GALANX: An Efficient Peer-to-Peer Search Engine System

Yuan Wang Leonidas Galanis David J. DeWitt
Computer Sciences Department, University of Wisconsin - Madison
1210 W. Dayton St.
Madison, WI 53706
USA
{yuanwang, lgalanis, dewitt}@cs.wisc.edu

ABSTRACT

Although peer-to-peer systems have recently emerged as a popular way to share data on the Internet, most applications still rely on centralized services, such as Google, to find relevant data. This paper presents the design and implementation of GALANX, a peer-to-peer search engine that was implemented using the Apache HTTP server and BerkeleyDB. GALANX directs user queries to relevant nodes by consulting a local peer index that is maintained on each node. The use of peer indices to direct searches was experimentally evaluated using a 100 processor cluster. A number of alternative query routing strategies were also implemented and evaluated in the GALANX framework. Experimental results clearly demonstrate that the use of peer indices can significantly improve performance.

1. INTRODUCTION

Since Napster™ [17] first made it possible for millions of people to exchange MP3 files, peer-to-peer (P2P) file sharing mechanisms have gained enormous popularity. P2P systems have rapidly emerged as a popular choice to share large amounts of data over the Internet. Compared to the traditional client-server systems, P2P systems have demonstrated some fundamental advantages:

1) ***Fault tolerance***. P2P systems are structurally more robust than centralized systems, in terms of resilience to both node and network failures.

2) ***Low cost***. P2P systems distribute hardware and network bandwidth costs among all the participants in the system.

3) ***Easy adaptation***. P2P systems can dynamically adapt when nodes join and leave, or even fail. Nodes autonomously connect to each other, with or without control from a central authority.

Because data objects are stored on the nodes of a P2P system and not in a centralized server, an important task is, given a request from a user, to be able to efficiently locate the desired objects. Currently, several different mechanisms are used to route user requests to the appropriate data source nodes.

1) **Central directory services**. The locations of all shared data in the network are registered on a central server [17]. In this case, a node must access the central directory before contacting the data source to process a user request. Obviously, this strategy requires central infrastructure and is not in the spirit of pure P2P systems. It is vulnerable to failures and is not scalable.

2) **Flooding-based routing**. This approach [4][6] does not employ any centralized servers or any data location directories (although individual nodes can maintain a cache of locations of previously requested data [4]). User requests are broadcast to neighbor nodes in a non-selective way such as breadth-first or depth-first. This method requires minimal system state at the expense of network bandwidth and CPU resources, causing scalability and efficiency problems.

3) **Distributed Hash Tables (DHT)**. In this approach every shared data object has an associated hash value and every node in the system is responsible for storing the data objects whose hash values fall into a certain range [3][19][24][29]. When a user asks for a file, the same hash function will be applied on the request to determine which peer holds the data.

In order to search in a large number of Internet sites people currently almost always employ a centralized search engine, such as Google, Yahoo, AltaVista, etc. Such services periodically crawl the Internet in order to construct indices against which search queries are processed. In a large and dynamic peer-to-peer environment, it is, essentially impossible to rely on such centralized search engines to get recent results because crawlers are not capable of providing up-to-date information. In addition, it is very difficult, if not impossible, for such systems to retrieve data that is stored in a database system and presented to the user with a dynamically created html page. Even if they can access the data [20], they may not be able to process the same queries as the data source web sites do.

This paper presents GALANX, a peer-to-peer search engine system that can efficiently route user queries to the appropriate information source nodes. The key features of

this system include:

- **Potential Support for complicated queries.** Through its design and implementation, the GALANX system does not set any limits on the format of the data objects and queries it can support. Each participating node publishes an index about what data it shares and, hence, what queries it can answer. GALANX helps peers to distribute and exchange the indices for routing queries. We conduct multi-term full text search in the experiments to illustrate this feature.

- **No requirements on data relocation**. Most DHT-based P2P systems employ a deterministic data location function that forces a shared data object to be stored on a particular node. Those P2P systems that rely on flooding to route queries require data to be cached for future accesses in order to obtain reasonable performance. The GALANX system does not move or duplicate any shared data objects for the following reasons:

1. Different peers may provide different query interfaces so that a peer may not be able to answer queries on another peer's data.
2. A peer's data may reside in a database system, which may be difficult to export.
3. Even though a peer may be willing to share its information (by processing user queries) it does not mean that it will be willing, or even able, to share the underlying source data. In such cases, we may have to respect the data ownership.

On the other hand, if a data cache is needed in order to increase system availability or to balance the load among nodes, the GALANX system just treats the cached data as newly inserted data.

- **Peers have great freedom to decide metadata storage**. Many peer-to-peer data sharing systems construct data lookup indices to help route user queries. DHT-based strategies force indices to be distributed to every node by applying a hash function. In GALANX, every node can cache part of, or even entire indices published by other nodes. Any node can make its own decision about what pieces of indices it wants to cache based on its storage capacity and processing capability. As will be demonstrated in Section 5, the ability to cache portions of another node's indices can significantly improve performance.

- **Efficient query routing**. The GALANX system provides more efficient query routing solutions than mechanisms that use either flooding or DHTs. Our preliminary experiments show that GALANX's indexing strategies significantly reduce the communication traffic in the system by eliminating intermediate routing hops.

The remainder of the paper is organized as follows. Related work is described in Section 2. Section 3 formulates the peer-to-peer search engine problem and discusses both advantages and disadvantages of current query routing techniques. Section 4 presents the details of the design and implementation of the GALANX system. Section 5 evaluates GALANX against other routing strategies and gives some preliminary performance results. Section 6 summarizes our conclusions and discusses future research directions.

2. RELATED WORK

Although peer-to-peer computing is not exactly a new idea, the design and implementation of P2P systems has only recently become an active research area. Napster® [17] was the major force that propelled the popularity of P2P systems. Ironically, its request routing approach is not peer-to-peer. When a user computer joins the system, it first registers its shared data on the Napster central directory server, which records the locations of all shared data objects in the system, along with the user's IP address and network connection speed. An MP3 file request is first sent to the server where it is matched against the file directory. The query results, including a list of users who have the file, are sent back to the user, so the user computer can open a direct connection with the peer. Notice that the central server itself does not share or cache files from other peers. Rather, it just maintains the index of shared files.

Other P2P media sharing systems, such as KazaA® [15] and Morpheus® [16], employ a hybrid server approach in which a central server registers the users in the system and facilitates the peer discovery process. When a peer joins the system, the server provides it with the address of one or more "super-nodes" to which the peer then connects for service. Note a "super-node" is a peer in the system that has been selected to become a super-node because it has sufficient bandwidth and processing power. A super-node indexes the files shared by the peers that connect to it and proxies search requests on behalf of those peers. A file request is sent to a super node, which may, in turn, contact other super nodes to resolve the location of a file. Once a peer receives its list of super-nodes from the central server, little further communication with the central server is required.

Both the central directory and the hybrid approaches can be categorized as employing centralized indexing and routing in that all users and shared data must be registered in the network. Although it locates files quickly, it is not scalable and it is vulnerable to point failures and censorship.

In Gnutella [6], every node connects to a number of other nodes when it joins the system. When a node receives a

user request, it forwards the request to all of its neighbors if it doesn't have the requested file. Each request message in Gnutella contains a *Descriptor Header* that carries a *Time-to-Live* (TTL) field to specify the maximum number of hops the request can be forwarded.

Similarly, in the Freenet system [4], to search for a file, a peer sends a request message specifying the key and a hop-to-live value. However, Freenet is more restrained in the traffic generated than Gnutella. When a Freenet client receives a request it cannot satisfy, it forwards the request to a single neighbor. If the client receives a failure notice because no further systems are known down the line, or if the client fails to get a response because the time-to-live timed out, it tries another one of its neighbors. If the requested data is ultimately found and returned, the node on the pathway will pass the data back to the upstream requestor, caching the file in its own database for future requests.

Both Gnutella and Freenet are decentralized and symmetric but neither can guarantee reliable and efficient content location.

In contrast to the flooding-based routing approaches, several research systems such as CAN [19], Chord [24], Pastry [3], and Tapestry [13][29], proposed independently, construct a distributed hash table (DHT) over the peer network in an effort to provide efficient query routing. In a DHT-based system, every shared file is associated with a key, either its name or a system id. All keys are hashed to a common key-space. Every peer is responsible for a portion of the key-space and stores the files whose keys fall into that key-space. Thus, every file request can be forwarded to the specific peer that corresponds to the file's key.

DHT-based data sharing systems route user requests to data source nodes more quickly because the destination of any request can be determined by the submitting node. However, DHT-based systems have their own limitations. Ratnasamy et al. [21] proposed a list of open questions to be studied in DHT systems. Harren et al. [11] discuss the possible architectures that can run complex queries in a DHT-based peer-to-peer database network. Huebsch et al. [12] present the PIER system that intends to move traditional database query processing utilities into a highly distributed environment. The PIER system processes SQL-like queries utilizing a DHT-based routing mechanism. Galanis et al. evaluate several different routing strategies and indices in a large peer-to-peer environment [9], and present a peer-to-peer XML data sharing system that employs the Chord system to direct XPath structure lookups [10].

Daswani et al. [2] study existing peer-to-peer systems and propose some future research directions. Gribble et al. [5] discuss the data placement issue in peer-to-peer systems. Yang and Garcia-Molina [28] propose *iterative deepening*, *directed breadth-first search*, and *local indices* techniques to improve query routing performance in a Gnutella-like P2P network. Crespo and Garcia-Molina [1] also introduce three types of routing indices to enhance routing performance comparing to the flooding-based approaches.

Most recently, Tsoumakos and Roussopoulos [27] analyze and simulate several typical search techniques. They observe the trade-off between search efficiency and update cost of flood-based schemes and *informed* (index-based) methods. Suel et al. [25] propose ODISSEA, a distributed search engine architecture, that constructs and distributes a global inverted index in a local P2P network and executes search queries within the network.

The JXTA project [14] provides an open source, language neutral, system independent platform for implementing peer-to-peer applications. It provides services such as indexing, searching and sharing of content over the basic functions of peer groups, peer monitoring and security.

3. ROUTING QUERIES IN PEER-TO-PEER SEARCH ENGINE SYSTEMS

Designing and implementing a peer-to-peer system is a challenging task and many open problems still remain [2]. The GALANX system focuses on one of the most interesting tasks of P2P systems, the query routing problem.

This section motivates the need of peer-to-peer search engines and formulates the query routing problem. Several existing approaches will be described and their pros and cons discussed.

3.1. Problem Overview

There are currently hundreds of thousands of web sites that update their content very frequently, as often as every day or even every second. Google™, on the other hand, only crawls the Internet (partially) about once every two weeks [22]. This makes searching for up-to-date information through traditional centralized search engines virtually impossible. Although Google News provides an instant news searching service by monitoring about 5,000 news websites, it still covers only a very small portion of the Web. No single search engine can monitor the vast number of dynamic websites that conduct business such as auctions, stock markets, shopping, etc.[1]

Furthermore, many large websites rely on the use of a relational database system and provide only dynamically con-

[1] At the time this paper is being written, Grub™ [8] is experimenting its distributed crawler system that aims at monitoring the Web updates by running a crawler on every participating node. This method still faces the next problem.

structed html pages. On such sites, users typically search for an item by typing in a query, and not by following a series of static links. In this case, it is hard for web crawlers to access the information stored in the databases, not to mention that such crawler-based search engines have to provide different query interfaces for different web sites.

Peer-to-peer search engines seem to be a promising solution to these problems. Such a scenario requires that every information provider is a participating node, sharing its data and updates with the rest of the network. Since information is *pushed* by its source nodes instead of being *pulled* by a centralized server, the need for crawlers is at least diminished, if not eliminated. Pushing also insures that information stored in backend database systems will also be accessible. In a P2P system, failure of one or more nodes will also have much a lower impact on the entire network.

Although peer-to-peer search engines appear to be the right solution, there are a number of open problems that prevent P2P search engines from competing with successful centralized search engines in the real Internet. Problems [2] such as what topologies peers should follow when constructing a P2P network in order to achieve efficiency while preserving autonomy, how to provide the necessary quality of service (QoS) to satisfy different kinds of users' requests, and how to verify the authenticity of data objects returned from a P2P system are certainly beyond what a single paper could possibly address. In this paper, we focus on a critical part of the entire picture – how to route user queries to the appropriate information providers efficiently.

The GALANX system is designed to achieve high performance in routing user requests only to the locations where the desired data is stored. There are three features that distinguish GALANX from many other P2P systems and proposals,

1. It does not restrict user queries to be simple lookup requests. In the experiments we demonstrate full-text search queries.
2. It respects the ownership of the data shared in the network and does not require data objects to be cached on other nodes. All user queries are forwarded to the node on which the data resides.
3. Every participating node in the GALANX system has much freedom to decide what meta information (peer indices that are described in Section 3.2) to store on its local disk based on its own preferences and capabilities.

Since query routing is the primary focus of this paper, we have simplified the other two parts of the problem as follows:

1. The data objects stored on each node are simple text files.
2. Each request is a conjunction query that consists of a list of keywords. Every qualified result file must contain all the keywords in the query.

Notice that peer-to-peer networks are usually dynamic by nature. Nodes can join and leave the network at any time. P2P systems should be able to adapt to such structure changes. GALANX has adopted the approach advocated by the JXTA system [14]. In this type of P2P network there are two types of nodes. The first type of nodes is the major information providers. Since the goal of such nodes is to attract as many users as possible, they are usually stable. The other nodes correspond to information "consumers". These nodes come and go. While the design of GALANX assumes that the key information providers will be stable for long periods of time, its design permits nodes to join, leave, and fail gracefully.

3.2. Data Indices and Peer Indices

Before discussing different query routing techniques, we first explain the indices that GALANX employs

- **Data Indices** -- We assume that each node stores a large number of text files. In order to efficiently determine those documents that contain all keywords in a query, an inverted list index [23] is constructed over the data files on each node as shown in Figure 3.1. Given a conjunctive query with multiple keywords, the query engine intersects the document id lists of those keywords to generate the qualified document id list.
- **Peer Indices** -- If the document id list of a keyword in an inverted index is replaced with a list of peer ids on which the keyword can be found, we can construct a peer index, as shown in Figure 3.2. Using a peer index, a node's query engine can perform the same intersection operation to determine if a remote peer has relevant documents. In order to determine whether a peer is potentially relevant to a query, one only needs to check whether it contains all keywords in a query. There are more advanced IR techniques that can be applied to generate more precise peer indices. This paper simply uses the basic peer index as a way to illustrate alternative routing policies. More sophisticated IR methods are beyond our focus for now and will not be discussed here.

3.3. The State of the Art in Query Routing

In this section three common query routing approaches are described.

I. Centralized Indices

Early peer-to-peer file sharing systems, such as Napster [17], employed a centralized index for all data objects. In

$\mathbf{Keyword_1}$	$\mathbf{Doc_{1,1}, Doc_{1,2}, \ldots, Doc_{1,k1}}$
$\mathbf{Keyword_2}$	$\mathbf{Doc_{2,1}, Doc_{2,2}, \ldots, Doc_{2,k2}}$
…	…
$\mathbf{Keyword_n}$	$\mathbf{Doc_{n,1}, Doc_{n,2}, \ldots, Doc_{n,kn}}$

Figure 3.1 The Local Data Index

$\mathbf{Keyword_1}$	$\mathbf{Peer_{1,1}, Peer_{1,2}, \ldots, Peer_{1,k1}}$
$\mathbf{Keyword_2}$	$\mathbf{Peer_{2,1}, Peer_{2,2}, \ldots, Peer_{2,k2}}$
…	…
$\mathbf{Keyword_n}$	$\mathbf{Peer_{n,1}, Peer_{n,2}, \ldots, Peer_{n,kn}}$

Figure 3.2 The Global Peer Index

this type of system one or more dedicated servers are used to hold the complete object index. As nodes join the network they report the data that they want to share with others to the central index server. When a peer receives a user query, it simply contacts the server to determine which peers the query should be sent to. Updates only need to be reported to the central server. This approach does not require any routing capability among the peers.

II. Flooding w/o Indices

Other peer-to-peer file sharing systems, such as Gnutella [6], fully decentralize the data location process and do not build any peer indices. All peers in the system self-organize into an overlay network. A user request for a file will be flooded to peers within a certain scope. More specifically, when a user submits a query, the query will be sent to all neighbor peers (if it cannot be answered by the submitting peer). The query will be forwarded to an exponentially expanding collection of peers until either it is answered or it reaches the hop limit set by the submitting node.

III. Distributed Hashing Table (DHT) Indices

CAN [19], Chord [24], Pastry [3], and Tapestry [13][29] each use a *distributed hash table* (DHT) in which every data object shared is associated with a key, and every participating node is responsible for storing the data objects whose keys fall within a certain range of values. Although their particular designs are quite different from one another, they all provide two basic functions, *put(key, data)* – insert a *data object* into the system given its *key value*, and *get(key)* – return the data object which matches the key value.

When a DHT mechanism is applied in the problem scenario discussed in this paper in which data objects are always stored at their original source nodes, one can construct peer indices on every node to map key values to locations as follows. First, each of the keywords in the global peer index (as shown in Fig. 3.2) is hashed to produce a key value. Then, the global peer index is partitioned and distributed among nodes in the system using these key values. Thus, every node maintains a part of the index, which corresponding to the keywords whose hash values fall into the node's responsible range.

When a peer receives a query, "Matrix movie review", it will first send three individual peer index requests to at most three distinct nodes that hold the peer index listings of the words, "Matrix", "movie", and "review". After obtaining the three lists of peer ids, the node submitting the query intersects the three lists to obtain a list of peers that may have the desired files. Finally, the node sends the query to all the peers on the list and waits for their answers.

The DHT method uses a deterministic way to distribute and locate data (or indices in our case). It is much more scalable than the centralized directory service as all query routing is conducted by the peers individually. Certainly, it is also more scalable than routing strategies that use flooding because every query follows a shorter path to only the relevant nodes that can answer it.

However, this determinism of location that DHTs provide incurs several undesirable side effects.

1. Participating nodes do not have a choice as to how much or what data (or indices) they store, regardless of their capabilities. A peer might also happen to hold some very popular data or indices without being able to handle the volume of requests that the data collection attracts.
2. It is hard, or at least inefficient, to route more complex queries that may consist of multiple structured parts because the number of index lookup messages rises with the complexity of the query.
3. Hash functions are likely to place context-related data objects on different nodes, increasing the overhead of query processing. For example, in the peer index distribution discussed above, given a multi-keyword query, the keywords in a query will most likely reside on different nodes. Thus, the node to which the query was submitted is likely to have to communicate with several keyword-holding peers before it can decide where to send the query.

4. THE GALANX SYSTEM

The goal of the GALANX project is to build a scalable peer-to-peer search system that allows users submitting complex search queries to find relevant information quickly. The first step to achieving this goal is finding an efficient solution to the simplified scenario presented in

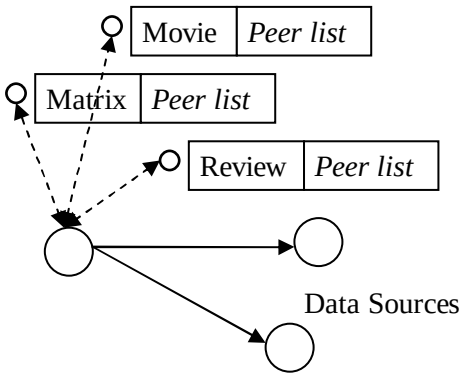

Figure 4.1 A Query Routing Example

Section 3.1. The system should also provide a flexible environment for communicating among peers so that different query routing mechanisms can be evaluated in the system.

This section presents the GALANX query routing approach, along with the design and implementation of the system.

4.1. Routing Queries in the GALANX System

Query routing is a process by which the node on which a query is submitted routes the query to the relevant data source node(s). Naturally, the fewer intermediate nodes involved in the process, the better. In GALANX, every node maintains a peer index table to assist in locating the relevant documents.

In the extreme case, every node would maintain a copy of the complete global peer index that contains all keywords occurring in all files shared in the network. In this case the potential destination nodes of every query can be resolved locally. The problem of this strategy is that, in a dynamic P2P environment, it is too expensive to maintain such a peer index on every peer. Whenever a node joins or leaves the network, or a node adds a new keyword or drops a keyword from its local inverted index, this update must be propagated to every other node in the system. Also, the complete peer index table may be very large considering it may contain millions of keywords and hundreds of thousands of peers.

In a system based on a DHT every node maintains a non-overlapping piece of the complete peer index[2]. The advantage of this approach is that every piece is relatively small, and a keyword update is only sent to a single corresponding peer. However, as discussed in Section 3.3, the use of hashing tends to distribute keywords from the same text file to multiple peers. Consider the example shown in Figure 4.1. A peer will most likely have to perform three individual peer index lookups to three different nodes before it can locate peers that hold the information about "Matrix movie review".

Figure 4.2 illustrates the structure of the peer indices employed by GALANX. These indices consist of two parts, the *direct peer index* and the *indirect peer index*. The *direct peer index* portion contains keywords and full lists of ids of their hosting peers. The *indirect peer index* portion has the rest of keywords and for each such keyword the ids of a few peers that hold the corresponding direct peer index. If constructed properly, this design can improve query routing performance in two ways. First, when a node receives a query, it can resolve the data source nodes if all keywords are in its direct peer index. Second, if any keywords in a query are not in its direct peer index, they can probably be found in the direct peer index on another node. In such a case, only a single lookup message is needed to locate the destination peer list.

One may argue that there are some popular words for which the corresponding peer indices will be stored on essentially every node. In this case an update to one of these words must be broadcast to all the nodes in the network. We believe that this cost is acceptable; the fact that a word is popular means that it occurs frequently in user queries everywhere. In turn, its peer index should be cached everywhere so that peers do not need to look it up on other peers every time they get such queries.

Next we describe two techniques for constructing such peer indices.

I. Direct Construction

When a peer-to-peer network starts with a small number of participating nodes, the size of the complete peer index is relatively small, and the overhead imposed by updates is also relatively low. Thus, every node can host the complete peer index, i.e., all keywords appear in the *direct peer index* while the *indirect peer index* is empty.

As the network grows, a node may decide not to maintain

$\mathbf{Keyword_1}$	$\mathbf{Peer_{1,1}, Peer_{1,2}, \ldots, Peer_{1,k1}}$
$\mathbf{Keyword_2}$	$\mathbf{Peer_{2,1}, Peer_{2,2}, \ldots, Peer_{2,k2}}$
…	…
$\mathbf{Keyword_j}$	$\mathbf{Peer_{j,1}, Peer_{j,2}, \ldots, Peer_{j,kj}}$

Direct Peer Index

$\mathbf{Keyword_{j+1}}$	$\mathbf{Peer_{j+1,1}, Peer_{j+1,2}}$
$\mathbf{Keyword_{j+2}}$	$\mathbf{Peer_{j+2,1}, Peer_{j+2,2}, Peer_{j+2,3}}$
…	…
$\mathbf{Keyword_n}$	$\mathbf{Peer_{n,1}, Peer_{n,2}}$

Indirect Peer Index

Figure 4.2 GALANX Peer Indices

[2] In practice, a DHT system may store a piece of peer indices on multiple nodes to achieve high availability.

the complete peer index. When it drops a keyword from the direct peer index, the keyword is moved to the indirect peer index. To do so, it randomly picks a few peers with shared text files that contain the keyword as the lookup peers for this keyword. It also notifies those nodes so it can update the indirect peer index entry in case one of them later drops the keyword. When a new node joins the network, it obtains one of its neighbor's peer indices which it updates based on its own keyword set.

As discussed in Section 3.1, it is assumed that most nodes (the major information providers) are stable for long periods of time, peer indices are updated primarily when keywords are inserted and deleted. There are two cases when a node gets a new keyword. First, if the word already exists in the network, i.e., it has an entry in the peer index, the update will be sent to its indexing nodes, which will add this peer to the list. The second case occurs when the keyword is new to the entire system. In this case the keyword is broadcast to all nodes so that they can add a new entry to their peer indices. Not all nodes may be interested in knowing this new word as they may not need it until they get a request for it. In this case the system can take a lazy update strategy by having only a few designated peers update their indices with the word. The rest will have a wildcard entry to refer to these nodes so that when they receive a request that contains an unknown word, they can always go to one of the wildcard indexing nodes to lookup the word. When a keyword is dropped from a node's data set, other nodes will learn this when their first query that contains the keyword gets bounced back and then update their peer indices.

II. DHT-based Construction

The previous method builds peer indices incrementally as the system evolves. An alternative is to build peer indices using an existing DHT-based network. At first, the DHT mechanism distributes the global peer indices in pieces to every participating node while coping with the dynamic structure evolution of the network. Every individual node then starts to reshape its peer index as it receives user queries.

At the beginning, a node's *direct peer index* is purely the portion of its corresponding key-space, and its *indirect peer index* is empty. When a node sees a keyword for the first time, if the word is not in its *direct peer index*, it applies the hash function of the DHT to locate the peer that will hold the keyword its index. The keyword is then added to the *indirect peer index* along with the id of the indexing node. If the word occurs frequently in user queries, the node will move the entry to the direct peer index table and expand the peer list to contain all peers that have the relevant data files. When updates occur, the methods in the previous strategy also apply.

The initial construction in both strategies may sound expensive when a large number of nodes join the network, but after the network turns stable, the maintenance cost (a new node joins in once in a while and peers update their data sets) is relatively low, which will be illustrated in the experimental section.

4.2. The Implementation

In order to evaluate our query routing mechanism against different existing approaches, we have implemented a prototype GALANX system.

The architecture of the prototype is shown in Figure 4.3. The Apache® HTTP server handles all communications among peers and provides users a Google-like html query interface. Both the local data index and the peer index are stored in B+-Trees in the BerkeleyDB® Data Store which we have integrated into the Apache Web server.

The query execution engine runs user queries over the local data index as well as processes queries using the peer indices to determine which nodes have relevant data files. The query engine obtains a peer id list for each keyword from the peer index and/or the index lookup results from other peers (through the Apache HTTP server). The peer administrator handles all peer information, such as maintaining the neighbor list and sending and receiving peer index update notifications about which keywords have been added or dropped.

All system functions, including the query engine, the peer administrator, and the user interface are implemented in C and embedded as modules in the Apache HTTP Server. All communications between peers are performed using only http requests. For example, node A asks node B for the peer index of a keyword through an http request. The Apache server on node B gets this request and calls the in-

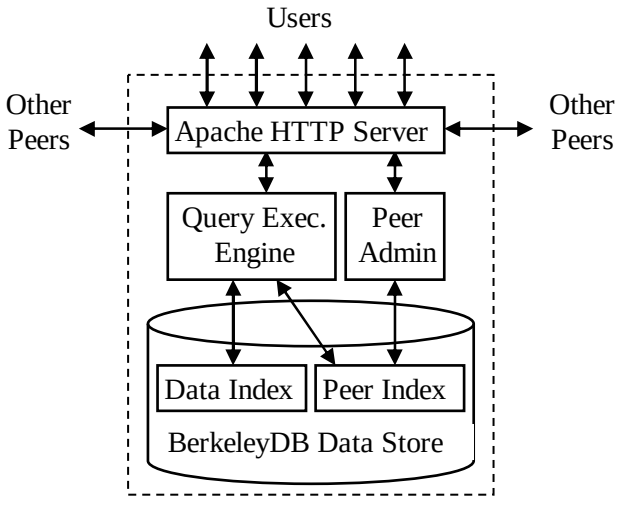

Figure 4.3 The GALANX System Architecture

dex-lookup module to handle the request.

Note that the Apache HTTP server (version 1.3.27) employed in the prototype system runs in a multi-process mode, i.e., each user query, which is an http request, is handled by an individual httpd process. Due to lack of API documentation, we are not able to use the Apache HTTP sever version 2.x that runs in a hybrid multi-process multi-threaded mode. This multi-process query execution mode forces BerkeleyDB to also be accessed in its multi-process mode, which significantly hurts the system performance, in terms of how many users can be served on each node. As will be shown in Section 5.2, in our experiments BerkelyDB has to be set to allow at most 4 processes for concurrent access in order to achieve maximum performance. User queries will be queued when the database access count is full.

The entire system is written in C and has about 16,000 lines of code.

5. EXPERIMENTS AND PERFORMANCE EVALUATION

In this section, we present the experimental results that compare the GALANX query routing mechanism against other routing techniques. Section 5.1 describes the experimental setup, including the data set, the query set, and the peer-to-peer network environment. Then we introduce the performance evaluation metrics in Section 5.2. Finally, Section 5.3 presents the results and evaluates the routing mechanisms.

5.1. Experimental Setup

There are five components in our P2P experiments, the network, the data set, the query set, the users, and several competing routing strategies.

5.1.1. The Network

The experiments were conducted on a 100-node computer cluster. Every machine runs RedHat® Linux 7.2 on a Pentium® III 933MHz or 550 MHz CPU with 1GB memory. All nodes are connected by a 100base-T LAN. The GALANX system is installed on every node.

5.1.2. The Data Set

The nature of this peer-to-peer search engine problem requires the use of a huge amount of data to validate the system scalability. Rather than use synthetically generated data, we wrote a web crawler to download data files from eBay®. The crawler ran for several months and downloaded more than 20 million html files. Each html file contains information about a single item. In order to perform meaningful text search, we took only the item title and description text from the raw html files. Every term in the text was added to the keyword pool after filtering out 570 stop words. For purposes of simplicity we did not stem the entries in the keyword set.

There are about 20 major categories of items on eBay. In an attempt to emulate a scenario in which most information providers are specialized, the item files are distributed to the nodes according to their category. i.e., one node contains auction items of only one major category, and several nodes share the same category. Each node contains 200,000 html files, which amounts to about 6GB.

There are about 2.49 million keywords[3] in the 20 million item files. On average, each node contains 125,000 keywords, i.e., 125,000 entries in the local data index table. The size of the local data index B+-tree is about 300~400 MB.

5.1.3. The Query Set

The data set was also used to generate the query test suite. The number of keywords in a query follows a standard distribution with a mean of 5 and a standard deviation of 1. The keywords were randomly selected from the 2.49 million keywords found in the data set. Only queries that can find at least one qualified item file in the data set are kept in the query pool so that we can check the actual results of a query to determine its *recall*.

From the initial query pool of more than 1 million queries, we generated a 1,000-query set for each user. The queries in every query set follow a Zipf distribution in terms of node frequency – the number of nodes that have files that satisfy the query. Queries with the highest node frequency usually receive results from all nodes in the network while queries with the lowest node frequency generally receive results from a single node.

5.1.4. User Simulation

Each node is assigned a number of users who randomly pick queries from their query set to submit. A user works in cycles thinking and typing before submitting a query [26]. Then he/she waits a certain amount of time to receive results before submitting another query. Every query carries a field that specifies the query expiration time. If the user does not receive any results within the specified time limit, the next submitting query will have a longer expiration time.

5.1.5. Routing Strategies

We implemented several other routing mechanisms in the GALANX framework, namely Gnutella, Chord, Chord+, and Complete.

- **Gnutella** The network topology was generated to obey the Power Laws [18]. On average, every node has

[3] Google claims to have about 14 millions of keywords in about 35 languages [7].

4 neighbors. The average distance between any two nodes in the network is 3.5 hops. The longest distance between two nodes is 9 hops. In order to obtain the complete result for each query, user queries are not given a TTL.

- **Chord** The size of the logical node ring [24] is set to be 2^{32}, so every node maintains a finger table that has 32 entries. When a peer looks up the peer index of a keyword, it follows its finger table to reach the corresponding indexing node through a chain of messages.
- **Chord+** In order to focus on the number of individual index lookup requests sent for each query, we let every node maintain a complete peer map. Thus, a peer can send a lookup request directly to the corresponding peer after applying the hash function on a keyword. Thus, no relay messages on the route to the indexing node in the original Chord approach are required.
- **Complete** In this scenario, every node has a copy of the complete global peer index. This is the best case because a peer does not need to ask other peers to determine where to send a query.

Two sets of peer indices for the GALANX strategy were constructed using the methods described in Section 4.1.

- **GALANX-1** With this approach the *direct peer index* for each node contains all the keywords that are found in the node's local data files while the rest of keywords are stored in the *indirect peer index*.
- **GALANX-2** Starting from the peer indices built using Chord, every node continuously constructs the indices by running a 5,000-query set. The queries in each query set are randomly selected from the query pool. 50% of them are related to the local data, and the other 50% are targeted toward the data shared by other peers.

5.1.6. Peer Indices Evolution

For every routing approach we built corresponding initial peer indices for every node before we started user query simulation. In order to verify the impact of network and data updates on the query routing performance, experiments are conducted in two scenarios. First, a few new nodes with data sets of similar size are added into the network. Second, new documents (with new keywords) are added to existing peers.

5.2. The Performance Metrics

This section defines the metrics we used to measure the effectiveness and efficiency of the different routing strategies. When a user submits a query, he/she would like to obtain all of the qualifying files in the minimum amount of time. We measured both processing speed and result quality.

- **Query Response Time** – The time between submission of a query and when the first result is received. Queries that produce no results are not included in this calculation (see below).
- **Query Response Ratio** – The percentage of queries that produce results. While queries are formulated in a manner that should produce results for every query submitted, because of limited computing resources some queries end up being dropped by a query engine on some peer node. A good routing strategy tries to avoid sending queries to irrelevant nodes to avoid wasting resources on those nodes.

  There are two reasons why some relevant files might be missed by a query. First, when a query arrives at a node, it might not get executed immediately and is put into the query queue where the query expires before it moves out of the queue. Second, the query might not even get forwarded to all relevant data sources because it expires and is dropped.
- **Query Recall** – Of the files that satisfy a query (which we determined offline), the percentage that are actually returned in response to a query.

5.3. Experiments

We ran a set of experiments for each routing strategy, varying the number of users on each peer submitting queries from 1 to 10. Every user submits 100 queries that are randomly picked from its query pool. The average think and type time for each user is 5 seconds. The initial query expiration time was set to 10 seconds. If no results are received for a query, the expiration time of the next query is increased by 50% but no longer than 30 seconds. Otherwise, it is reduced by 50%, but no shorter than 5 seconds.

Figure 5.1 shows the average query response time. As the number of users increases, the performance of Gnutella drops quickly, which clearly indicates the scalability problem of its flooding-based routing strategy. On the other hand, both GALANX-1 and GALANX-2 outperform both Chord and Chord+. They are also close to the best case, Complete, in which every node has a copy of the complete global peer index.

Figure 5.2 illustrates the average number of queries received on each node in different routing cases. With the Gnutella routing strategy, nodes receive many more queries than with the other strategies. Notice that all other routing strategies use peer indices to direct queries to only the potential related nodes. So in the ideal case that no queries are dropped, they should receive the same number of queries. However, when a node handles a query whose

expiration time has passed, it will not process it or forward it to other peers. Thus, peers with Complete receive the most number of queries because they do not need to allocate their resources for answering index lookup requests. Similarly, both GALANX-1 and GALANX-2 receive more queries than the Chord approaches because of more efficient query routing.

Figure 5.3 compares the two GALANX peer indices with Chord+ by plotting the average number of peer index lookup requests received by a node. Neither Gnutella nor Complete are included because these two routing schemes do not employ this type of request. Chord and Chord+ are essentially the same except Chord+ does not need other nodes to relay the peer index lookup requests. Nodes send out almost half of the number of lookup requests in the GALANX approaches because it is likely that they can either resolve a keyword in their own *direct peer indices*, or resolve multiple keywords through a single lookup request. Since a peer processes fewer index lookup requests in both the GALANX cases, they can devote more resources executing user queries. Thus, user queries have faster response times.

These experiments have stressed each node in the network to its limit in terms of computing capacity. Queries are queued when they are received by a node if it is not able to process them. In turn, some of them are dropped once their expiration time has passed. Figure 5.4 presents the percentage of the queries received by a node that are processed before expiring. The Gnutella routing strategy has the lowest execution percentage because many queries time out waiting in a query queue. On the other hand, the Complete strategy has the highest percentage of completion because nodes do not need to handle peer index lookup requests. The performance of the GALANX strategies is better than the ones based on the Chord protocol because fewer lookup requests leave more resources available for executing queries. Figure 5.5 shows the same trend in query response ratios – the more queries actually executed, the higher the percentage of queries that actually obtain results.

Figure 5.6 examines the result quality. Obviously, the GALANX approaches return more relevant results than the Chord approaches because more queries are executed successfully.

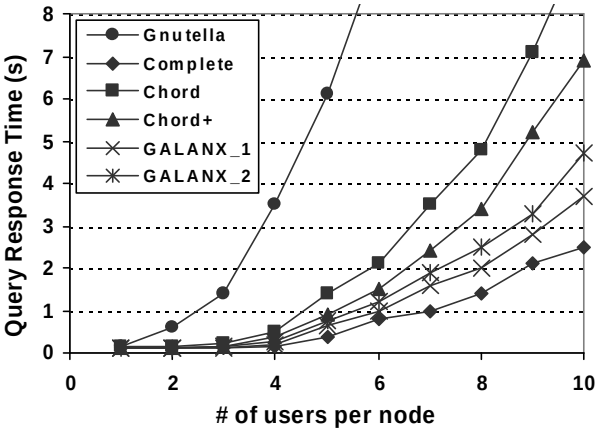

Figure 5.1 Query Response Time

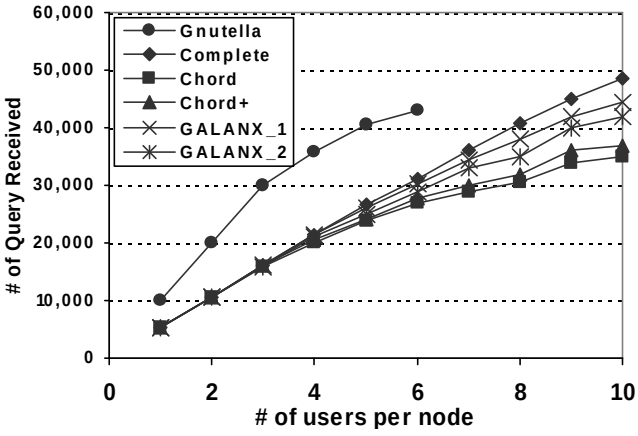

Figure 5.2 # of Queries received on Each Peer

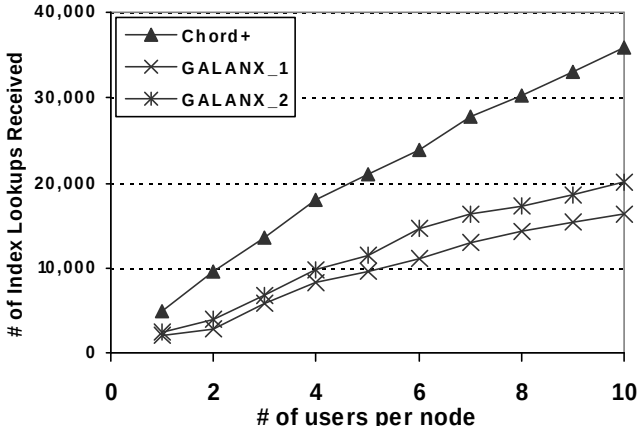

Figure 5.3 # of Index Lookup Requests Received

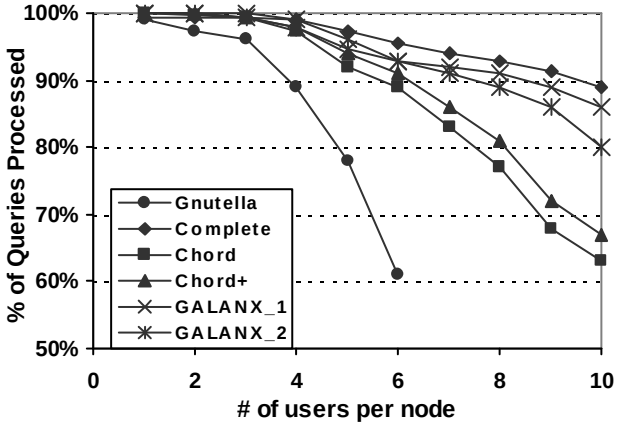

Figure 5.4 Query Execution Ratio

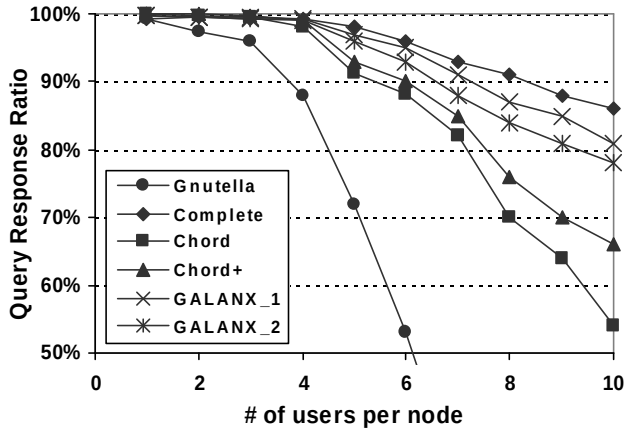

Figure 5.5 Query Response Ratio

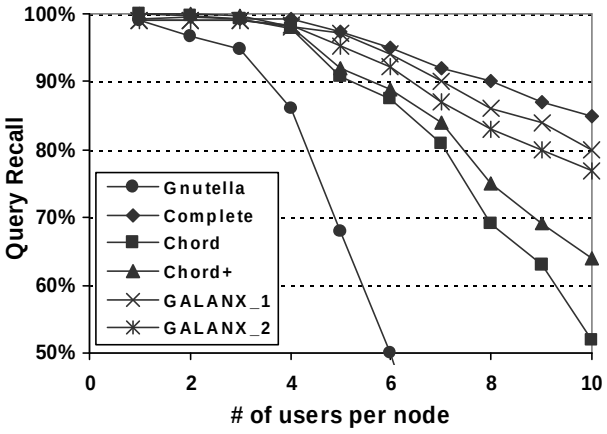

Figure 5.6 Query Recall

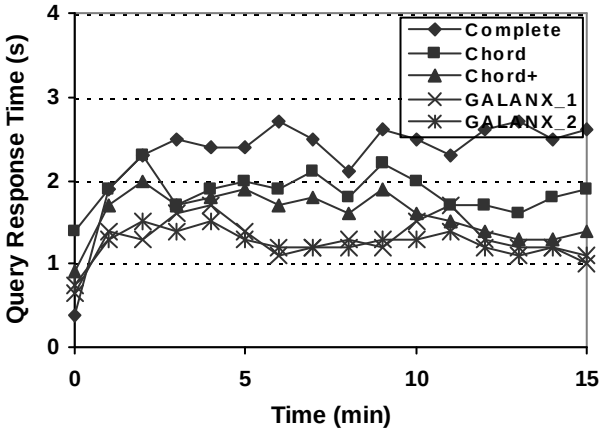

Figure 5.7 Query Response Time (with new data)

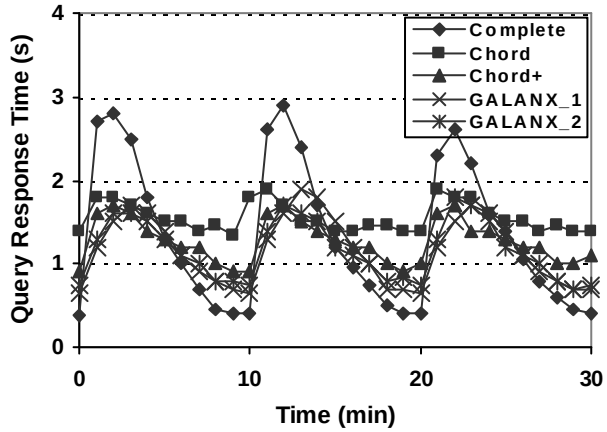

Figure 5.8 Query Response Time (with new nodes)

The results of previous experiments are consistent with our expectations. The combination of the *direct peer* and *indirect peer indices* can quickly locate relevant data sources for queries. Such data oriented peer indices are more flexible and efficient than DHT-based mechanisms.

In addition, we also conduct two experiments simulating data updates and network evolution to verify the impact of the update cost on the system's query routing performance. First, every 5 minutes we randomly select 5 nodes (5% of the network) and add 1% more new documents to their data sets. The data insertion occurs independently on each node and randomly during the period. Each node is connected with 5 users. Figure 5.7 presents the average query response time of the entire network in the period. The Complete approach is the victim of such intensive data update because every node in the network has to modify their direct peer index for new keywords found in any nodes. Although more index updates are needed in two GALANX strategies than the two Chord methods (distributing keywords and updates corresponding *direct peer indices)*, both GALANX approaches still out-perform the Chord methods.

Finally, we test those query routing techniques in the situation of network evolution. During a 30-minute period, we add 3 new nodes into the network at minute 0, 10, and 20. The new peers contain the similar size of documents. Each one of the original 100 nodes is connected with 5 users, and no users in the new nodes. Figure 5.8 illustrates the average query response time over the period. The Complete strategy has the highest peak values that indicate user queries are much delayed by the peer communication and updates after new nodes join the network. Both Chord methods do not suffer much because they only have to redistribute some keyword entries (according to the hash function) in the peer indices to the new nodes. The two GALANX approaches have similar peak values as the Chord methods, because the higher update cost is balanced by the lower query routing expense. The last two experiments demonstrate that the GALANX system can still perform efficient query routing in the dynamic network environment.

6. CONCLUSIONS AND FUTURN WORK

In this paper we present a new mechanism for efficiently routing queries in a peer-to-peer search engine system. The *direct peer index* and *indirect peer index* are introduced

along with two index construction strategies. We have designed and implemented GALANX, a prototype P2P search engine system, in which our approach is compared against several existing query routing techniques. The experimental results indicate that the peer indices used by GALANX significantly reduce the communication cost among peers in locating the nodes with data files relevant to a query. As a result, our routing methods help users obtain results faster and with a more complete result set.

To meet some real word challenges [2], there are two extensions that can be launched on the GALANX platform. First, we intend to extend GALANX to support more complex forms of queries. Second, information retrieval techniques, which used to be applied in static and centralized environments, such as relevance ranking, can be introduced into the system.

7. REFERNCES

[1] A. Crespo, H. Garcia-Molina. "Routing Indices For Peer-to-Peer Systems", in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[2] N. Daswani, H. Garcia-Molina, B. Yang. "Open Problems in Data-Sharing Peer-to-Peer Systems", in *Proceedings of the 9th International Conference on Database Theory (ICDT'03)*, 2003.

[3] P. Druschel, A. Rowstron. "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[4] The Freenet website, http://freenet.sourceforge.net.

[5] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, D. Suciu. "What Can Peer-to-Peer Do for Databases, and Vice Versa?", *WebDB'01*, 2001.

[6] The Gnutella website, http://www.gnutella.com.

[7] The Google website, http://www.google.com.

[8] The Grub Project website, http://www.grub.org.

[9] L. Galanis, Y. Wang, S.R. Jeffery, D.J. DeWitt. "Processing Queries in a Large Peer-to-Peer System", in *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE 2003)*, 2003.

[10] L. Galanis, Y. Wang, S.R. Jeffery, D.J DeWitt. "Locating Data Sources in Large Distributed Systems", in *Proceedings of the 27th International Conference on Very Large Data Bases* (*VLDB'03)*, 2003.

[11] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, I. Stoica. "Complex Queries in DHT-based Peer-to-Peer Networks", *1st International Workshopon Peer-to-Peer Systems*, 2002.

[12] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, I. Stoica. "Querying the Internet with PIER", in *Proceedings of the 27th International Conference on Very Large Data Bases* (*VLDB'03)*, 2003.

[13] K. Hildrum, J.D. Kubiatowicz, S. Rao, B.Y. Zhao. "Distributed Object Location in a Dynamic Network", *Proc. ACM Symp. Parallel Algorithms and Architectures*, 2002.

[14] The JXTA project website, http://www.jxta.org.

[15] The KazaA website, http://www.kazaa.com.

[16] The Morpheus website, http://www.musiccity.com.

[17] The Napster website, http://www.napster.com.

[18] C.R. Palmer, J.G. Steffan. "Generating Network Topologies That Obey Power Laws", *IEEE Globecom 2000*, 2000.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. "A Scalable Content-Addressable Network", *ACM SIGCOMM'01*, 2001.

[20] S. Raghavan, H. Garcia-Molina. "Crawling the Hidden Web", in *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, 2001.

[21] S. Ratnasamy, S. Shenker, I. Stoica. "Routing Algorithms for DHTs: Some Open Questions", *the First International Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.

[22] The Search Engine Watch website, http://www.searchenginewatch.com.

[23] Gerard Salton, Michael J. McGill. "Introduction to Modern Information Retrieval", McGraw Hill, New York, 1983.

[24] I. Stoica, R. Morris, D. Karger, M. Kaashoek, H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", *ACM SIGCOMM'01*, 2001.

[25] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, K. Shanmugasundaram. "ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Rerieval", in *Proceedings of the International Workshop on Web and Databases (WebDB'03)*, 2003.

[26] TPC-C Benchmark Standard Specification Revision 5.0.

[27] D. Tsoumakos, N. Roussopoulos. "A Comparison of Peer-to-Peer Search Methods", in *Proceedings of the International Workshop on Web and Databases (WebDB'03)*, 2003.

[28] B. Yang, H. Garcia-Molina. "Improving Search in Peer-to-Peer Networks", in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[29] B.Y. Zhao, J.D. Kubiatowicz, A.D. Joseph. "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing", UC Berkeley Computer Science Division Report No. UCB/CSD 01/1141, 2001.