time was more than *post period* time ago, the *send-post* function is called and a post is sent containing the current post information. This is basically an empty query with the post information attached.
   (c) The post information is updated by extracting keywords from the appropriate file names or cached queries. Then the post statistics information being maintained is reset.
2. **process-post:** A peer performs the process-post function while in the *Process Post Information* state.
   (a) The peer transitions to the *Process Post Information* state after receiving a query containing post information (2).
   (b) The peer checks the post information to see if it desires to cache it or not.
   (c) If the peer decides to cache the information, the post cache is updated and a previously cached entry may need to be removed.
3. **send-post:** A peer performs the send-post function while in the *Generate Post Information* state as described above or when transitioning from the *Initial State* to the *Query Sent* state.
   (a) When a query is to be sent or the post information is to be updated, the peer checks the time of the last post.
   (b) If the last post time is more than *post period* time ago, a post is sent along with the query or as an empty query. Then the time of the last post is set to the current time.
4. **request-listing:** The request-listing function is called when the peer desires file names (5,6,8,18) from a posting peer whose post keywords matched the query. A listing request is similar to a query but is sent directly to the posting peer and not forwarded over the Gnutella network.
   (a) The query keywords are placed in the listing request.
   (b) The listing request is sent as an HTTP request directly to the posting peer. A listing request has the following format:
```
GET /listing/<Query Keywords>/ HTTP/1.0\r\n
Connection: Keep-Alive\r\n
Range: bytes=0-\r\n
```

```
User-Agent: Gnutella\r\n
\r\n
```

5. **process-listing:** The process-listing function is called from the *Process Listing Request* state.
   (a) The peer searches the files it is sharing for any matches to the keywords given in the listing request.
   (b) If a match is found, the file name is included in the listing response. All file matches are compiled into a successful listing response. The format response is an HTTP 200 OK response followed by the set of matching files with the given structure:
   `<File Index> <File Size> <File Name>`
   (c) If no match is found, a HTTP 404 error response is returned.
6. **check-cache:** When the peer is in the *Process Query Request* state the check-cache function is called.
   (a) The query keywords are compared to the keywords contained in each of the post cache entries.
   (b) If there is a match, that post information is added to the response generated.
   (c) Otherwise, if no matches are found in the post cache, the query is processed as usual by checking the peer's files for a match.
7. **update-post-info:** When the peer is in the *Process File Request* or *Process Query Request*, it may be necessary to update the file or query keyword statistics being maintained if a file is returned or a query response is generated. Examples of the post statistical information are a count of times the files has been served, the last time the file was sent as a response or the count of times a query keyword matched the peer's stored files. (See Sect. 4.1.1 for the types of post information that can be sent.)

5. EVALUATION

We simulated the posting protocol using an event-driven simulation and query/response trace files. We obtained query/response trace files from colleagues.[11, 12] The trace files we used for our simulation were collected from five different locations and details on their collection can be found in Ref. 12. We sampled the same amount of time from each of the five trace files used with our simulation.During the simulation, file names are extracted from trace responses and assigned to peers. Trace queries are randomly assigned to active peers. Using the results from[10] we mark 22% as low bandwidth peers, 48% as medium bandwidth peers and 30% as high bandwidth peers. The bandwidth type is then used to determine the minimum number of connections the peer tries to maintain. The post size was set to 64 bytes, including the bytes needed for the post header. The search TTL is set to 7 which is the default for most current Gnutella clients. We chose such a small post size to minimize the overhead of including the post with the query messages and to provide a lower-bound examination of the benefits of posting. Using a larger post size would allow for more keywords to be included in the post and potentially more matches to queries. Our evaluation metrics of interest are:

1. *Average Number of Results per Query* - The average number of unique filename and cached post entries received in response to a query.
2. *Average Number of Hops to the First Post Match* - When a match was found in a post cache, the average number of hops the search request traveled before finding the first post match.

The following section describes the results when we simulated the different posting approaches described in Sect. 4.1. Section 5.2 provides a comparison of our posting protocol to the Gnutella protocol without posting. A 90% confidence interval computed over 5 runs is shown in each of the performance figures.

5.1. Posting protocol policies

Our posting protocol has various policies that can be varied. These posting approaches were described in Sect. 4.1. In this section we evaluate the different approaches to determine which to use when comparing the posting protocol to the Gnutella protocol without posting. First we vary the type of information to send in the post messages. Then we evaluate the various caching and cache replacement policies. Finally, we experiment with different post cache sizes.

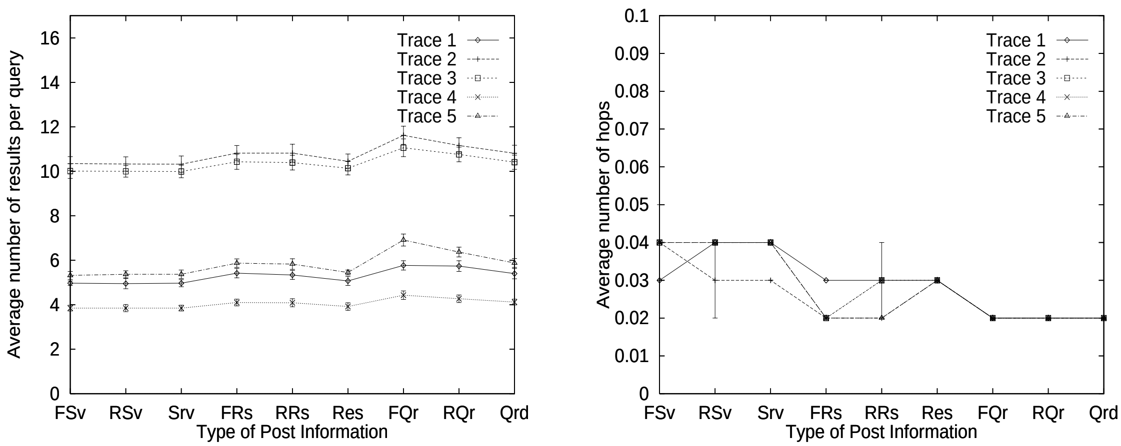
5.1.1. Post information

As listed in Sect. 4.1.1, there are many ways to determine which keywords are sent in the post. We simulate each of the methods described in that section to determine what type of post information should be sent to improve the amount of search results. For these simulations we set the cache size to 16384 bytes and use the **All** caching policy. The cache

Table 1. Post Information Types

FSv - Frequently Served	FRs - Frequently Responded	FQr - Frequently Queried
RSv - Recently Served	RRs - Recently Responded	RQr - Recently Queried
Srv - Served	Res - Responded	Qrd - Queried

replacement policy is set to **FIFO** but with such a large cache size it should not effect the post caching. Table 1 provides a legend for the x-axis labels for the different types of post information. Figure 4(a) shows the average number of results



(a) Avg. Results per Query

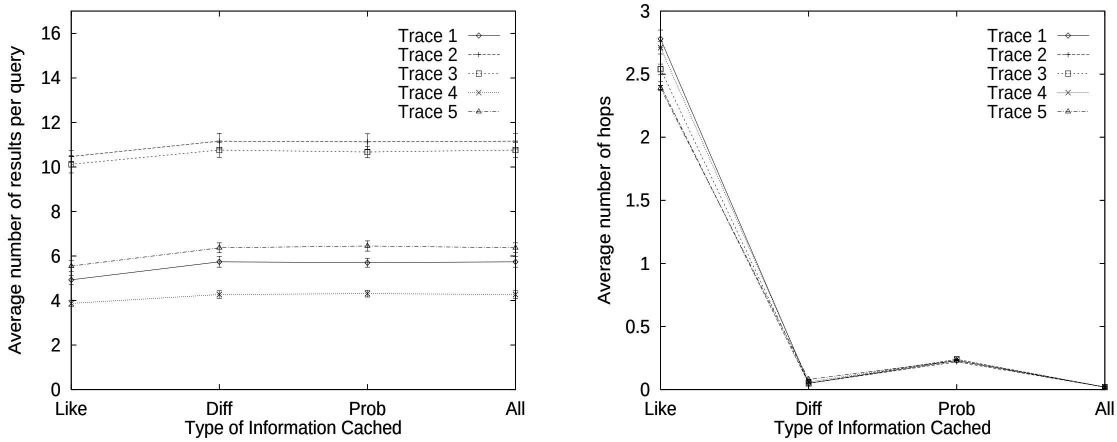
(b) Avg. Hops to First Post Match

Figure 4. Post Information

received per query. The **Served** post information types returned the lowest average number of results per query, with **Responded** performing better and **Queried** performing the best. Posting keywords from files a peer has served to others results in a less proactive approach because a peer will only have keywords to post after it has served a file. Sending keywords from the files a peer has responded to improves the performance but can result in extracting keywords that do not need to be post. Thus when posting keywords from queries to which the peer responded the keywords to post do not need to be extracted from the file names. Posting **Frequently Queried** keywords produced the highest average number of results per query with **Recently Queried** keywords providing a slightly lower average. However, **Recently Queried** keywords posting had a higher percentage of queries receiving post results. The type of post information had a small effect on the average number of hops to the first post match as shown in Fig. 4(b). Note that the number of hops is close to zero because all of the peers are posting and caching all of the posts received, thus when a post match is found it is usually found in the searching peer's cache. From these results we chose **Recently Queried** keywords as the type of post information to be used in the remaining simulations.

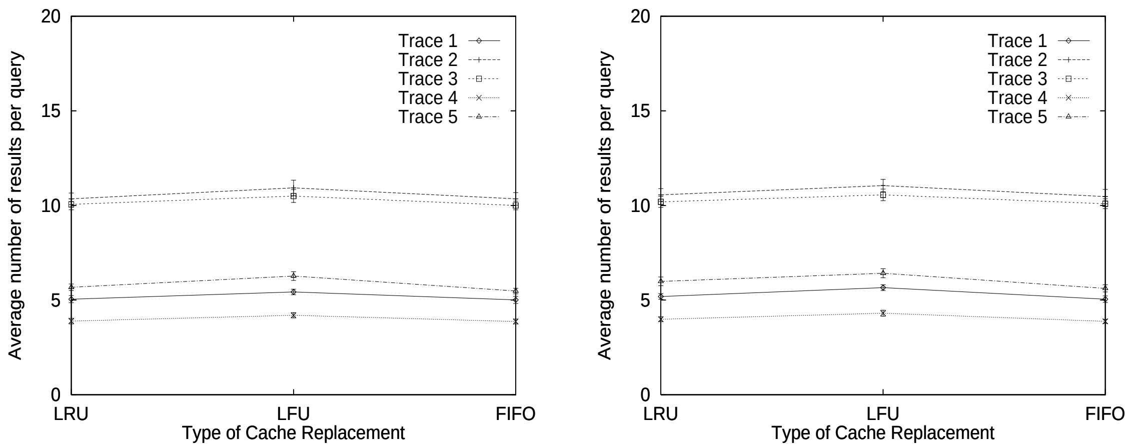
5.1.2. Cache policy

In Sect. 4.1.3 four policies for caching post information are listed. Figure 5 shows the results of using each of the caching policies. Again we set the cache size to 16384 bytes and use **FIFO** as the cache replacement policy. Figure 5(a) shows that the **Like** caching policy produced a slightly lower average number of results per query than the other three caching policies. For the **Diff** and **All** caching policies, the average number of hops to the first post match was close to zero. So for those two policies, the first post match was most often found within the searching peer's post cache. On average the first file match was found within 2.5 hops which means that the **Like** method usually found a post match once a file match was also made. Thus when **Like** posts are cached the query must travel farther to find a post match. Since the **Diff** and **All** caching policies produced quite similar results, we have chosen to use the **Diff** caching policy in the remainder of the simulations. We chose **Diff** over all to limit the effects of the cache size on the posting protocol. With the **All** method some entries may be cached that are similar to what the peer is already posting and, as the **Like** method results indicate, these entries are less likely to provide an increase in query results.



(a) Avg. Results per Query (b) Avg. Hops to First Post Match

Figure 5. Cache Policy Effects



(a) 640 byte Cache Size (b) 1280 byte Cache Size

Figure 6. Cache Replacement Policy Effects Avg. Results

5.1.3. Cache replacement policy

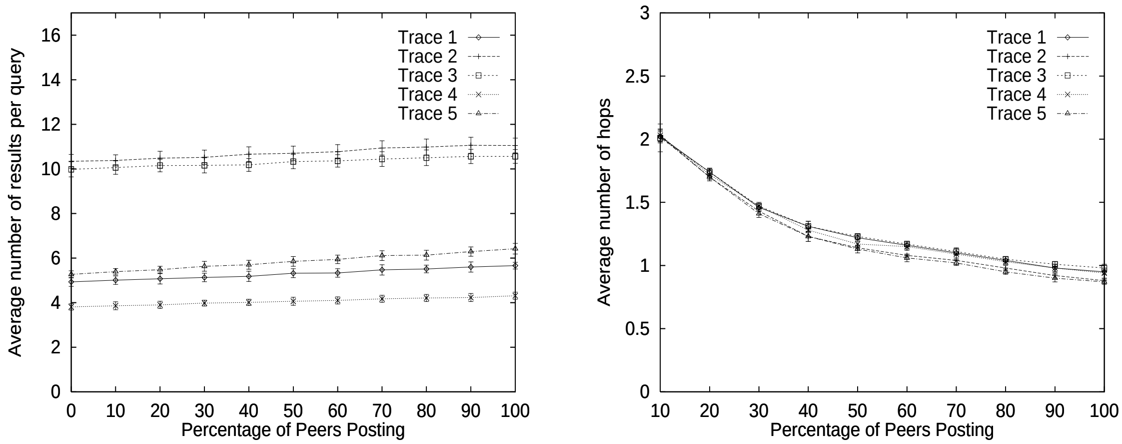
We simulate three cache replacement policies as were listed in Sect. 4.1.3. Figure 6 shows the effects of varying the cache replacement policy on the average number of results per query. Since the cache size effects the performance of the cache replacement policy, we vary the cache to be 640 or 1280 bytes and the results are shown in Fig. 6(a) and Fig. 6(b) respectively. We chose such small caches size, compared to the large files that are transferred within the file sharing systems, to illustrate the lower-bound benefits of posting. As the work in Ref. 11 showed, caching queries and results improved the search performance of the Gnutella network and reduced the search traffic but the cache size was a limiting factor. By posting keyword information the cache size required by our posting protocol is significantly smaller. From Fig. 6 we see that regardless of the cache size **LFU** provides a higher average number of results per query. Of the three cache replacement policies, the **FIFO** method provides the lowest percentage of queries receiving post results regardless of the cache size. The average number of hops to the first post match was about one for all three methods with **FIFO** requiring the fewest hops on average. Thus we chose **LFU** as the cache replacement policy. We also selected 1280 bytes to be the size of the post cache.

5.2. Comparison to the Gnutella protocol

In this section we compare the performance of our posting protocol to that of the Gnutella protocol without posting. We post **Recently Queried** keyword information and use the **Diff** caching policy. The cache replacement policy is set to **LFU** and the cache size is fixed at 1280 bytes. We evaluate the effects of incremental deployment of our posting protocol by varying the percentage of posters from 10% to 100%. We also show results for having 0% of the peers posting which is equivalent to the Gnutella protocol without posting. The next section shows the results of varying the percentage of posters and the following section shows how changing the search TTL effects the performance of the posting protocol.

5.2.1. Percentage of posters

For these simulations, the search TTL is set to 7 for all peers but only the peers marked as posters will send the periodic post messages. From Fig. 7(a) we see that posting increases the average number of results per query when compared to



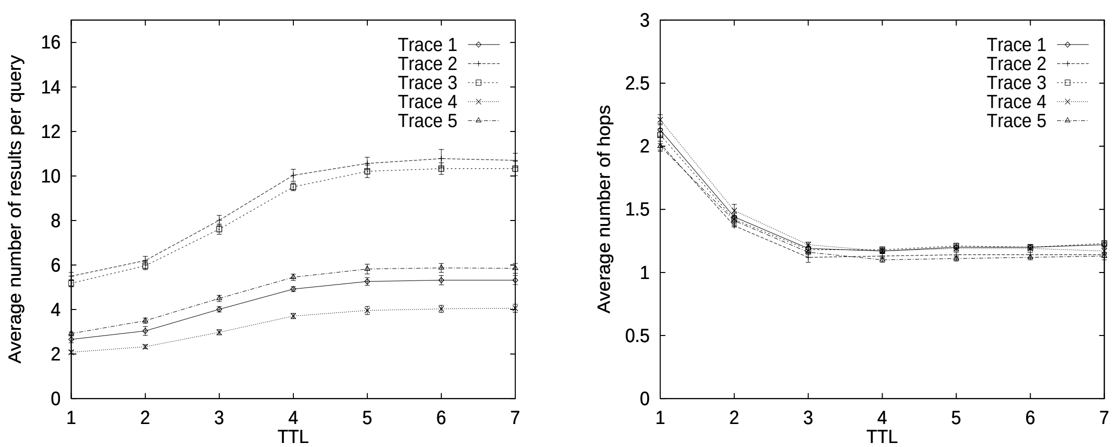
(a) Avg. Results per Query (b) Avg. Hops to First Post Match

Figure 7. Percentage of Posters Effects

having 0 percent of the peers posting. As the percentage of posters increases so does the average number of results per query. Figure 7(b) also shows that increasing the number of peers posting also reduces the hops need to find the first post match. With just 10 percent of the peers posting the first match to a post cache entry can be found within 2 hops while it takes on average 2.5 hops to find the first file match.

5.2.2. Search TTL

The results in the previous sections used a search TTL of 7 which is the default for most Gnutella clients. One benefit of posting keyword information is that matches to the query can be found within fewer search hops. We vary the search TTL from 1 to 7 hops to determine the effects on the search performance. We examine the results when the percentage of posters is set to 50 percent. For the peers that are not posting the search TTL is set to 7. Figure 8(a) shows reducing the search TTL



(a) Avg. Results per Query (b) Avg. Hops to First Post Match

Figure 8. 50% posters - Search TTL Effects

to 5 hops will produce an equivalent average number of results per query as using the default 7 hops. In Fig. 8(b) we see that with a search TTL of 1 for the posting peers the first post match is found on average about 2 hops away. This means that with such a small TTL, the majority of the post matches are being received by non-posting peers whose searches travel farther. Once the search TTL reaches 3 hops the first post match is found about one hop away which is the same as when a search TTL of 7 is used. Thus, even with halve of the peers posting the search TTL of the posting peers can be reduced from 7 to 5. The results for 100% of the peers posting also shows that a search TTL of 5 hops produces the same results as using a TTL of 7. Given the large number and frequency of queries from these trace files, reducing the TTL by 2 hops will greatly reduce the amount of query traffic.

6. CONCLUDING REMARKS

We have described a posting protocol to improve search success for decentralized P2P file sharing system such as Gnutella. Keywords are posted to other peers as part of query requests and peers cache the post information for use in responding to future queries. Our motivation for the protocol is to improve the search success of browsing keyword searches where the exact name of the desired content is not known. We have performed simulations to evaluate the various posting policies and compare the protocol performance to that of the Gnutella protocol without posting. Of the eleven types of keyword information to include in the post messages sending **Recently Queried** keywords provided the highest percentage of queries receiving post results. The **Diff** and **All** caching policy performed the best out of the four caching policies and **LFU** performed the best of the cache replacement policies.

We selected a 64 byte post size and a 1280 byte post cache to provide a lower bound evaluation of the benefits of posting. Even with such small sizes the results indicate that posting does increase the amount of query results when compared to not posting. Posting also allows the search TTL to be reduced as post matches are found within just a few hops or even within the querying peer's post cache. The overhead of the posting protocol is small since the post and post cache size can be on the order of bytes and not kilobytes or megabytes. Also, the posting protocol listing requests are sent between individual peers and do not travel over the P2P network. With just a small percentage of peers posting, the search TTL can be reduced without effecting the search success and only a slight reduction in the amount of results per query.

The posting protocol can be used to improve searching in decentralized P2P file sharing systems. Future research includes determining the search TTL, post size, and post cache size so as to limit the protocol overhead and stale cache information while increasing search success and reducing the search traffic.

REFERENCES

1. "Napster homepage." http://www.napster.com.
2. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proc. of ACM Sigmetrics*, June 2002.
3. "Gnutella homepage." http://gnutella.wego.com.
4. 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. of ACM SIGCOMM*, (San Diego, CA), Aug. 2001.
5. S. Ratnasamy, P. Francis, M. Handley, and R. Karp, "A scalable content-addressable network," in *Proc. of ACM SIGCOMM*, (San Diego, CA), Aug. 2001.
6. A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Proc. of IFIP/ACM Middleware*, (Heidelberg, Germany), Nov. 2001.
7. B. Zhao, J. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Technical Report UCB/CSD-01-1141, University of California at Berkeley, Apr. 2001.
8. S. J. Mullender and P. M. B. Vitányi, "Distributed match-making for processes in computer networks," *Operating Systems Review (reprint from PODC '85)* **20**, pp. 54–64, Apr. 1986.
9. "Gnutella protocol specification v0.4." http://www9.lime-wire.com/developer/gnutella_protocol_0.4.pdf.
10. S. Saroiu, P. K. Gummadi, and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Proc. of SPIE Multimedia Computing and Networking*, (San Jose, CA), Jan. 2002.
11. K. Sripanidkulchai, "The popularity of gnutella queries and its implications on scalability." http://www-2.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html, Feb. 2001.
12. B. Krishnamurthy, J. Wang, and Y. Xie, "Early measurements of a cluster-based architecture for P2P systems," in *ACM SIGCOMM Internet Measurement Workshop*, (San Francisco, CA), Nov. 2001.