Integrating Semantics-Based Access Mechanisms with P2P File Systems

Yingwu Zhu
Department of ECECS
University of Cincinnati
Cincinnati, OH 45221, USA
zhuy@ececs.uc.edu

Honghao Wang
Department of ECECS
University of Cincinnati
Cincinnati, OH 45221, USA
wanghh@ececs.uc.edu

Yiming Hu
Department of ECECS
University of Cincinnati
Cincinnati, OH 45221, USA
yhu@ececs.uc.edu

Abstract

We present an architecture for a peer-to-peer (P2P) file system which supports semantics-based access. Central to this work is to provide semantic indexing and retrieval capabilities. Our semantic indexing and locating approach is based on distributed hash tables (DHTs) where the indices of semantically close files are clustered to the same peers with high probability (nearly 100%) by the use of locality sensitive hash functions. A query for finding semantically close files can be answered by consulting only a small number of peer nodes which are most responsible for such a query, instead of by query flooding. Our approach only adds index information to peer nodes, thus imposing only a small storage overhead. This paper constitutes an initial step to integrate semantics-based access mechanisms into a P2P file system.

1. Introduction

Human brains normally remember objects based on their *contents* or *features*, while the hierarchical structure of existing file systems solely supports name-based access. To bridge the gap between the human memory and the simple hierarchical namespace, *semantic file systems* such as SFS [5] and HAC [6] have been designed to support content-based access to file objects in addition to name-based access. They allow users to organize their files by content and present them with alternative views of data through the concept of *semantic directories*. In such systems, semantics-based access is provided by *queries*, and queries are mapped into semantic directories, each pointing to files that satisfy a query.

The work reported in this paper differs from that of SFS [5] and HAC [6], in that we strive to support semantics-based access over P2P file systems. Recent P2P file systems such as CFS [4] and PAST [17] are layered on top of a Distributed Hash Table (DHT) [19, 16, 14]. Table 1 summarizes the software layering of a P2P file system. In these systems, each file is assigned a unique identifier called *fileID* as a DHT key, e.g., produced by SHA-1 hash of its content or name. The capabilities of storing and retrieving a file are provided by their DHTs. In spite of significant implementation differences between these DHTs, they all implement a hash-table interface of *put(fileID, file)*, which stores the file in the DHT by mapping from the *fileID* to a peer node, and *get(fileID)* which retrieves the file corresponding to the *fileID*. However, DHTs support only *exact-match* lookups. They do not directly support text search or content search. This is fine for the simple hierarchical namespace of these P2P file systems, from the name-based access perspective. But for a P2P file system targeting for supporting not only name-based access but also semantics-based access, it is far from enough.

Layer	*Responsibility*
FS	Stores/retrieves file objects into/from the DHT; presents a file system interface to applications /users
DHT	Supports a hash-table interface of *get(*$fileID$*)* and *put(*$fileID$*,file)*

Table 1. Software layering in a P2P file system

One of the biggest challenges to current P2P file systems is to provide *convenient* access to vast amount of information. By "convenience" we mean not only the ability to quickly transfer information from one place to another, but the ability to find the "right" information and deal with it [6]. We therefore argue that semantics-based access mechanisms should be integrated into a P2P file system itself. This not only provides semantics-based retrieval capabilities to locate semantically close files, but also allows users to cluster semantically close files through semantic directories for the purposes of browsing relevant materials and purging, if the files stored in the system are already indexed

Proceedings of the Third International Conference on Peer-to-Peer Computing (P2P'03)
0-7695-2023-5/03 $17.00 © 2003 IEEE

according to their semantics or content.

What kind of application can be addressed by such a P2P file system which provides semantics-based access capabilities? To illustrate our vision, we describe an imaginary scenario below.

Bob, a P2P file system user, wants to locate those files semantically close to a file f among enormous documents stored in the system. He submits a query "locate files similar to f" to the system and then waits for the query result. If the query result is not empty, he now can browse all these semantically close files related to f. If Bob has proper permissions on these files, he can even modify or remove these files.

Some time later, if another user Alice submits the same query "locate files similar to f" to the system, she can get all those semantically close files from the previous query submitted by Bob, thus saving the expensive cost of query processing. Note that the system automatically generates materialized views of query results and reuses previous query results when possible.

Leveraging the state-of-the-art information retrieval (IR) algorithms such as vector space model (VSM) and latent semantic indexing (LSI) [1], files and queries can be represented as *semantic vectors* (SVs). In the above example, the query "locate files similar to f" can be represented as a SV derived from f by using VSM or LSI. All the files retrieved and the query have similar SVs — that is, semantically closes files are considered to have similar SVs. Further, as described above, semantic searches in our system are expressed in natural language, instead of *simple keyword match*.

In this paper, we focus on exploring the issue of how to integrate semantics-based access mechanisms into a P2P file system. Central to this work is to provide semantic indexing and retrieval capabilities. In a centralized system the SVs of all files are at one location, and the problem of finding semantically close files can be solved by building an index over all the files. Such an approach, however, results in a single point of failure and a performance bottleneck at the indexing server. In a P2P system, all files are distributed across peer nodes, the problem of finding semantically close files becomes more complicated because we need to locate peers which are responsible for those files without global knowledge. We can simply flood the query over a P2P network, but such a query-flooding approach consumes huge amount of network bandwidth and processing power, posing a scalability issue.

Hence we address this problem by trying to develop techniques that use semantic information (i.e., SVs) to index files (without altering their physical locations) and retrieve files. Our semantic indexing and locating approach is based on DHTs where the *indices* of semantically close files are clustered to the same peers with high probability (nearly 100%) by the use of locality sensitive hash functions (LSH) [9, 11]. A query for finding semantically close files can be answered by consulting only a small number (e.g., 20) of peer nodes which are most responsible for such a query, instead of by query flooding. Furthermore, The query results are also indexed in DHTs according to their corresponding SVs by using LSH, thereby allowing the query results to be reused even without knowing their corresponding semantic directories.

Our approach is based on LSH, and thus provides approximate answers to queries. The main motivation for this is that approximate answers are normally expected because from user's perspectives "semantically-close" is not a precise concept anyway. The second motivation is that the query is just a quick first step to obtain more or less the information users are looking for and that the result of a query can be automatically or manually modified and refined later [6]. The third motivation is that few query systems are perfect, and currently most are not even close, in terms of either partial results or expensive cost of querying processing. The last motivation for our approach is that P2P users often ask broad queries even when they are only interested in a few results and therefore do not expect perfect answers [8]. This paper exhibits an initial step to integrate semantics-based access mechanisms into P2P file systems.

The remainder of the paper is organized as follows. Section 2 gives an overview of related work. Section 3 presents a system architecture for supporting semantics-based access, and Section 4 discusses our semantic indexing and locating approach. Section 5 describes our experimental results. We finally conclude on Section 6.

2. Related Work

Recently, a new generation of P2P systems [19, 16, 14], offering distributed hash table (DHT) functionality, have been proposed. In spite of significant implementation differences, these systems all implement a hash-table interface of *put(key,object)*, which stores the object with the key in the DHT, and *get(key)* which retrieves the object corresponding to the key. A number of recent P2P file systems [4, 17, 10] are designed by layering file system functionalities on top of DHTs, while the capabilities of storing and retrieving an object are provided by their underlying DHTs. However, all these DHTs support only *exact-match* lookups. Therefore these file systems currently are unable to provide semantics-based access capabilities.

There have been recent proposals for P2P text search [8, 12, 20, 15] over DHTs. Harren et al. [8] propose traditional relational database operators on top of DHTs to resolve queries. Reynolds et al. [15] discuss a search infrastructure using distributed inverted indexing, which is similar to our "naive" approach described in [21]. Mahalingam

et al. [12, 20] have proposed to integrate semantic storage and retrieval capabilities into a file system based on CAN [14], where a document index is stored by using its vector representation as the coordinates as a result of vector-CAN Cartesian space transformation. However, our approach to supporting semantics-based access is different, which is based on locality sensitive hash functions (LSH) [9, 11] and can be applied to all DHTs [19, 16, 14].

Semantic file systems such as SFS [5] and HAC [6] have been proposed to support both name-based and content-based access to file objects, allowing users to organize their files by content and presenting them with alternative views of data through the concept of semantic directories. However, these systems are based on traditional file systems. Therefore, the techniques used in these systems for supporting content-based access cannot be simply adopted in a P2P file system to support semantics-based access. Further, SFS and HAC provide support only for simple keyword-based queries while in our system queries are expressed in natural language instead of simple keyword match.

The research work on Information Retrieval (IR) algorithms such as the Vector Space Model (VSM) and Latent Semantic Indexing (LSI) [1] is orthogonal to our study. Our system can employ these techniques to represent files and queries as semantic vectors on which our semantic indexing and locating approach relies.

Linial et al. [11] first showed the existence of locality sensitive hash functions. And Indyk et al. [9] introduced LSH for the nearest neighbor problem. Recently, Gupta et al. [7] have proposed to use LSH to locate data partitions of relations relevant to a SQL query in a P2P system that shares data in the form of database relations. Our system instead strives to support semantics-based access in P2P file systems by leveraging the state-of-the-art IR algorithms and using LSH.

3. System Architecture

Our design starts with a P2P file system and extends it to support semantics-based access. Figure 1 illustrates the architecture of our design. In the rest of the paper, we use the terms *query-based*, *content-based* and *semantics-based* interchangeably.

In order to support semantics-based access, we have to add two major components into an existing P2P file system: *a registry of semantic extractors*, and *semantic indexing and locating utility*. We describe these two components briefly in the following subsections due to space constraints. Please see [21] for more detail.

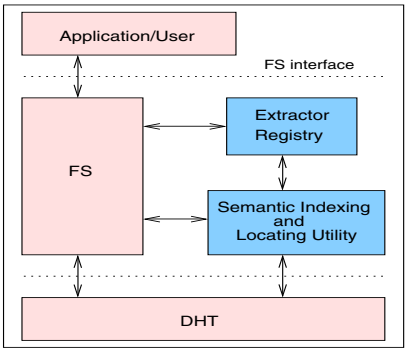

Figure 1. Major components of the system architecture.

3.1. Semantic Extractor Registry

The semantic extractor registry consists of a set of semantic extractors for each known file type. A semantic extractor is an external plug-in module. It is a file-type specific filter that takes as input the content of a file and outputs the corresponding semantic vector (SV) of this file.

Leveraging the state-of-the-art IR algorithms such as VSM and LSI, the fundamental functionality of the semantic extractor registry is to represent each *file* and *query* as a semantic vector where each dimension is associated with a distinct keyword. Recent studies show that a small number of keywords (typically 200-300) can characterize a file.

Whenever a user/application on a peer node X wants to store a file f into the system, the semantic extractor registry on X is responsible for deriving a SV for f. The resulting SV, as will discussed later, is then used to produce a small number of $semIDs$ (semantic identifier) as the DHT keys for f. As a result, the file f can be indexed into the DHT according to its semantic presentation, i.e., with the resulting $semIDs$ as the DHT keys.

3.2. Semantic Indexing and Locating Utility

The semantic indexing and locating utility provides semantics-based indexing and retrieval capabilities. The functionality of semantic indexing is to index each file/query automatically according to its semantic vector whenever a file/query is created or modified. The resulting indices are independent of and do not alter the physical locations of file objects in the system (all files reside where they are located), thereby leaving room for the system or users or applications to organize their files in certain way for optimizations (e.g., organize files by access locality for performance considerations). The functionality of semantic locating is to locate semantically close files for a given

query.

The semantic indexing and locating utility also interacts with the file system (FS) to generate materialized views of query results, and allows users to access these materialized views as regular file system objects through semantic directories. Due to space constraints, please refer to [21] for more detail.

4. Semantic Indexing and Locating

In this section, we first introduce the locality sensitive hashing (LSH), and then discuss our LSH-based semantic indexing and locating scheme.

4.1. Locality Sensitive Hashing

From [3, 9], a family of hash functions $\mathcal{F}$ is said to be a locality sensitive hash function family corresponding to similarity function $sim(A, B)$ if for all $h \in \mathcal{F}$ operating on two sets A and B, we have:

$\mathbf{Pr}_{h\in\mathcal{F}}[h(A) = h(B)] = sim(A, B)$.

Where $\mathbf{Pr}$ is the probability, and $sim(A, B) \in [0, 1]$ is some similarity function.

Min-wise independent permutations [2] provide an elegant construction of such a locality sensitive hash function family with the Jaccard set similarity measure $sim(A, B) = \frac{|A\cap B|}{|A\cup B|}$. In our system, both sets A and B represent a semantic vector of a file/query and $sim(A, B)$ represents the similarity of two SVs. Therefore, semantically close files/queries can be defined by $sim(A, B)$. For example, consider two files/queries semantically close with the $sim(A, B)\geq 0.9$.

The hashing scheme of min-wise independent permutations on a semantic vector is as follows. Note that the SV is composed of keywords. We cannot perform permutations directly on it. So we first have to convert the SV into a set of integers produced by SHA-1 hash of each keyword. Let π represent a random permutation on the resulting integer's universe U. Given a semantic vector A, we convert it into $A' = \{a_1, a_2, ..., a_n\} \subseteq U$, which is a set of integers. The hash function h_π is defined as $h_\pi(A') = min\{\pi(A')\} = min\{\pi(a_1), \pi(a_2), ..., \pi(a_n)\}$ (that is, the hash function $h_\pi(A')$ applies the permutation π on each integer component in A' and then takes the minimum of the resulting elements). Then for two semantic vectors A and B, which are first converted into A' and B' respectively, we have $x = h_\pi(A') = h_\pi(B')$ if and only if $\pi^{-1}(x) \in A' \cap B'$. That is, the minimum element after permuting A' and B' matches only when the inverse of the element lies in both A' and B'. In this case, we also have $x = h_\pi(A' \cup B')$. Since π is a random permutation, each integer component in $A' \cup B'$ is equally likely to become the minimum element of $\pi(A' \cup B')$. Hence we conclude that $min\{\pi(A')\} = min\{\pi(B')\}$ (or $h_\pi(A') = h_\pi(B')$) with probability $p' = sim(A', B') = \frac{|A'\cap B'|}{|A'\cup B'|}$. Because the SHA-1 hash function is supposed to be collision-resistant and it would not change the similarity p between A and B after converting A and B into A' and B' respectively, we conclude that $p = sim(A, B) = sim(A', B') = \frac{|A'\cap B'|}{|A'\cup B'|}$. Due to space constraints, we here cannot elaborate min-wise independent permutations. Please see [2] for more detail.

4.2. LSH-Based Semantic Indexing

In this section, we discuss how files/queries are indexed into the underlying DHT according to their SVs by using LSH. The goal of semantic indexing is to cluster the indices of semantically close files/queries to the same peer nodes with high probability. Without loss of generality, our focus here is on files, since queries can also apply the same indexing procedure.

Given a file's semantic vector A, semantic indexing hashes A into a small number of $semIDs$ by using LSH. This process can be described as follows:

1. For each vector component k of A, convert it into a 64-bit integer (since we evaluate our system on the Pastry simulator of 64-bit identifier space, we here convert k into a 64-bit integer) by taking the first 64 bits of k's SHA-1 hash. Therefore, A is converted into A', which is a set of 64-bit integers.

2. Using a group of m min-wise independent permutation hash functions, we derive a 64-bit $semID$ from A'. Therefore, applying n such groups of hash functions on A' can yield n $semIDs$ (as shown in Figure 2).

```
Semantic Indexing Procedure:
(1) convert A into A', which is a set of integers
(2) for each g[j] do        \\ g[j] is one of n groups of hash functions
(3)     semID[j]=0
(4)     for each h[i] in g[j] do        \\ g[j] has m hash functions
(5)         semID[j] ^= h[i](A')        \\ ^ is a XOR operation
(6)     endfor
(7) endfor
(8) for each semID[j] do
(9)     insert the tuple <semID,fileID,A> into DHT by having
        semID[j] as the DHT key
(10) endfor
     end
```

Figure 2. Semantic indexing procedure.

Note that for two SVs A and B, their similarity is $p = sim(A, B) = sim(A', B')$. This is because that the SHA-1

hash function is supposed to be collision-resistant and the above process would not change the similarity p.

Let A denote the SV of a file with a $fileID$, Figure 2 describes a rough sketch of semantic indexing procedure. Note that a $semID$ is produced by XORing m 64-bit integers which are produced by applying a group of m hash functions on A'. Thus, applying n such groups of hash functions on A' yields n $semIDs$. By having the resulting $semIDs$ as the DHT keys, the file is indexed into the DHT in the form of $< semID, fileID, A >$. Such semantic indexing could have the indices of semantically close files hashed to the same peer nodes with probability $\geq 1-(1-p^m)^n$ (see [21] for detailed probability analyses).

4.3. LSH-Based Semantic Locating

We now discuss the issue of how to locate semantically close files that satisfy a query, given the fact that all files in the system are automatically indexed according to their SVs in response to file system mutation operations such as file creation or modification. The goal of semantic locating is to answer a query by consulting only a small number of peer nodes which are most responsible for the query.

Given the semantic indexing scheme described earlier, we show that this goal can be easily achieved. For example, Let A be a query Q's semantic vector. Suppose Q wants to locate those files whose SVs are similar to A (with certain similarity degree). The semantic locating procedure (as shown in Figure 3) produces n $semIDs$ from A for Q using the same set of hash functions (used in the semantic indexing procedure). So if a file f satisfies such a query Q, it will be retrieved by Q with very high probability. Note that the SVs of file f and query Q could be hashed to the same $semIDs$ with high probability (i.e., $1-(1-p^m)^n$). Thus, by having these $semIDs$ as the DHT *keys* in the DHT's interface of *get(key)*, Q is able to retrieve semantically close files from the peer nodes which are responsible for these $semIDs$. n is very small (e.g.,20) in our system, which implies that a query can be answered by consulting only a small number n of peer nodes.

In the Figure 3, upon a request, each destination peer (at most n) locally checks the list of tuples $< semID, fileID, SV >$ and finds the $fileIDs$ such that their associated SVs are similar to the query's SV with certain similarity threshold, and sends the list of $fileIDs$ to the requesting peer. Then, the requesting peer merges the replies from all destination peers, generates a materialized view of the query result and indexes the query according to its SV.

Actually, each destination peer can organize its own tuples in such a way that these tuples are clustered locally according to their SVs by using data clustering techniques

```
Semantic Locating Procedure:
(1) convert A into A', which is a set of integers
(2) for each g[j] do     \\ g[j] is one of n groups of hash functions
(3)     semID[j]=0
(4)     for each h[i] in g[j] do     \\ g[j] has m hash functions
(5)         semID[j]^=h[i](A')       \\ ^ is a XOR operation
(6)     endfor
(7) endfor
(8) for each semID[j] do  \\ request for semantically close files
(9)     send a request to a peer which is the destination of the
        semID[j] in the DHT
(10) endfor
(11) get replies from all the destination peers
(12) merge the fileIDs that satisfy the query from all replies
(13) create a materialized view of the query result asynchronously
(14) index the query according to its SV asynchronously
     end
```

Figure 3. Semantic locating procedure.

such as *hierarchical k-means*. Each destination peer uses hierarchical k-means to cluster the tuples into collections according to the semantic vector until the variance inside a collection falls below certain threshold. Managing the tuples in the unit of collections allows each peer to narrow the search range (within a single collection instead of the whole indices) upon a request, thereby making the search efficient and fast.

It is worth pointing out that we can use a similar semantic locating procedure to locate semantically close queries for reuse. Please refer to [21] for more discussions about query reuse.

5. Experimental Results

In this section we evaluate our semantic indexing and locating approach, in terms of load distribution of the semantic indexing scheme and in terms of the percentage of matched semantically close files obtained by the semantic locating scheme given a set of queries.

Our semantic indexing and locating approach is evaluated based on the Pastry [16]. The identifier space in Pastry is 64-bit. Moreover, as mentioned in Section 4.2, semantically close files are indexed and located in the same peers with high probability by using n groups of m hash functions. In the rest of the paper, the values of n and m are chosen to be 20 and 5 respectively unless otherwise specified. The LSH used in our experiments is min-wise independent permutations.

5.1. Load Distribution of Semantic Indexing

Our first experiment involved a P2P file system that stores 10,000 files, whose semantic vectors are chosen from 10,000 unique keywords in the following way: viewing the 10,000 keywords as a circle space, from which we chose uniformly at random an arc of 200 keywords in length as a semantic vector for each file. We expect that, in this case, many files share similarity with each other. Thus, it probably causes a very skewed index distribution as a result of our semantic indexing approach. We hashed each file's semantic vector into 20 $semIDs$ of 64 bits using min-wise independent permutations, and indexed each file into our system by applying the semantic indexing procedure. Figure 4 (a) shows the mean, 5-percentile and 95-percentile of the number of indices stored per peer node in the system, as the number of peer nodes in the system varies from 100 to 5000. We note that the load distribution of indices including both the 95-percentile and mean drops linearly as the number of peer nodes increases. Figure 4 (b) shows the mean, 5-percentile and 95-percentile of the number of indices stored per peer node in a 1000 node system where the number of indexed files varies from 10,000 to 150,000. The 95-percentile grows sublinearly with the increasing number of indexed files while the mean shows a superlinear increase.

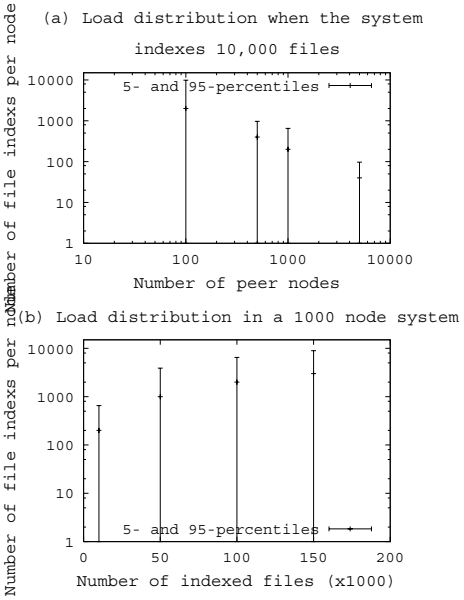

Figure 4. Semantic indices per peer node.

We also conducted our experiments on other two sets of 10,000 files. One is that, each file shares no similarity with any other files. That is, there is no similarity between any two semantic vectors. We named this case "No-Sim". The other is that, we generated 100 *base* files, each of which is represented by a 200-keyword semantic vector. Among these 100 files, there is no similarity. The remaining 9,900 files each share certain similarity with only one of these 100 base files, following a Zipf-like distribution. Namely, the frequency of the remaining 9,900 files sharing similarity with the i^{th} base file is proportional to $\frac{1}{i^{\alpha}}$. In our experiments, α was set to be 0.7. In addition, we determined the similarity value between each of the remaining files and its corresponding base file in the following way: similarity p $\in$ (0,0.3), [0.3,0.6), and [0.6,1.0) with a probability of 20%, 30%, and 50% respectively. We called this case "Zipf-like".

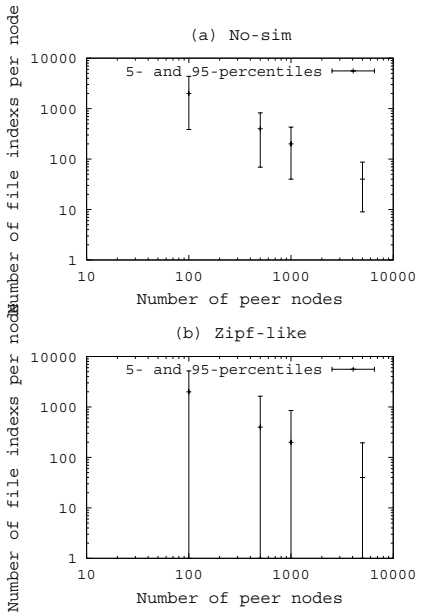

Figure 5. Semantic indices per peer node.

Figure 5 shows the results for both cases as the number of peer nodes in the system varies from 100 to 5000. As expected, No-sim shows a much less skewed load distribution in contrast to Zipf-like as well as the results shown in Figure 4 (a). This is because the semantic indexing approach only intends to cluster the indices of semantically close files into the same peers with high probability (by deriving the same $semIDs$), rather than dissimilar files. We also note that, in Zipf-like, the load distribution of indices including both the 95-percentile and mean drops linearly as the number of peer nodes increases.

We further conducted experiments for both No-sim and Zipf-like in a 1000 node system where the number of indexed files varies from 10,000 to 150,000. The results are similar to those presented in Figure 4 (b), but No-sim shows a much less skewed load distribution.

Discussion. In spite of synthetic workloads used above, we believe that they could give a flavor of how the load dis-

tribution looks like under real-world workloads. Moreover, the semantic indexing scheme adds only index information to nodes in the form of $< semID, fileID, SV >$. The storage overhead is expected to be small. We here offer some back-of-the-envelope numerical support for our expectation. For instance, consider a 1000 node system storing 150,000 files (as shown in Figure 4 (b)). We estimate the storage requirement for the 95-percentile of the number of indices stored per peer node (about 9000 index tuples). Both $semID$ and $fileID$ are 8 bytes, and a SV of 200 keywords is 1,600 bytes (a SV can be represented as a set of 64-bit integers). Therefore the 95-percentile consumes about 14 MB. Since the load distribution is skewed, most nodes in the system contribute less than 14 MB storage space to the indices. Another concern is that, clustering the indices of semantically close files into the same peers might cause a "hot spot" or "flash crowd" problem. We can address this issue by using the *cache diffusion* method suggested in [18].

5.2. Performance of Semantic Locating

The performance of our semantic locating approach is measured in terms of the percentage of semantically close files that can be located given a set of queries.

As mentioned earlier, Leveraging IR algorithms such as VSM and LSI, files and queries are represented as semantic vectors. But, we currently do not have such VSM and LSI tools to derive semantic vectors for files. Hence, we used chunk fingerprints as a semantic vector instead in this experiment. Some recent work [3, 13] have suggested to use fingerprints to identify similar files. For example, a file can be divided into a list of chunks based on its content. Each chunk is identified by a fingerprint which is produced by SHA-1 hash of the chunk. Thus, we can use a list of chunk fingerprints as a SV to represent a file. If two files have similar fingerprint lists, they are considered to be similar — semantically close.

We checked out 205 unique C++ program files from a CVS repository. Each of these 205 program files consists of 3 different versions on average. Hence we checked out 615 files in total (about 10 MB). We divided each single file into a list of variable-sized chunks using the technique suggested in LBFS [13]. As a result, each file can be represented as a list of chunk fingerprints, each of which is a 64-bit integer by taking the first 64 bits of the chunk's SHA-1 hash [13]. We started our experiment with an empty P2P file system simulator of 1000 peer nodes built atop Pastry, and indexed each file into our simulator by applying the semantic indexing procedure. Then we issued a set of queries to locate different versions for each unique C++ program file, since here we consider different versions of a program file semantically close due to similar fingerprints.

Figure 6 (a) shows the results of queries in terms of the

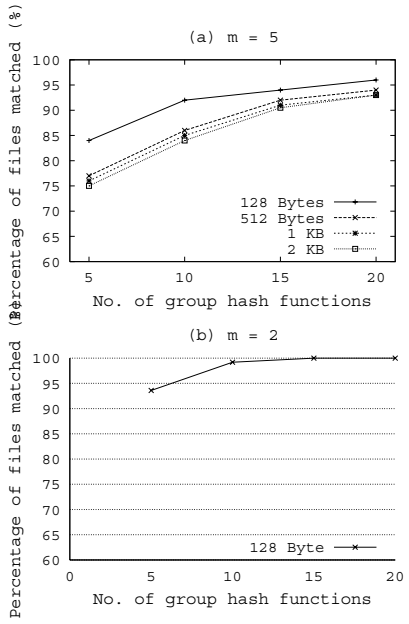

Figure 6. Performance of semantic locating.

percentage of semantically close files found for different minimum chunk size limits [1] including 128 Byte, 512 Byte, 1 KB and 2 KB. The x-axis represents the number of group hash functions n used in the semantic indexing and locating procedures, while the y-axis represents the percentage of files found. As expected, the percentage of files matched increases with the number n increasing from 5 to 20. Moreover, as the minimum chunk size varies from 128 byte to 2 KB, the percentage decreases. This is because chunks with a smaller minimum chunk size limit are able to identify more similarity between different versions of a file. But even when the minimum chunk size limit is 128 Byte and n is 20, our semantic locating was still unable to find all the files. This is because some files are very small, even a minimum chunk size limit of 128 Byte could not make the similarity high enough between different versions of a file. According to $(1-(1-p^m)^n)$, if p is small (say, ≤ 0.7), our locating approach might fail to find *all* semantically close files with such a small similarity (because it cannot guarantee a 100% probability). Further, a small m, as discussed in [21], could dramatically improve the percentage. Figure 6 (b) shows the result for a minimum chunk size limit of 128 Byte with m = 2.

6. Conclusions

In this paper we have explored the issue of how to integrate semantics-based access mechanisms into a P2P file

[1] When dividing a file into chunks, we impose a minimum chunk size limit like in LBFS [13].

system, and shown that it can be implemented efficiently and reasonably clean. Central to our work is to provide semantic indexing and retrieval capabilities. Our semantic indexing and locating approach is based on DHTs where the indices of semantically close files are clustered to the same peers with high probability (nearly 100%) by the use of locality sensitive hash functions. A query for finding semantically close files can be answered by consulting only a small number of peer nodes which are most responsible for such a query, instead of by query flooding. Our approach only adds index information to peer nodes, thus imposing only a small storage overhead. This paper constitutes an initial step to integrate semantics-based access mechanisms into a P2P file system. A number of issues such as query consistency, query dependency and query refinement need to be explored in our future work.

7. Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments on drafts of this paper. We also thank Abhishek Gupta for helpful discussion on LSH.

References

[1] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[2] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[3] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 380–388, 2002.

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Banff, Canada, Oct. 2001.

[5] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*, pages 16–25, Oct. 1991.

[6] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the 3th USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 265–278, 1999.

[7] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[8] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the first International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002.

[9] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 13th annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[10] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, Cambridge, MA, Nov. 2000.

[11] N. Linial and O. Sasson. Non-expansive hashing. In *Proceedings of the 13th annual ACM Symposium on Theory of Computing*, pages 509–518, 1996.

[12] M. Mahalingam, C. Tang, and Z. Xu. Towards a semantic, deep archival file system. In *The 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, 2003.

[13] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 174–187, 2001.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, San Diego, CA, Aug. 2001.

[15] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference(Middleware 2003)*, June 2003.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed System Platforms (Middleware 2001)*, Heidelberg, Germany, Nov. 2001.

[17] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Banff, Canada, Oct. 2001.

[18] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *Proceedings of the first International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002.

[19] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, CA, Aug. 2001.

[20] C. Tang, Z. Xu, and M. Mahalingam. Peersearch: Efficient information retrieval in peer-peer networks. Technical Report HPL-2002-198, Hewlett-Packard Labs, 2002.

[21] Y. Zhu, H. Wang, and Y. Hu. SemPFS: Integrating semantics-based access mechanisms with P2P file systems. Technical Report TR-260/03/03/ECECS, University of Cincinnati, Mar. 2003.