

PRISM: Indexing Multi-Dimensional Data in P2P Networks using Reference Vectors*

O. D. Sahin A. Gulbeden F. Emekci D. Agrawal A. El Abbadi

Department of Computer Science
University of California Santa Barbara

{odsahin, gulbeden, fatih, agrawal, amr}@cs.ucsb.edu

ABSTRACT

Peer-to-peer (P2P) systems research has gained considerable attention recently with the increasing popularity of file sharing applications. Since these applications are used for sharing huge amounts of data, it is very important to efficiently locate the data of interest in such systems. However, these systems usually do not provide efficient search techniques. Existing systems offer only keyword search functionality through a centralized index or by query flooding. In this paper, we propose a scheme based on *reference vectors* for sharing multi-dimensional data in P2P systems. This scheme effectively supports a larger set of query operations (such as k-NN queries and content-based similarity search) than current systems, which generally support only exact key lookups and keyword searches. The basic idea is to store multiple replicas of an object's index at different peers based on the distances between the object's feature vector and the reference vectors. Later, when a query is posed, the system identifies the peers that are likely to store the index information about relevant objects using reference vectors. Thus the system is able to return accurate results by contacting a small fraction of the participating peers.

Categories and Subject Descriptors: H.3.3 [Information Search and Retrieval]: Retrieval models C.2.4 [Distributed Systems]: Distributed applications

General Terms: Algorithms, Design, Experimentation

Keywords: Peer-to-Peer Systems, Similarity Search, Reference Vectors

1. INTRODUCTION

With the recent advances in storage and networking technologies, more data is being shared and circulated over the Internet. One example of such an environment is the Web, where people and organizations serve data through their web

*This research was supported in parts by NSF grants CNF-04-23336, IIS-0209112, and IIS-02-23022.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MM'05, November 6–11, 2005, Singapore.

Copyright 2005 ACM 1-59593-044-2/05/0011 ...\$5.00.

sites. Another example is the P2P file sharing systems that have a large number of users scattered all around the world, sharing their data with each other. It is a challenging task to search and locate the data of interest in such systems. Existing models, which are usually centralized, are not scalable enough to work in a large scale distributed setting. Additionally, most of these models only offer *keyword search* functionality, i.e., they return the objects that are associated with a given set of keywords. This might be reasonable when searching for a particular term over the Internet or looking for a file in a P2P file sharing system. However, keyword search may be insufficient in other cases and more powerful search mechanisms are needed.

A large fraction of the data shared in these systems is text, image, or video, which can be represented with multi-dimensional vectors. Using these feature vectors, a more expressive set of query operations such as *k Nearest Neighbor* (k-NN) queries and content-based similarity searches can be supported. k-NN search is an important query operation over high dimensional data as it finds the *k* objects from a data set that are most similar to a given object in terms of a distance function, such as the Euclidean or cosine distance. Some example k-NN queries are searching for the documents that are similar to a query document in terms of their content, or retrieving the images whose color features are similar to those of a particular picture. In a distributed environment with a large number of nodes sharing data, it may not be feasible to access all nodes to answer a query or to replicate the data at multiple places due to the high communication and storage costs involved. An important observation that can be made in such cases is that users are usually willing to accept approximate answers given that the retrieval cost is much less than computing the exact answers.

In this paper, we introduce and evaluate PRISM (which stands for “P2P Reference-based Index for Sharing Multi-Dimensional Data”), a novel method for sharing and locating multi-dimensional data in a large scale distributed system. PRISM effectively supports a larger set of query operations (such as k-NN queries and content-based similarity search) than current systems, which generally support only exact key lookups and keyword searches. It operates on top of a Distributed Hash Table (DHT), a class of P2P systems that offers very efficient exact key lookups, and uses a set of global reference vectors to identify the peers that are likely to contain relevant data to a given query. Due to the underlying DHT layer, the system harnesses the resources, such as storage and CPU, available at the user machines and operates in a decentralized manner without relying on

a central component. In addition, it has valuable properties such as scalability, dynamic node insertion and departure, and self-organization.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 gives an overview of PRISM. The reference-based distribution scheme is described in Section 4. Section 5 discusses some optimizations to the basic design such as load balancing and using different dimensionality reduction techniques to reduce the index size (which in turn translates to reduced communication cost). The experimental setup and the results are presented in Section 6, and the last section concludes the paper.

2. RELATED WORK

P2P systems, which allow the sharing of distributed resources in a decentralized manner, has gained a lot of attention with the popularity of file sharing applications such as Napster [17] and Gnutella [10]. These applications allow their users to exchange data with each other over the Internet. Locating the data of interest in these systems is accomplished by either employing a central server that keeps track of all the data in the system [17] or flooding the query to the network [10]. In order to improve search efficiency, researchers have proposed different techniques such as keeping at each peer routing indices that contain information about the documents stored at neighbors [4, 27], returning the same answers for similar queries [13], maintaining a replica of the global directory at each peer through gossiping protocols [5], and assigning peers into topic-segments based on the documents they are sharing [1].

Since the search mechanisms provided by the file sharing P2P applications are not efficient and effective enough, another group of P2P systems, called the *structured P2P systems*, has appeared. These systems dynamically assign each object to a peer in the system by hashing the object's key. They are also referred to as *Distributed Hash Tables (DHTs)* because they implement hashtable-like functionality in a distributed manner, such that it is possible to insert and retrieve data efficiently using a key (Examples are Chord [23], CAN [19], Tapestry [28], and Pastry [21]). DHTs offer very efficient lookup performance (logarithmic or sub-linear), however they only support exact key matching.

Several techniques provide *keyword search* capabilities over DHTs. One approach inserts the corresponding index information for each keyword associated with an object into the system. Each keyword is mapped to a peer in the system through hashing and that peer is responsible for storing the list of objects associated with this keyword. Reynolds et al. [20] discusses three methods to facilitate this approach: using Bloom filters, caching, and calculating the answer incrementally. Gnawali [9] proposes to hash a set of keywords for efficient multiple keyword matching. eSearch [25] publishes the documents only for the top keywords that are determined through information retrieval techniques. Another approach is to split each keyword into *n-grams*, i.e., distinct substrings of length n , and then to insert index information for each of these *n-grams* into the system [12].

There have been other techniques that provide similar functionality to PRISM. pSearch [26] uses some information retrieval techniques for content-based text search. It represents documents with feature vectors and then uses these vectors as the coordinates to store the documents in a CAN [19] system. Hence the documents that are seman-

tically similar also appear close to each other in the CAN system. Each query is then routed towards its feature vector within the system and flooded to a small number of neighbors at the destination. As a result of the underlying CAN layer, however, the amount of routing state maintained increases with the number of dimensions and also it is very hard to provide dynamic load balancing in pSearch. Sahin et al. [22] use a set of reference vectors to map document indices onto a Chord ring based on their feature vectors. The proposed scheme maps each reference pair to a single point on the ring, and thus runs into load balancing problems. If a reference pair is selected for a large number of document indices and queries, then the corresponding peer gets overloaded. Zhu et al. [29] propose using locality sensitive hash functions to distribute the document indices based on their feature vectors, thus allowing semantic based access to the files. In [16], peers are advertised using their compact data summaries for efficient content-based P2P image retrieval. ODISSEA [24] investigates the issue of designing a P2P-based search engine for the Web. Li et al. [15] analyze the feasibility of P2P keyword search over the Web and conclude that it is feasible with optimizations and compromises.

3. PRISM OVERVIEW

PRISM provides a framework for answering approximate similarity search queries in a distributed environment, where there can be a large number of nodes sharing data. In such a setting, the naive solution of contacting each participating node for a query is not feasible due to the high communication cost involved. Similarly, keeping a central index of all shared data introduces scalability and robustness concerns. Hence PRISM introduces a novel scheme that distributes the index over participating peers and returns accurate answers by contacting only a few peers. It also offers tunable system parameters that enable users to trade search accuracy with storage and communication cost.

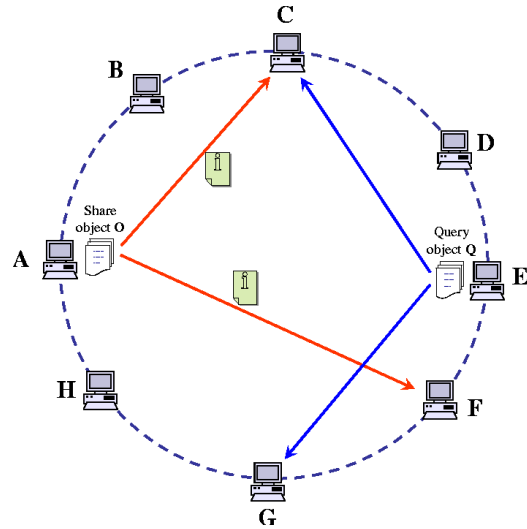


Figure 1: PRISM Overview

Figure 1 depicts the general architecture of PRISM. The participating peers are organized into a structured P2P overlay network, Chord [23] in particular¹. Chord uses an m -bit

¹Any other DHT can be used instead of Chord.

identifier ring, $[0, 2^m - 1]$, for routing and locating objects. Both the objects and the peers in the system are assigned m -bit keys through a uniform hash function and mapped to the identifier ring. An object is stored at the closest peer following it on the ring. Every peer maintains a finger table with pointers to other peers that are at exponentially increasing distances from the peer on the identifier ring. Using these finger tables, a lookup request is delivered to its destination via $O(\log N)$ hops. Chord also allows dynamic peer joins and departures.

For each shared object, PRISM stores the corresponding index, which contains the feature vector of the object and the IP address of the peer sharing it, at multiple locations in the system. Indices are mapped onto the Chord ring and stored at the peers responsible for these locations. Since the indices are small compared to the original objects, this does not create any significant storage overhead on the peers. When a query is initiated, it is routed only to a few peers that are most likely to contain the indices of relevant objects. In Figure 1, the index of object O shared by peer **A** is stored at 2 locations, peers **C** and **F**. Later, peer **E** queries the system for an object Q which happens to be similar to O . The query request is then forwarded to 2 peers in the system. In this case, one of these requests reaches **C** and the index of O is returned to **E**.

Due to the underlying DHT layer, PRISM supports efficient routing and lookup operations, allows dynamic peer insertion and departure, and is self-organizing. The mapping of object indices to peers and routing of queries are accomplished through Chord. Chord requires each object to have an m -bit key. Thus the object indices and query lookups are mapped to IDs. To provide accurate answers to approximate similarity search queries by contacting a small number of peers, this mapping should satisfy the following requirements in particular:

- Index distribution should be based on the same distance function as the retrieval.
- Indices of similar objects should be stored at the same peer.
- It should be possible to identify the peers that are likely to store the indices of objects that are similar to a given query.

To meet these requirements, PRISM maps objects to IDs based on their feature vectors. However it is challenging to semantically map multi-dimensional feature vectors to a flat ID space, i.e., the Chord ring. With the above requirements in mind, PRISM uses a set of *Reference Vectors* for distributing object indices and querying an object. Basically, a set of reference vectors, $R = \{R_0, R_1, \dots, R_n\}$, are selected at the time of system startup and used by all peers in the system. For a given object, the distance of the object to the reference vectors is used when inserting indices or querying². The main idea behind the use of reference vectors is that ‘if two objects are similar to each other, then their distances to the reference vectors are also similar’. As a result of reference-based distribution, similar objects are mapped to the same locations in the system and queries are routed to relevant locations.

²This scheme bears similarity to the landmark routing optimization proposed in CAN [19], where peers measure their RTTs to a set of landmarks to determine the region they join in the virtual space.

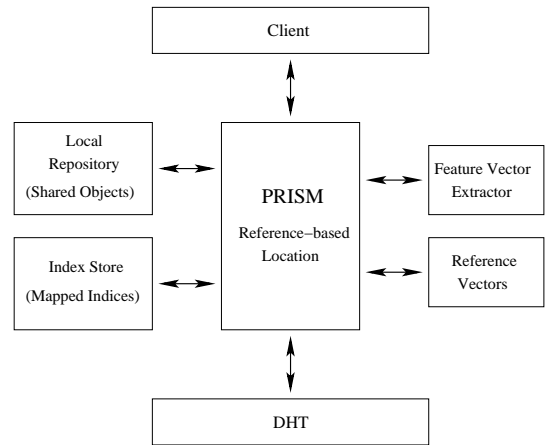


Figure 2: Components of a PRISM Node

Figure 2 shows the components of each PRISM peer. A client uses PRISM to share local objects with other peers or query the system. It specifies the number of index replicas for shared objects that should be stored in the system and the number of lookup requests to be sent out for queries. Each PRISM peer has a local repository of the objects it shares with other peers. It also stores in its index store the object indices mapped to itself. Each peer uses a Feature Vector Extractor component to extract the feature vectors of objects. It is essential that all peers use the same extractor since both index distribution and lookup operations are based on feature vectors. Peers also keep a set of Reference Vectors, which are used to determine the IDs associated with object indices and query lookups. The reference set is fixed at system startup and every peer uses the same set of references. The dimensionality of the reference vectors are of the same size as the object vectors. The details of how the reference vectors are selected and used will be discussed in Section 4.

Once the IDs associated with an operation are determined, PRISM routes corresponding *publish index* or *lookup request* messages through the underlying DHT layer using the IDs as the destinations. A *publish index* message contains the index of the corresponding object. Similarly, each *lookup request* message has the feature vector of the query object and the address of the querying peer. Upon receiving a *publish index* message from the DHT layer, the local PRISM component stores the index in its index store. The index entries expire after a timeout period and are deleted from the index store. Thus the indices of shared objects are refreshed periodically. With this refresh scheme, stale index entries (indices of the objects that are no longer shared) are automatically deleted and an index is reinstalled at another peer if the peer storing the index leaves the system.

4. REFERENCE BASED DISTRIBUTION

Reference vectors are the core of PRISM as they are used to determine the locations on the Chord ring for storing object indices and sending query requests. For a given object, the associated Chord keys are computed based on the distances between the object and the reference vectors.

Once the feature vector of an object is extracted, the distance of the object’s vector to each reference vector is computed. Then the reference vectors are sorted based on their

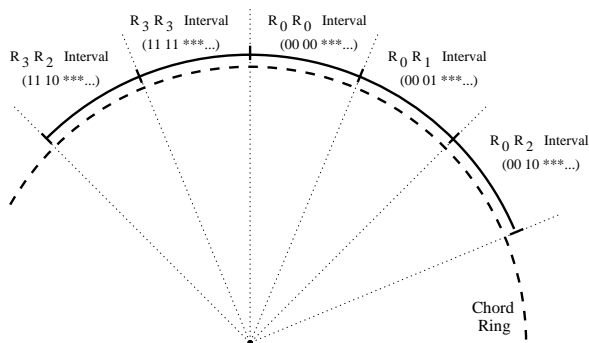


Figure 3: Reference pair key intervals for a system with 4 reference vectors, R_0 to R_3

distances to the object such that the reference most similar to the object comes first. At this point, pairs of reference vectors from the top positions of the sorted reference list are used to compute associated Chord keys for the object³. For each selected reference pair, a Chord key is constructed as follows: the high order bits of the key are constructed by concatenating the binary representations of the selected reference vectors' id's, and the remaining bits are set randomly (random offset). This scheme assigns a key interval on the Chord ring for every possible reference pair as shown in Figure 3.

In order to fully utilize the Chord ID space, the number of reference vectors should be a power of 2. Assuming that there are 2^k reference vectors, there will be $2^k * 2^k = 2^{2k}$ reference pair intervals on the ring and the upper $2k$ bits of the Chord keys will be assigned from the reference id's (k bits for each reference vector). Figure 3 shows some of the key intervals for a system with 4 reference vectors.

The top reference vectors are used when constructing the IDs associated with an object because they are most similar to the object and thus good candidates for identifying it. The reason for using reference pairs is because it represents a good balance between distribution granularity and retrieval accuracy. Note that using more reference vectors to construct a Chord key results in more fine-grained key distribution (so less number of reference vectors are required to get a certain number of key intervals, thus the size of the global state is reduced), but it also results in reduced accuracy as the chances of a query and a similar object being distributed on the same reference vector group is smaller.

4.1 Publishing an Object

When a peer wants to share an object, it sorts the reference vectors based on their similarity to the object. Then it picks a set of reference pairs for which the object's index will be published in the system. For each selected pair, it constructs an ID and sends a *publish index* message toward that ID through the DHT layer. The peer that is responsible for ID receives the message and stores the index in its index store. Algorithm 1 sketches the object insertion operation. Note that due to the random offset in the generated ID, the index will be stored at a random peer within the corresponding reference pair interval.

The number of index replicas to be published for an object, *indexPerm*, is a system parameter, and poses a trade-

³The same reference vector may be selected twice for a pair.

Algorithm 1 Insert *object*'s index at *indexPerm* locations in the system

- 1: Extract the feature vector of *object*
 - 2: Calculate *object*'s distance to each reference vector
 - 3: Sort reference vectors based on distance to *object*
 - 4: **for** $i = 1$ to *indexPerm* **do**
 - 5: Select the reference pair to publish index
 - 6: Construct corresponding Chord key
 - 7: Send *publish* message toward generated key
 - 8: **end for**
-

off between the storage space and recall rate. Storing more replicas of an object's index, i.e., publishing the object for more reference pairs, requires more storage but also provides better chances of finding this object when it is in fact similar to a query. A peer can decide on the value of *indexPerm* for an object depending on different criteria. For example if the object is considered important, its index can be replicated more. Note that the indices of shared objects should be refreshed periodically.

4.2 Querying for an Object

The querying process is similar to index distribution. Again, a set of reference pairs from the sorted reference list is selected and a lookup request is sent for each pair (see Algorithm 2). Once a lookup request reaches its destination, that peer will forward it to other peers within the reference pair interval, because the indices are distributed randomly within an interval. Each lookup request contains the address of the querying peer and the feature vector of the query object. As a result, every peer that receives the lookup request can compute the similarity between the query vector and the object vectors in its index store, and returns the qualifying entries to the querying peer. That peer then aggregates the returned results and presents them to the user.

Algorithm 2 Query the system for *qObject* by sending lookup requests to *queryPerm* locations

- 1: Extract the feature vector of *qObject*
 - 2: Calculate *qObject*'s distance to each reference
 - 3: Sort reference vectors based on distance to *qObject*
 - 4: **for** $i = 1$ to *queryPerm* **do**
 - 5: Select the reference pair to send lookup request
 - 6: Construct corresponding Chord key
 - 7: Send *lookup* message towards generated key
 - 8: **end for**
 - 9: Aggregate returned answers and return the result
-

The querying peer can also specify the similarity measure to be used, such as Euclidean or cosine similarity, for its query. Since the peers receiving the request will have the feature vectors for both the query and locally stored indices, they can use the specified similarity measure when calculating the local results.

The number of reference pairs to lookup a query, *queryPerm*, is another system parameter and it poses a trade-off between accuracy and communication cost. When *queryPerm* is increased, the query is sent to more peers. This increases the probability of finding correct answers, but also incurs communication overhead in the system.

A peer can also choose to perform a progressive lookup so that the query answer is incrementally refined at each

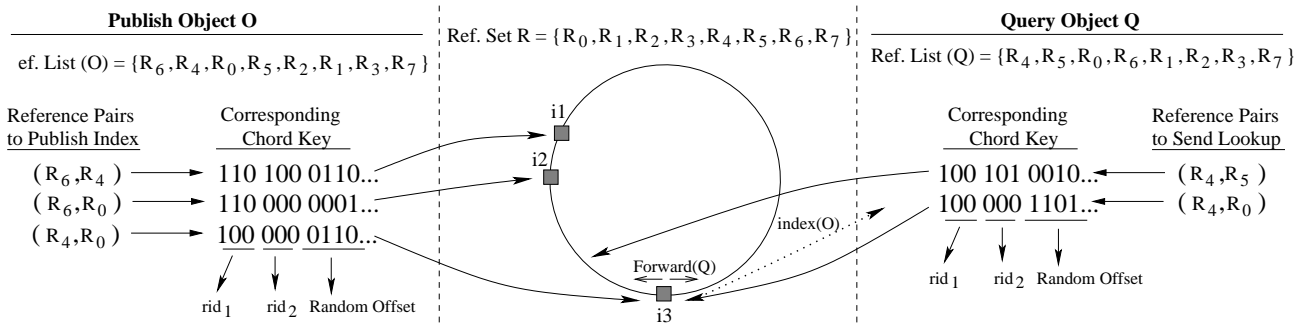


Figure 4: A simple example for publishing and querying an object in a system with 8 references

step. In this case, the querying peer generates only one lookup request for the query. If the results returned by this lookup are not satisfactory, then another lookup request corresponding to some other reference pair is generated and the query answer is refined using the results of this lookup. The querying peer continues the progressive lookup process until the cumulative results returned are satisfactory.

Example. Figure 4 illustrates object publishing and querying operations over a system with 8 reference vectors. When publishing object O , the reference vectors are sorted and 3 reference pairs are selected for publishing the index. Then three replicas of O 's index, i_1, i_2 , and i_3 , are stored on the Chord ring using the corresponding IDs. Later, a query for object Q , which is similar to O , is issued. The lookup requests are sent for two reference pairs that are selected from the sorted reference list. Although the lookup request for the first pair (R_4, R_5) cannot locate any of O 's indices, the second one for pair (R_4, R_0) is received by the peer that is storing i_3 (possibly after the request has been forwarded within the key interval), and thus the index is returned to the querying peer. That peer can now obtain the address of the peer P sharing O from the index and then contact P .

4.3 Selecting Reference Vectors

The reference vectors are selected at system startup and then used by all peers in the system. Hence it is important to decide the number and values of the reference vectors. The peer that initiates the system picks the reference vectors. Thereafter the new peers obtain them during the join process from the existing peer they contacted. It is necessary for each peer to use the same reference vectors and reference id's in order to have a consistent Chord key construction. Two important decisions regarding the reference vector selection are the number of reference vectors and the actual vectors to be used. Below we discuss how these decisions are made.

4.3.1 Number of Reference Vectors

Since the number of reference vectors determines the number of key intervals, it is a trade-off between accuracy and query cost. With more references, the probability of finding similar objects to queries reduces because there are more intervals the objects can be distributed to. On the other hand, it reduces the cost of querying because the reference pair intervals will contain less number of peers to which the lookup requests should be forwarded. It is a good idea to keep the number of reference pair key intervals around or more than the number of peers as discussed in Section 6.2.

4.3.2 Reference Vector Selection

The simplest approach is to pick random reference vectors (*Random Selection*). For instance, the initiating peer might select random objects among those that will be shared within the system and use their feature vectors as references. More educated selections can be made assuming that the initiating peer has access to a sample of objects to be shared in the system. In this case, the peer samples the objects and picks the references based on the sample set. Assuming that the sample set is a good representative of the overall system, this selection scheme is expected to perform better than *Random Selection*. We will consider the following two schemes for reference selection from a sample set.

Cluster Selection: The sample set is clustered and cluster centroids are selected as reference vectors. The idea here is that the references represent concept groups for shared objects and thus the reference based distribution will be more accurate since the indices of the objects within a concept will be stored at the same place.

Partition Selection: The sample set is partitioned using a space partitioning algorithm and the center of each partition is used as a reference. This scheme is expected to provide a balanced distribution of the indices among the reference pairs and thus result in more consistent querying cost (number of peers contacted).

4.4 Supported Query Operations

Since PRISM distributes object indices based on feature vectors, it is able to support a larger set of query operations, such as k-NN queries and content-based similarity search, than current systems that generally support only exact key lookups and keyword searches. We now describe how different types of queries can be implemented using PRISM:

Exact Key Lookup: Note that DHTs are designed for very efficient exact key lookups. Since PRISM uses an underlying DHT layer, it benefits from this layer to support exact key operations. When an object is shared, PRISM stores replicas of its index in the system using the reference vectors, and also inserts the object into the underlying DHT layer using its key. The DHT layer then hashes the key to an ID and stores the object's index⁴ at the peer responsible for that ID⁵. Given an exact key lookup request, PRISM queries the underlying DHT for that key to get the results.

⁴The index used at the DHT layer is different from the index used by PRISM. The DHT index does not contain the feature vector of the object. In addition to the object's key, it may contain the object itself or the address of the peer sharing the object, depending on the DHT implementation.

⁵DHTs usually replicate the index at multiple locations for

Keyword Lookup: PRISM can use any of the techniques described in Section 2 to implement keyword search functionality at the DHT layer. For example, for a given set of keywords associated with a shared object, it can insert the corresponding index information into the DHT layer for each keyword using the keyword as key. A query with a list of keywords can be answered by querying the DHT layer for each keyword to retrieve the list of associated objects and then taking the intersection of returned results.

k-NN and Similarity Range Queries: For a given query object O , a k-NN query searches for k objects shared in the system that are most similar to O , whereas a similarity range query is interested in all objects that are within distance r from O (The values for k and r are to be specified by the query). PRISM can return accurate approximate answers for both types of queries using the querying scheme given in Section 4.2, however the result set returned by a peer that receives the lookup request changes depending on the query type. For a k-NN query, the peer searches its index store and returns k indices that are most similar to the query. For a similarity range query, it returns all indices that are within distance r from the query.

4.5 Discussion

When generating the ID of a reference pair, the least significant bits are set randomly (random offset). This is particularly useful for load balancing purposes, since it provides a better distribution of indices over the ID space. Without random offset, a reference pair would always map to the same ID causing load imbalance in the system, and load balancing would be very hard in this case [22, 25]. As a result of this scheme, however, the lookup requests have to be disseminated within a key interval thus increasing query latency. One possible way to reduce query latency is to replicate object indices at every peer within an interval [11]. In this case, *publish index* messages will be forwarded within the key interval and stored at every peer, while the lookup request will go to a single peer and retrieve all results without any forwarding.

PRISM requires the feature vectors of all objects and reference vectors to have the same dimensionality. This requirement is reasonable if all objects are of the same type, such as images, text documents or mp3 files, since the feature vectors will be extracted using the same method. If the system should support objects of different types, the feature vectors will probably be of different sizes. In this case, PRISM needs to maintain a different pair of Feature Vector Extractor and Reference Vector set for each different object type. However, it can still use the same underlying DHT structure to store indices for all types because the DHT layer is only presented with IDs and is unaware of PRISM-level properties such as object types and feature vectors.

A limitation of the current PRISM design is that the reference vectors should be set initially and cannot be changed later. Here we propose a method for dynamically changing the references vectors without disrupting the normal operation. Assume that a well-known node (the peer that initiated the system or an external monitor) periodically samples the objects shared in the system and computes a new set of reference vectors. If the difference between the currently used reference set and the new one is significant (greater than a certain threshold), then a *ReferenceUpdate* operation fault tolerance and efficiency purposes.

is initiated by broadcasting the new reference set to all peers in the system⁶. Each peer receiving this message enters a *transition period* that lasts for a duration of δ . During the transition period the peer keeps both the current and the new reference sets for inserting indices and querying (The number of index replicas and query lookup requests double). When the transition period ends, the peer removes the current reference set and starts using only the new reference set. The transition interval, δ , should be selected based on the estimated broadcast time, i.e., the time for the broadcast message to reach all peers. Selecting a large enough value for δ ensures that no results are missed during the transition to the new reference set. Since the index replicas distributed with the old reference set are not refreshed any more, they will disappear after a while.

5. OPTIMIZATIONS

In this section, we discuss two optimizations to the basic design discussed in the previous sections.

5.1 Load Balancing

Original DHT designs use uniform hashing to assign IDs to objects, thus providing a uniform distribution of IDs over the ID space. However PRISM uses the semantics of the objects, i.e., the feature vectors, to distribute the indices. As a result, the distribution of the IDs is not uniform and can be very skewed. Since the peer IDs are still uniform, some peers might end up storing most of the indices and getting a lot of lookup requests. Therefore PRISM employs explicit load balancing techniques to address this problem. It implements two schemes for load balancing.

Static Load Balancing: When joining the system, a peer contacts k random peers and then joins at the peer with the highest load. It splits the interval of that peer at the median point, thus halving its load.

Dynamic Load Balancing: The static scheme does not provide dynamic balancing as it can only be executed at peer joins. Thus PRISM also has a dynamic scheme that provides periodic load balancing based on the algorithm proposed by Karger et al. [14]. Each peer P_i periodically contacts another peer P_j at random to see if they should perform a load exchange operation. The operation is initiated if the load ratio of the two peers is below a certain threshold ϵ (if $\frac{L_1}{L_2} \leq \epsilon$, where L_1 and L_2 denotes the smaller and larger of the loads of the involved peers, respectively). The load of a peer is measured as the number of indices stored in this case. To exchange load, the peer with less load leaves its current location in the DHT and joins at the peer with the higher load.

5.2 Reducing Index Size

The feature vectors can have a very high dimension, as in the case of text documents where a document can contain a large number of keywords. Then the object indices stored and the messages exchanged in the system will be large since they contain the feature vectors of the objects. One way of reducing storage and bandwidth requirements in such a case is to reduce the feature vectors of the objects. For this purpose, the Feature Vector Extractor can apply a dimensionality reduction technique on the original vectors to

⁶An efficient broadcast algorithm for DHTs is presented by El-Ansary et al. [7].

output lower dimensional feature vectors. Such a reduction might also improve retrieval accuracy by removing noise or discovering hidden semantic associations among the objects as in the case of Latent Semantic Indexing of documents [6]. We implemented 4 different dimensionality reduction techniques: Singular Value Decomposition [2], FastMap [8], Random Projection [3], and Fourier Transformation. The effect of these reduction techniques on the retrieval accuracy is explored in Section 6.4.

In order for PRISM to function properly, all peers should use the same reduction technique and map feature vectors to the same reduced space. Some of the reduction methods might require global knowledge about the object set, such as SVD and FastMap. In this case, a certain node (the initiating peer or an external monitor) samples the objects shared in the system and applies the reduction on the sample set. It then distributes the global information to other peers. For example, in the case of FastMap, a set of pivots are selected from the sample and then distributed to all peers, so that they can map the feature vectors to the same space using the pivots. Note that this is a one time operation and performed at initiation time. However the quality of the reduction might decrease as a result of the changes in the object set. Thus the monitor node might keep sampling the objects and updating the global information periodically.

6. EXPERIMENTS

We have evaluated the performance of PRISM using a simulator implemented in Java. All experiments presented here were run on a machine with 2GHz CPU and 1.5GB memory running Linux RedHat 8.0.

Data Set

We have used an image data set for our experiments. The data set consists of 80 dimensional Edge Histogram Descriptor (EHD) features, as proposed by MPEG-7 standard, for more than 3 million JPEG images collected from the Web as part of the Cortina project [18]. Every 100th image vector from the first 10000 vectors is selected as a query, thus giving a set of 100 queries.

Evaluation Metrics

In the experiments, we measured the retrieval accuracy, the number of peers contacted for a query, and the distribution of indices among the peers. Our main focus was to determine the performance of PRISM for k-NN queries, i.e., efficiency in terms of retrieving a fixed number, k , of objects that are most similar to given queries. We compared the performance of PRISM against a centralized setting, where all the objects are examined to compute the answer. For each query, the top k similar objects obtained by scanning the whole data set were considered as the actual results (set A). Then the objects are distributed using PRISM and k objects are retrieved (set B). The recall rate for a query is the percentage of the actual results included in B and computed as: $Recall = \frac{|A \cap B|}{|A|} \times 100\%$. We set k to 10, which is a reasonable value since most users only view the top 10 results when searching the Web [26]. Thus each lookup request for a query returned the top 10 local results from the receiving peers and these results were merged at the querying peer to compute the final result. Note that in this case, since we are interested in the top k objects and $|A| = |B|$, precision and recall values are the same. The effect of k on the recall rate is measured in Section 6.1.

Selecting Reference Pairs

The reference pairs for publishing indices and looking up queries were selected from the top 5 reference vectors of the sorted reference list. We experimentally determined the following ordered list of 21 possible reference pairs for publishing objects⁷: $\{1,1\}$, $\{1,2\}$, $\{2,3\}$, $\{3,2\}$, $\{2,1\}$, $\{4,3\}$, $\{5,2\}$, $\{1,3\}$, $\{1,4\}$, $\{1,5\}$, $\{2,4\}$, $\{2,5\}$, $\{3,4\}$, $\{3,5\}$, $\{3,1\}$, $\{4,2\}$, $\{4,5\}$, $\{4,1\}$, $\{5,3\}$, $\{5,4\}$, $\{5,1\}$. Thus the reference pairs for publishing an object are determined according to this order based on the value of *indexPerm*. For example, if *indexPerm* was 3 for object O from Figure 4, it would be published for the reference pairs (R_6, R_6) , (R_6, R_4) , and (R_4, R_0) . The identity pair, $\{1,1\}$, is used in order to utilize the whole Chord ring, however it is used only once to avoid the overloading of the peers responsible for these pairs.

Setting *indexPerm* to a big value is usually affordable because it only requires limited additional storage for storing the replicas of the index. However, increasing *queryPerm* is more costly as it increases the number of visited peers and the bandwidth consumed for processing a query. To reduce the number of query reference pairs, if the pair $\{i,j\}$ is used, then its inverse, $\{j,i\}$, is ignored. Note that since the object indices are likely to be published for both pairs, this will not have any significant impact on the results. Hence the 21 possible pairs reduce to 11: $\{1,1\}$, $\{1,2\}$, $\{2,3\}$, $\{1,3\}$, $\{1,4\}$, $\{2,5\}$, $\{2,4\}$, $\{3,4\}$, $\{1,5\}$, $\{4,5\}$, $\{3,5\}$.

Experimental Settings

For a run with a certain number of objects and peers, the objects were evenly assigned to peers and the peers joined an initially empty system. The queries were issued after all peers joined the system and initiated at randomly chosen peers. The reported results are the average values over all the queries. The values of *indexPerm* and *queryPerm* remained the same for all objects and queries respectively during a run. The Euclidean distance is used as the similarity measure. Both load balancing schemes discussed in Section 5.1 were implemented. For static load balancing, a new peer contacted 6 peers while joining the system and then joined the one with the highest load in terms of the number of indices stored. For dynamic load balancing, peers exchanged load if the ratio of their load was less than 1/4. The dynamic algorithm was executed 8 times by each peer after all the peers joined. Unless stated otherwise, a system with 1000 peers, 100000 objects, and 32 randomly selected reference vectors were used as the default scenario.

6.1 Recall Rates

Figure 5 shows the corresponding recall rates for different values of *indexPerm* (1 to 21) and *queryPerm* (1 to 11) for the default scenario. The recall rate starts out at 55.0% for *indexPerm*=1 and *queryPerm*=1, and improves as *indexPerm* and *queryPerm* increases, going up to 99.4% for the case with *indexPerm*=21 and *queryPerm*=11. Thus PRISM can retrieve almost all answers when it is configured for high accuracy. The recall improvements gained by increasing the values of *queryPerm* and *indexPerm* diminish as their values increase. Most of the correct results are returned by the first few query lookups and the improvement in the recall rates becomes smaller as *queryPerm* further increases. A similar observation can be made for *indexPerm*

⁷We ran a set of queries over a randomly selected data set of 10K objects, and sorted the reference pairs based on the number of correct top-10 results they returned.

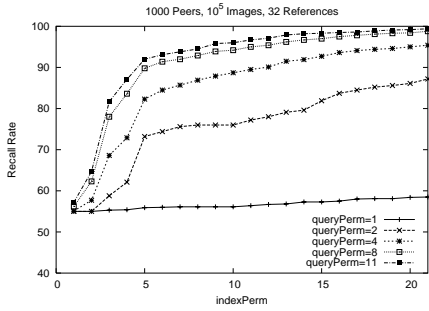


Figure 5: Recall Rates

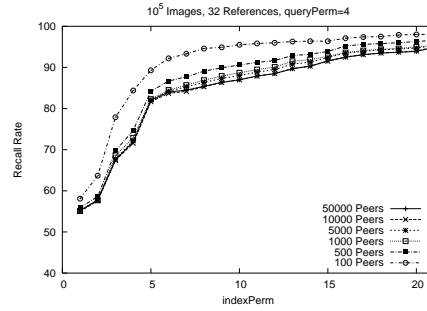


Figure 6: Recall vs. Peer No

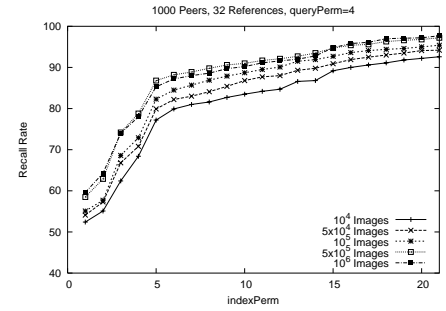


Figure 7: Recall vs. Object No

too. Setting the value of *queryPerm* between 4-8 results in good recall. When *queryPerm*=8 and *indexPerm*=15, for example, the recall rate is 97.0%.

When *queryPerm* is 1 in Figure 5, the recall rate remains almost the same as *indexPerm* increases. That is because the query only goes to the peer corresponding to the identity pair, and thus increasing *indexPerm*, i.e., publishing object indices for reference pairs other than the identity pair, does not improve the result. The reason for the slight increase in recall is that some of the peers that are queried might have matching indices if they are responsible for Chord keys falling in more than one key interval.

Number of Peers

Figure 6 shows the recall rates as the number of peers changes. The recall rates degrade as the number of peers in the system increases, however the amount of degradation is relatively small and the system is able provide over 90% accuracy even with very large number of peers. The recall rates are very high for 100 peers because in this case a peer usually spans over multiple key intervals (since the number of peers is much less than the number of key intervals) and so it also scans over the indices mapped to those intervals when it answers a query destined for one of its intervals.

Number of Objects

Figure 7 shows that the recall rates increase slightly as the number of objects shared in the system increases. When there are more objects in the system, the chance of having objects that are very similar to the query increases, and thus PRISM is able to locate these similar objects using its reference based index location scheme, resulting in higher retrieval accuracy. The above results suggest that PRISM can sustain high levels of recall even when there are big increases in system and data set size.

Number of Reference Vectors

We measured the effect of the number of reference vectors on the recall performance. As shown in Figure 8, the recall rates are very dependent on the number of reference vectors. A larger set of reference vectors distributes the objects into a larger set of possible intervals, thus making it harder to retrieve the results. The results are still good as the recall rates quickly go over 60% and reaches 85% even for the worse case where there are 128 references. Although these results suggest that having a smaller set of references is desirable, it results in higher querying cost due to larger reference pair intervals as will be shown in Section 6.2.

Reference Vector Selection

We evaluated the effect of the reference vector selection method on the recall performance. For reference vector selection, we used three methods as described in Section 4.3.2: 1) Selecting randomly, 2) Clustering the sample set and

using the cluster representatives as references (we used k-Means clustering), and 3) Partitioning the sample space and using the partition centers as references (we used kd-Tree algorithm). Figure 9 shows the results. The partition selection is less accurate than the others because it selects more reference vectors from the clustered regions and so it is harder to distinguish those objects using reference vectors. On the other hand, the cluster selection picks one reference from each clustered region thus increasing the effectiveness of the reference-based distribution.

Number of Results

Figure 10 depicts the effect of the number of returned results, *k*, on the retrieval accuracy. The recall rates drop as *k* increases because some of the results might not be very similar to the query and can be missed. However, even for *k*=500, the recall rates are high and go up to 86%. Also, the number of hops visited for a query is independent of *k*, and remained almost the same in all cases.

6.2 Number of Peers Visited

Table 1 shows the average number of peers contacted to answer a query for different values of *queryPerm* for 100, 1000, and 10000 peers. The routing path length is the number of peers visited during the routing of lookup requests from the querying peer to the destination peers, whereas the forwarding path length is the number of peers that are forwarded the requests within corresponding key intervals. Both path lengths increase linearly with *queryPerm* as expected. The routing scales well in terms of the number of peers. A single lookup is routed via 4.69 peers (0.47%) in a 1000 peer system, whereas it visits 6.42 peers (0.064%) when the peer number is increased to 10000. However, the forwarding cost might increase quickly if the number of reference vectors is not set properly with respect to the number of peers. In the case of 10000 peers for example, the number of key intervals is $32 \times 32 = 1024$, which is much smaller than the number of peers. Hence there are multiple peers in each key interval to which the lookup requests should be forwarded, causing an excessive number of peer visits. It is a good practice to set the number of reference vectors in such a way that the number of reference pair key intervals is around or more than the number of peers. So, if the maximum number of peers that can be in the system (call it *Max*) can be estimated, then the number of reference vectors can be set to a value approximately \sqrt{Max} . Since the key intervals will have a single peer on the average, this ensures that the requests are not forwarded to many peers and the forwarding cost remains reasonable. In this case, the number of peers visited is expected to increase logarithmically with the number of peers.

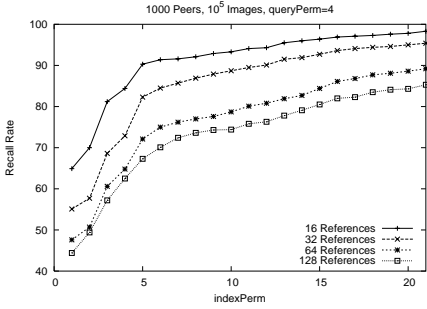


Figure 8: Recall vs. Ref. No

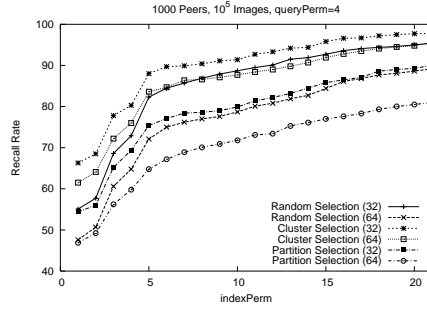


Figure 9: Recall vs. Ref. Selection

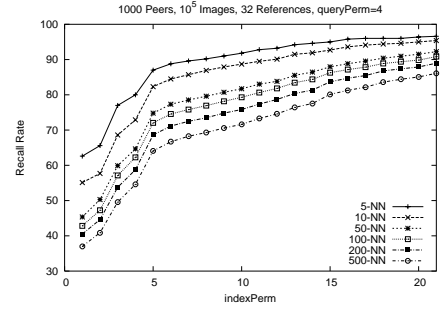


Figure 10: Recall vs. Result No

Table 1: Average number of peers visited to answer a query (32 References, $indexPerm=21$)

query Perm	100 peers		1000 peers		10000 peers	
	Routing	Forwarding	Routing	Forwarding	Routing	Forwarding
1	3.02	0.45	4.69	2.45	6.42	25.18
2	6.31	0.97	9.64	6.76	13.14	67.61
4	12.88	1.73	19.70	14.35	26.32	144.20
8	25.87	2.95	39.17	27.30	51.97	270.61
11	35.21	3.78	53.95	35.54	71.61	354.19

6.3 Load Balancing

In this section, the effectiveness of the load balancing schemes implemented by PRISM are investigated. Figure 11 shows the index distribution for different cases. The peers in the system are sorted in decreasing order of the number of indices they are assigned. The X axis shows the percentage of peers, and the corresponding values are the percentage of indices assigned to these peers. Without load balancing, the index distribution is very skewed and 20% of the peers store around 60% of the indices. Both load balancing schemes greatly improve the index distribution. The dynamic scheme appears to be more effective, but the static scheme can help reduce the number of load exchange operations performed by the dynamic scheme. The load exchange threshold, ϵ , of the dynamic scheme provides great flexibility as it can be adjusted to obtain the desired level of index distribution balance.

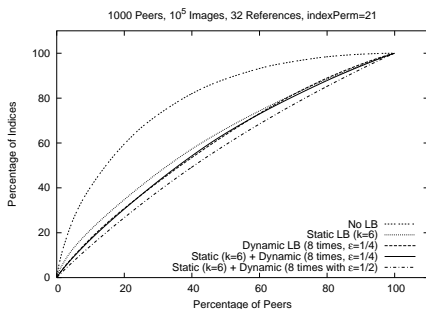


Figure 11: Effect of the load balancing (LB) techniques on the index distribution

6.4 Accuracy of Reduction Techniques

We evaluated the recall performance after implementing the dimensionality reduction techniques discussed in Section 5.2. The original 80 dimensional feature vectors were reduced to 32 dimensions using each technique and then the reduced vectors were used in PRISM. Figure 12 shows the percentage of the original results that were retrieved. We

also measured how many of the original 10 nearest neighbors were included within the 100 nearest neighbors in the reduced space. The results were 88.8%, 86.5%, 51.0%, and 38.6% for Single Value Decomposition (SVD), FastMap (FM), Random Projections (RP), and Fourier Transformation (DFT) respectively. In comparison to information loss due to reduction, the loss due to distribution (PRISM) is significantly small. The results suggest that as long as the reduction step is accurate, PRISM will continue to return accurate answers.

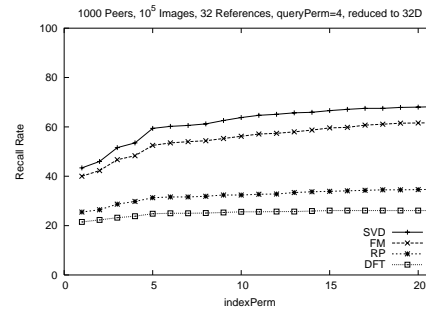


Figure 12: Recall rates after the feature vectors are reduced to 32D using 4 different techniques

6.5 Discussion

The PRISM architecture demonstrated efficient and scalable retrieval performance in the experiments. The results suggest that setting the value of $queryPerm$ between 4-8 and $indexPerm$ between 10-21 returns high recall rates. For example, in a system with 1000 peers with $queryPerm=4$ and $indexPerm=12$, 90.1% recall can be achieved by visiting only 34.05 peers (3.4%). The recall rate goes up to 98.8% by visiting 66.47 peers (6.7%) when $indexPerm=21$ and $queryPerm=8$. Thus PRISM can be configured for high recall if storage and communication overheads are tolerable.

The storage and bandwidth requirements of PRISM are reasonable. Assume that an object index contains a 100-dimensional vector (4x100B) and some additional information (~100B). Then the size of an index is around 0.5KB. The amount of storage needed to store 10000 indices is

only 5MB. Thus it is usually affordable to have a greater *indexPerm*. Publishing an object with *indexPerm*=21 takes a total of 10.5KB storage. The data transferred to process a single query lookup, *L*, can be computed as: $L = p \cdot Q + k \cdot I$, where *p* is the average number of peers visited for the lookup, *Q* is the query message size, *k* is the number of results returned, and *I* is the size of an index. The total amount of data transferred to process a query, *B*, is:

$$B = \text{queryPerm} \cdot L = \text{queryPerm} \cdot (p \cdot Q + k \cdot I).$$

Both *Q* and *I* are 0.5KB. *p* is independent of the number of objects, query size, and object size, and increases logarithmically with the number of peers. Using the data from Table 1 for the 10000 peer system, the total amount of data *D* transferred to answer a query with *queryPerm*=11 is:

$$D = 11 \cdot [(6.42 + 25.18) \cdot 0.5 + 15 \cdot 0.5] \approx 256\text{KB}.$$

The number of peers visited and thus the data transferred for processing a query are independent of the number of objects, query size, and object size, and depend only on the number of peers, number of reference vectors and *queryPerm*. The increase in the number of visited peers is logarithmic in terms of the number of peers (given that the number of reference vectors is selected properly), and linear in terms of *queryPerm*.

PRISM returns the *k* most similar documents to a query. In this setting of top-*k* document retrieval, recall and precision correspond to the same value. In the experiments, we compared the results returned by PRISM with those of a centralized setting to determine the accuracy of the system. Experimental results show that PRISM is accurate and scalable. It delivers high recall by visiting a small number of peers. In a 1000 peer system with 100000 objects and 32 reference vectors, 87.2% and 98.8% recall can be achieved by visiting 16.40 peers (1.64%) and 66.47 peers (6.7%), respectively when *indexPerm*=21. By carefully tuning the parameters, PRISM can be configured for very high accuracy at the expense of storage and communication overhead.

7. CONCLUSION

In this work, we introduced PRISM, a novel scheme based on reference vectors for distributing and querying high dimensional data in P2P networks. PRISM is able to support a larger set of query operations than current systems as it distributes the index information of objects based on their feature vectors. In addition to supporting exact key lookups and keyword searches, it can return accurate approximate answers to similarity search queries by visiting only a few number of participating peers. It also allows users to tune system parameters for higher accuracy at the expense of storage and communication overhead. Due to the underlying DHT architecture, PRISM is scalable, dynamic, and fault tolerant.

As future work, we plan to investigate efficient ways of dynamically adjusting the reference vectors and evaluate the system with various other data sets. We also plan to analyze how PRISM compares to other schemes.

8. REFERENCES

- [1] M. Bawa, G. S. Manku, and P. Raghavan. Sets: search enhanced by topic segmentation. In *SIGIR*, pages 306–313, 2003.
- [2] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.
- [3] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *SIGKDD*, pages 245–250, 2001.
- [4] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS*, pages 23–32, 2002.
- [5] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. Planetp: Using gossiping to build content addressable peer-to-peer information sharing communities. In *HPDC-12*, page 236, 2003.
- [6] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.
- [7] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi. Efficient broadcast in structured p2p networks. In *IPTPS*, pages 304–314, 2003.
- [8] C. Faloutsos and K.-I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD*, pages 163–174, 1995.
- [9] O. D. Gnowali. A keyword-set search system for peer-to-peer networks. Master’s thesis, Massachusetts Institute of Technology, 2002.
- [10] Gnutella. <http://www.gnutella.com/>.
- [11] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *IPTPS*, pages 160–169, 2003.
- [12] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *IPTPS*, pages 242–259, 2002.
- [13] P. Kalnis, W. Ng, B. Ooi, and K. Tan. Answering similarity queries in peer-to-peer networks. *Information Systems*, 1(1):1–1, 2005.
- [14] D. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA*, pages 36–43, 2004.
- [15] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *IPTPS*, pages 207–215, 2003.
- [16] W. Muller and A. Henrich. Fast retrieval of high-dimensional feature vectors in p2p networks using compact peer data summaries. In *MIR*, pages 79–86, 2003.
- [17] Napster. <http://www.napster.com/>.
- [18] T. Quack, U. Monich, L. Thiele, and B. Manjunath. Cortina: A system for large-scale, content-based web image retrieval. In *Multimedia*, 2004.
- [19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.
- [20] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Middleware*, pages 21–40, 2003.
- [21] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.
- [22] O. D. Sahin, F. Emekci, D. Agrawal, and A. E. Abbadi. Content-based similarity search over peer-to-peer systems. In *DBISP2P*, pages 46–63, 2004.
- [23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.
- [24] T. Suel, C. Mathur, J.-W. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In *WebDB*, pages 67–72, 2003.
- [25] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *NSDI*, pages 211–224, 2004.
- [26] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM*, pages 175–186, 2003.
- [27] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *ICDCS*, pages 5–14, 2002.
- [28] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.
- [29] Y. Zhu, H. Wang, and Y. Hu. Integrating semantics-based access mechanisms with peer-to-peer file systems. In *P2P*, pages 118–125, 2003.