38

Semantic Search in Peer-to-Peer Systems

Yingwu Zhu
Yiming Hu

38.1 Introduction

A recent report[18] has shown that 93 percent of information produced worldwide is in digital form. The volume of data added each year is estimated at more than one terabyte (i.e., 10^{18} bytes) and is expected to grow exponentially. This trend calls for a scalable infrastructure capable of indexing and searching rich content such as HTML, music, and image files.[28]

One solution is to build a search engine such as Google. Such a solution, however, needs to maintain an enormous centralized database about all the online information. Also, for such search engines to appear to be "scalable," they need a very large and expensive infrastructure to support their operations (e.g., Google uses tens of thousands of computers). The costs of hardware and software, as well as maintenance and utilities, are very high.[2] The centralized database approach also poses a single point of failure problem. Moreover, newly created or modified information often is not indexed into the search database for weeks. Similarly, search results often contain stalled links to files that have been removed recently.

On the other hand, as P2P (peer-to-peer) systems gain more interest from both the user and research communities, building a search system on top of P2P networks is becoming an attractive and promising alternative for the following reasons:

- *High availability*. Centralized search systems are vulnerable to distributed denial-of-service attacks. However, P2P search tends to be more robust than centralized search as the demise of a single node or some nodes is unlikely to paralyze the entire search system. Furthermore, it is not easy for an attacker to bring down a significant fraction of geographically distributed P2P nodes. Recent work[9] has shown that the failure of a reasonable portion of P2P nodes will not prevent a P2P system from functioning as a whole.
- *Low cost and easy of deployment*. As discussed above, a centralized search engine requires a huge amount of investment in both hardware and software, as well as in maintenance. A P2P search system, however, is virtually free by pooling together slack resources in P2P nodes and can be deployed incrementally as new nodes join the system.
- *Data freshness*. In centralized search systems, it usually takes weeks for newly updated data to enter the data center that hosts the search database, due to the fact that it takes time for robots or crawlers to collect such information into the search database as well as the bandwidth constraints between the data center and the Internet. Therefore, there is no freshness guarantee on the index maintained in the centralized database (i.e., weeks delay). On the other hand, P2P nodes can publish their documents immediately once they appear, and the publishing traffic goes to geographically distributed nodes, thereby avoiding the bandwidth constraints imposed by centralized search systems.
- *Good scalability*. A recent study[13] has shown that no search engine indexed more than 16% of the indexable Web. The exponentially growing data added each year would be beyond the capability of any search engine. However, the self-organizing and scalable nature of P2P systems raises a hope to build a search engine with very good scalability.

The purpose of this chapter is to give an overview of P2P search techniques and present two semantic search systems built on top of P2P networks. The remainder of this chapter is structured as follows. We review the search systems built on top of unstructured P2P networks in Section 38.2. Section 38.3 provides a survey of the search systems built on top of structured P2P networks. Section 38.4 provides the necessary background. We present two representative search systems in Sections 38.5 and 38.6, respectively. Finally, we conclude with Section 38.7.

38.2 Search in Unstructured P2P Systems

In unstructured P2P systems such as Gnutella, the unstructured overlay organizes nodes into a random graph and uses flooding on the graph to retrieve relevant documents for a query. Given a query, each visited node evaluates the query locally on its own content and then forwards the query to all its neighbors.

Arbitrarily complex queries therefore can be easily supported on such systems. Although this approach is simple and robust, it has the drawback of the enormous cost of flooding the network every time a query is issued.

Improvements to Gnutella's flooding mechanism have been studied along three dimensions: *random walks*, *guided search*, and similar content group-based search.

38.2.1 Random Walks

Random walks represent the recommended search technique proposed by Lv et al.[15] It is used to address the scalability issue posed by flooding on unstructured P2P systems like Gnutella. Given a query, a random walk is essentially a blind search in that at each step, the query is forwarded to a randomly chosen node without considering any hint of how likely the next node will have answers for the query. Two techniques have been proposed to terminate random walks: TTL (time-to-live) and "checking." TTL means that, similar to flooding, each random walk terminates after a certain number of hops, while "checking" means a walker periodically checks with the query originator before walking to the next node.[15]

38.2.2 Guided Search

Guided search represents the search techniques that allow nodes to forward queries to neighbors that are more likely to have answers, rather than forward queries to randomly chosen neighbors or flood the network by forwarding queries to all neighbors.[8]

Crespo and Garcia-Molina[8] introduced the concept of *routing indices* (RIs), which give a promising "direction" toward the answers for queries. They present three RI schemes: the compound, the hop-count, and the exponentially routing indices. The basic idea behind guided search is that a distributed index mechanism maintains indices at each node. These distributed indices are small (i.e., compact summary) and they give a "direction" toward the document, rather than its actual location. Given a query, the RI allows a node to select the "best" neighbors to which to send a query. An RI is a data structure (and associated algorithms) that, given a query, returns a list of neighbors, ranked according to their *goodness* for the query. The notion of goodness generally reflects the number of relevant documents in "nearby" nodes.

38.2.3 Similar Content Group-Based Search

The basic idea of similar content group-based search[3,7,25,31] is to organize P2P nodes into similar content groups on top of unstructured P2P systems such as Gnutella. The intuition behind this search technique is that nodes within a group tend to be relevant to the same queries. As a result, this search technique will guide the queries to nodes that are more likely to have answers to the queries, thereby avoiding a significant amount of flooding.

The search in SETS[3] uses a topic-driven query routing protocol on a topic-segmented overlay built from Gnutella-like P2P systems. The topic-segmented overlay is constructed by performing node clustering* at a single designated node, and each cluster corresponds to a topic segment. Therefore, SETS partitions nodes into topic segments such that nodes with similar documents belong to the same segment. Given a query, SETS first computes R topic segments that are most relevant to the query and then routes the query to these segments for relevant documents. However, the designated node is potentially a single point of failure and performance bottleneck.

Motivated by research in data mining, Cohen et al.[7] introduced the concept of *associative overlays* into Gnutella-like P2P systems. They use *guide rules* to organize nodes satisfying some predicates into associative overlays, and each guide rule constitutes an associative overlay. A guide rule is a set of nodes that satisfies some predicate; each node can participate in a number of guide rules; and for each guide rule it participates

*The node is represented by a node vector that summarizes a node's documents.

in, it maintains a small list of other nodes belonging to the same guide rule. The key idea of guide rules is that nodes belonging to some guide rule contain similar data items. As a result, guided search restricts the propagation of queries to be within some specified guide rules, that is, some associative overlays, instead of flooding or blind search.

Sripanidkulchai et al.[25] propose a content location approach in which nodes are organized into an interest-based overlay on top of Gnutella by following the principle of *interest-based locality*. The principle of interest-based locality is that if a node has a piece of content in which one is interested, then it is likely that it will have other pieces of content in which one is also interested. Therefore, nodes that share similar interests create shortcuts to one another, and interest-based shortcuts form the interest-based overlay on top of Gnutella's unstructured overlay. Given a query, the interest-based overlay serves as a performance enhanced layer by forwarding the query along shortcuts. When shortcuts fail, nodes resort to the underlying Gnutella overlay.

ESS[31] is another example of efficient search on Gnutella-like P2P systems, by leveraging the state-of-the-art information retrieval (IR) algorithms. The key idea is that ESS employs a distributed, content-based, and capacity-aware topology adaptation algorithm to organize nodes into semantic groups. Thereby, nodes with similar content belong to the same semantic group. Given a query, ESS uses a capacity-aware, content-based search protocol based on semantic groups and selective one-hop node vector replication, to direct the query to the most relevant nodes responsible for the query, thereby achieving high recall while probing only a small fraction of nodes. We defer the detailed discussion of ESS to Section 38.5.

38.3 Search in Structured P2P Systems

Following the first-generation P2P systems such as Gnutella and KaZaA, structured (or DHT-based) P2P systems,[19,22,26,30] generally called second-generation P2P systems, have been proposed to provide scalable replacement for unscalable Gnutella-like P2P systems.

Such DHT-based systems are adept at *exact-match* lookups: given a key, the system can locate the corresponding document with only $O(\log N)$ hops (N is the number of nodes in system). In structured P2P systems, replication (and caching) has been exploited to improve data availability and search efficiency. Rhea and Kubiatowicz[21] proposed a probabilistic location algorithm to improve the location latency of existing DHT deterministic lookups, if the replica of a requested document exists close to query sources. Their approach is based on *attenuated bloom filters*, a lossy distributed index structure constructed on each node.

However, supporting complex queries such as *keyword search* and *semantic/content search* on top of DHTs is a nontrivial task. The following sections provide a survey of keyword search and semantic search on top of DHTs.

38.3.1 Keyword Search

In keyword search, a query contains one or more keywords (or terms) (e.g., $Q = K_1$ AND K_2 AND K_3, where Q is a query that contains three unique keywords K_1, K_2, and K_3), and the search system returns a set of documents containing all the requested keywords for the query. Basically, three indexing structures have been proposed to support keyword search in structured P2P systems: *global indexing*,[14,20] *hybrid indexing*,[27] and *optimized hybrid indexing*.[27] Figure 38.1 illustrates these three indexing structures, each of which distributes metadata for three documents (D,E, and F) containing terms from a small vocabulary (a,b, and c) to three nodes.

In global indexing (as shown in Figure 38.1(1)), the system as a whole maintains an inverted index that maps each potential term to a set of documents containing that term. Each P2P node stores the complete *inverted list* of those terms that are mapped into its responsible DHT identifier region. An inverted list $a \rightarrow D, F$ indicates that term a appears in documents D and F. To answer a query containing multiple terms (e.g., a and b), the query is routed to nodes responsible for those terms (e.g., nodes 1 and 2). Then, their inverted lists are intersected to identify documents that consist of all requested terms. Although

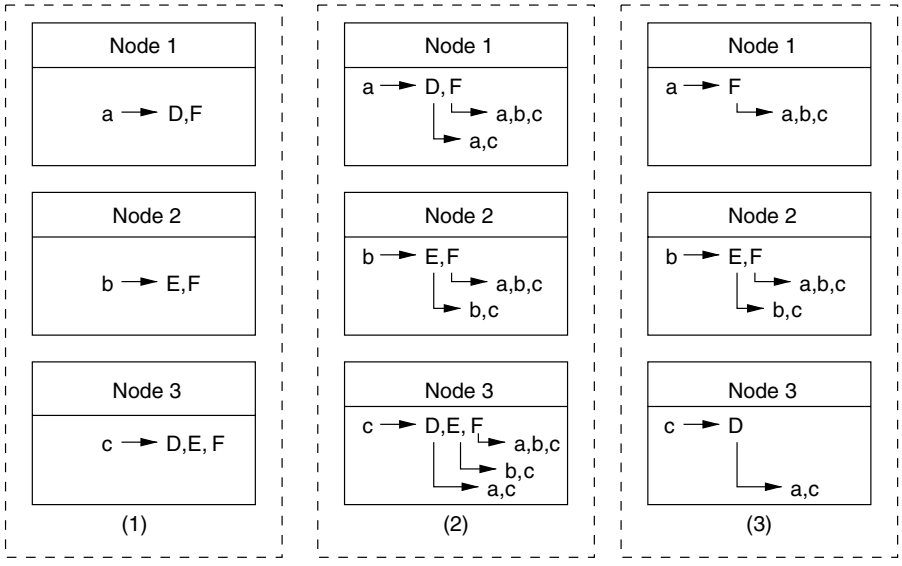

FIGURE 38.1 Three indexing structures on top of DHTs: (1) global indexing; (2) hybrid indexing; (3) optimized hybrid indexing. a, b, and c are terms; D, E, and F are documents. Forward list $D \rightarrow a, c$ indicates that document D contains terms a and c. Inverted list $a \rightarrow D, F$ means that term a appears in documents D and F.

global indexing involves only a small number of nodes for a query (i.e., proportional to the number of terms in the query), it has the drawback of requiring communication in the intersection operation for multiple term conjunctive queries. The communication cost grows proportionally with the length of the inverted lists.

In hybrid indexing (as shown in Figure 38.1(2)), each P2P node maintains the complete inverted list of those terms mapped into its responsible DHT identifier region. In addition, for each document (say, D) in the inverted list for some term t, the node also maintains the complete forward list for document D (a forward list $D \rightarrow a, c$ indicates document D contains terms a and c). Given a multiple keyword query, the query is routed to nodes responsible for those terms. Each of these nodes then performs a local search without contacting others, because they have the complete forward list for each document in their respective inverted lists. This hybrid indexing achieves search efficiency at the cost of publishing more metadata, requiring more communication and storage.

To address the associated cost in hybrid indexing, Tang and Dwarkada[27] proposed an optimized hybrid indexing scheme (as shown in Figure 38.1(3)). The basic idea behind the optimized hybrid indexing is that the metadata for a document is published under the document's top terms,* rather than all of its terms. Figure 38.1(3) illustrates such an optimization. For example, document D containing terms a and c publishes its forward list only at node 3 (responsible for term c), due to the fact that only term c is a top term in D. Given a query containing terms a and c, node 3 can still determine that document D is the answer because it stores complete forward lists for documents in its inverted lists. However, the quality of search results may be degraded because optimized hybrid indexing only publishes the metadata for a document under its top terms. Tang and Dwarkada[27] proposed to adopt automatic query expansion techniques[16] to address this problem. More details can be found in Ref. 27.

*Top terms are defined as those terms central to a document. In the IR algorithms such as vector space model (VSM), terms central to a document are automatically identified by a heavy weight.

38.3.2 Semantic Search

Semantic search is a content-based, full-text search, where queries are expressed in natural language instead of simple keyword match. When a query is issued by a user, a query representative is first derived from its full text, abstract, or title, and then presented to the information retrieval system. For example, when a user issues a query such as "find files similar to file F," a query representative is derived from its full text. Then the query representative is presented to the information retrieval system for those files that are similar to F. Semantic search presents a challenging problem for structured P2P systems: given a query, the system either has to search a large number of nodes or miss some relevant documents. Some semantic search systems[28,32] have been proposed on top of structured P2P systems. One important feature of such search systems is to extend the state-of-the-art information retrieval (IR) algorithms, such as Vector Space Model (VSM) and Latent Semantic Indexing (LSI),[4] to the P2P environment.

pSearch[28] introduces the concept of *semantic overlay* on top of a DHT (i.e., CAN) to implement semantic search. The semantic overlay is a logical network in which documents are organized under their semantic vectors* such that the distance (e.g., routing hops) between two documents is proportional to their dissimilarity in semantic vectors.

Two basic operations are involved in pSearch: *indexing* and *searching*. Whenever a document D enters the system, pSearch performs the indexing operations as follows:

1. Use LSI to derive D's semantic vector V_d.
2. Use rolling-index to generate a number p of DHT keys ($k_i, i = 0, \ldots, p-1$) from V_d.
3. Index D into the underlying DHT using these DHT keys.

Whenever a query Q is issued, pSearch performs the search operations as follows:

1. Use LSI to derive Q's semantic vector V_q.
2. Use rolling-index to generate a number p of DHT keys ($k_i, i = 0, \ldots, p-1$) from V_q.
3. Route Q to the destination nodes that are responsible for these DHT keys.
4. Upon reaching the destination, Q is either flooded to nodes within a radius r or forwarded to nodes using content-directed search.
5. All nodes that receive the query do a local search using LSI and return the matched documents to the query originator node.

More details of pSearch can be found in Ref. 28, and we leave the discussion of the work by Zhu et al.[32] to Section 38.6.

38.4 VSM and Locality-Sensitive Hashing

38.4.1 VSM

We provide an overview of VSM.[4] In VSM, each document or query is represented by a vector of terms. The terms are stemmed words that occur within the document. In addition, stop words** and highly frequent words*** are removed from the term vector. Each term in the vector is assigned a weight. Terms with a relatively heavy weight are generally deemed central to a document. To evaluate whether a document is relevant to a query, the model measures the relevance between the query vector and the document vector.

*A semantic vector is a vector of terms. VSM or LSI represents documents and queries as semantic vectors.

**Stop words are those words that are considered non-informative, like function words (*of*, *the*, etc.), and are often ignored.

***Words appear in a document very frequently, but are not useful to distinguish the document from other documents. For example, if a term t appears in a document very frequently, it should not be included in its term vector. Generally, stop words and high frequency words are removed by using a *stop list* of words.

Typically, VSM computes the relevance between a document D and a query Q as (suppose the term vectors of D and Q have been already normalized):

$$REL(D, Q) = \sum_{t \in D,Q} d_t \cdot q_t \tag{38.1}$$

where t is a term appearing in both D and Q, q_t is term t's weight in query Q, and d_t is term t's weight in document D. Documents with a high relevance score are identified as search results for a query.

A number of term weighting schemes have been proposed for VSM, among which *tf-idf* is a scheme in which the weight of a term is assigned a high numeric value if the term is frequent in the document but infrequent in other documents. The main drawback of *tf-idf* is that it requires global information (i.e., *df*, the document frequency, which represents the number of documents where a term occurs) to compute a term's weight which is not an easy task in the P2P environment. To avoid such a global information requirement, a "dampened" *tf* scheme is proposed, wherein each term is assigned a weight in the form of $1 + \log d_t$ (d_t is the term frequency in a document). Previous work[24] has shown that this scheme not only does not require global information, but also produces higher-quality document clusters.

38.4.2 Locality-Sensitive Hashing (LSH)

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 **Pr** is the probability, and $sim(A, B) \in [0, 1]$ is some similarity function.[6,11]

38.4.2.1 Min-Wise Independent Permutations

Min-wise independent permutations[5] provide an elegant construction of such a locality sensitive hash function family with the Jaccard set similarity measure $sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$, where sets A and B each represents a set of integers.

Let π represent a random permutation on the integer's universe U, $A = \{a_1, a_2, \ldots, a_n\} \subseteq U$, and $B = \{b_1, b_2, \ldots, b_n\} \subseteq U$. The hash function h_π is defined as $h_\pi(A) = min\{\pi(A)\} = min\{\pi(a_1), \pi(a_2), \ldots, \pi(a_n)\}$; 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 sets A and B, 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 $x = h_\pi(A \cup B)$. Because 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 $p = sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$. We refer readers to Refs. 5 and 10 for further details.

38.5 Case Study on Unstructured P2P Systems

We present ESS,[31] an architecture for efficient semantic search on unstructured P2P systems, leveraging the state-of-the-art IR algorithms such as the VSM and relevance ranking algorithms.

38.5.1 Overview

The design goal of ESS is to improve the quality of search (e.g., high recall*) while minimizing the associated cost (e.g., the number of nodes visited for a query). The design philosophy of ESS is that we improve search efficiency and effectiveness while retaining the simple, robust, and fully decentralized nature of Gnutella.

*Recall is defined as the number of retrieved relevant documents, divided by the number of relevant documents.

In ESS, each node has a *node vector*, a compact summary of its content (as shown in Section 38.5.2). And each node can have two types of links (connections), namely *random links* and *semantic links*. Random links connect irrelevant nodes, while semantic links organize relevant nodes* into semantic groups. The topology adaptation algorithm (as discussed in Section 38.5.3) is first performed to connect a node to the rest of the network through either random links, or semantic links, or both.** The goal of the topology adaptation is to ensure that (1) relevant nodes are organized into the same semantic groups through semantic links, and (2) high capacity nodes have high degree and low capacity nodes are within short reach of higher capacity nodes.

Given a query, ESS's search protocol (as discussed in Section 38.5.5) first quickly locates a relevant semantic group for the query, relying on selective one-hop node vector replication (as shown in Section 38.5.4) as well as its capacity-aware mechanism. Then ESS floods the query within the semantic group to retrieve relevant documents. ESS will continue this search process until sufficient responses are found. The intuition behind the flooding within a semantic group is that semantically associated nodes tend to be relevant to the same query.

The main contributions of ESS include the following:

- We propose a distributed, dynamic, and capacity-aware topology adaptation algorithm to organize nodes into semantic groups for efficient search.
- We propose a capacity-aware, content-based search protocol based on semantic groups and selective one-hop node vector replication, to direct queries toward the most relevant nodes that are responsible for the queries, thereby achieving high recall at very low cost.
- Our findings suggest that an appropriate size of node vectors is a very good design choice in both search efficiency and effectiveness, and justify that a good node vector size plays a very important role in system design.
- We introduce automatic query expansion into our system to further improve the quality of search results in both recall and precision. To the best of our knowledge, this is the first work to employ the automatic query expansion technique on Gnutella-like P2P systems.
- We show that ESS's capacity-aware abilities can exploit heterogeneity to make search even more efficient.

38.5.2 Node Vectors

A node vector is a representation of the summary of a node's content. It is derived from a node's locally stored documents as follows. First, each document is represented as a term vector using VSM. The terms in a term vector are the stemmed words that occur within the document. Stop words and highly frequent words are also removed from the term vector. Each term t in the term vector is assigned a weight d_t, the term frequency. Then, all term vectors of a node's documents are summed up and we get a new vector, in which each term component t has a weight d'_t. For each term vector t, we replace its weight d'_t with $1 + \log d'_t$. Finally, we normalize the new vector, and the normalized vector is called the node vector.

As described above, the node vector characterizes a node's content. They are used to determine the relevance of two nodes (say, X and Y) according to the Equation (38.2).

$$REL(X, Y) = \sum_{t \in X,Y} w_{X,t} \cdot w_{Y,t} \tag{38.2}$$

where t is a term appearing in both X and Y; $w_{X,t}$ is term t's weight in X; and $w_{Y,t}$ is term t's weight in Y. If the relevance score is less than a certain relevance threshold, nodes X and Y are deemed irrelevant; otherwise, they are deemed relevant.

*Nodes with similar contents are considered relevant.

**If a node cannot find a relevant node, it connects itself to the rest of the network *only* through random links.

Node vectors are also used to determine the relevance of the node X and a query vector (say, Q) according to Equation (38.3), as shown later in biased walks during search:

$$REL(X, Q) = \sum_{t \in X,Q} w_{X,t} \cdot w_{Q,t} \tag{38.3}$$

Note that in the IR community, a number of term weighting schemes have been proposed. The motivation for why ESS uses the "dampened" tf scheme instead of $1 + \log tf$ in its design is as follows. First and most importantly, unlike other schemes such as *if-idf*, this term weighting scheme does not require global information such as *df*. A term weighting scheme requiring global information contradicts its design philosophy that we retain the simple, robust, and fully decentralized nature of Gnutella. Second, it has been shown by Schutze and Silverstein[24] that such a term weighting scheme works very well and can produce higher quality clusters.

38.5.3 Topology Adaptation Algorithm

The topology adaptation algorithm is a core component that connects a node to the rest of the network and, more importantly, connects the node to a semantic group (if it can find one) through semantic links.

When a node joins the system, it first uses a bootstrapping mechanism in Gnutella to connect to the rest of the network. However, it may not have any information about other nodes' content (i.e., node vectors) as well as semantic groups. Its attempt to gain such information is achieved through *random walks*. A random walk is a well-known technique in which a query message is forwarded to a randomly chosen neighbor at each step until sufficient responses are found. In ESS, the duration of a random walk is also bound by a TTL (time-to-live).

A random walk query message contains a node's node vector, a relevance threshold *REL_THRESHOLD*, the maximum number of responses *MAX_RESPONSES*, and TTL. The random walk returns a set of nodes. In ESS's implementation, a node actually periodically issues two queries: one requesting nodes whose relevance is lower than *REL_THRESHOLD*, and the other requesting nodes whose relevance is higher than or equal to *REL_THRESHOLD*. Note that the relevance score is computed using the Equation (38.2).

The returned nodes are added to the query initiator node's two *host caches* — *random host cache* and *semantic host cache*, respectively — according to their relevance scores. Each entry of host caches consists of *a node's IP address, port number, node capacity, node degree, node vector*,* and *relevance score*. These two caches are maintained throughout the lifetime of the node. Each cache has a size constraint and uses FIFO as replacement strategy.

The goal of topology adaptation is to ensure that, on the one hand, a node is connected to the most relevant nodes through semantic links (thereby forming semantic groups); and on the other hand, its capacity-aware mechanism makes a node connected to higher capacity nodes** through random links. To achieve this goal, each node periodically checks its two caches for random and semantic neighbor addition or replacement.

To add a new *semantic neighbor* (a neighbor node connected by a semantic link), a node (say, X) chooses a node from its semantic cache that is not dead and not already a neighbor and with the *highest* relevance score. Node X then uses a three-way handshake protocol to connect to the chosen neighbor candidate (say, Y). During the handshake, each node decides independently whether or not to accept the other node as a new semantic neighbor based upon its own *MAX_SEM_LINKS* (the maximum number of semantic neighbors), *SEM_LINKS* (the number of current semantic neighbors), and the new node (i.e., the relevance score). If *SEM_LINKS* is less than *MAX_SEM_LINKS*, the node automatically accepts this new connection. Otherwise, the node must check if it can find an appropriate semantic neighbor to drop and replace with the new neighbor candidate. X always accepts Y and drops an existing semantic neighbor

*The semantic host cache does not contain node vectors.

**We assume a node's capacity is a quantity that represents its CPU speed, bandwidth, disk space, etc.

if Y's relevance score is higher than all of X's current semantic neighbors. Otherwise, it makes a decision whether or not to accept Y, as follows. From all of X's semantic neighbors whose relevance scores are lower than that of Y and which are not poorly connected,* X chooses the neighbor Z that has the lowest relevance score. Then it drops Z and adds Y to its semantic neighbors.

To add a new *random neighbor* (a neighbor node connected by a random link), node X chooses a node from its random cache that is not dead and not already a neighbor, and has a capacity greater than its own capacity (e.g., the highest capacity node is preferred). If no such candidate node exists, X randomly chooses a node. X then initiates a three-way handshake to the chosen random neighbor candidate (say, Y). During the handshake, each node independently decides whether or not to accept the other node as a new random neighbor upon the capacities and degrees of its existing random neighbors and the new node. If X's *RND_LINKS* (the number of random neighbors) is less than *MAX_RND_LINKS* (the maximum number of random neighbors, and it dynamically changes as the *SEM_LINKS* changes), the node automatically accepts this new node as a random neighbor. Otherwise, the node must check if it can find an appropriate random neighbor to drop and replace with the new node. X always drops an existing random neighbor in favor of Y if Y has capacity higher than *all* of X's existing random neighbors. Otherwise, it decides whether or not to accept Y as follows. From all of X's random neighbors that have capacity less than or equal to that of Y, X chooses the neighbor Z that has the highest degree. Z will be dropped and replaced with Y *only* if Y has lower degree than that of Z. This ensures that ESS does not drop already poorly connected neighbors and avoids isolating them from the rest of the network.

38.5.3.1 Discussion

The goal of the topology adaptation is to ensure that (1) relevant nodes are organized into semantic groups through semantic links, by periodically issuing random walk queries for semantic neighbors and performing semantic neighbor addition, and (2) high capacity nodes have high degree and low capacity nodes are within short reach of higher capacity nodes, by periodically issuing random walk queries for random neighbors and performing random neighbor addition. Note that in ESS, a *direct* semantic link connects two *most* relevant nodes; while in other systems like SETS, a *local* link does not necessarily mean that two connected nodes are most relevant within a topic segment.

For random walk queries requesting relevant nodes, ESS can actually do an optimization as follows. When a query arrives at a node (say, Y) that is deemed to be relevant to the initiator node (say, X), Y can first choose other relevant nodes from its semantic host cache for responses with some probability. If the *MAX_RESPONSES* has been reached, the query reply is routed back to X. Otherwise, Y biased walks the query through one of its semantic links with some probability. Further, relevant nodes within a semantic group can exchange the content of their host caches. Currently, ESS does not adopt such optimizations. Each node also continuously keeps track of the relevance scores of both semantic links and random links. If the relevance score of a semantic link drops below the *SEM_THRESHEOLD* due to dynamically changing documents in either node (and thus changing node vectors), ESS simply drops the semantic link and adds the neighbor information into the random host cache. Similarly, if the relevance score of a random link rises above the *SEM_THRESHOLD*, ESS simply drops the random link and adds the neighbor information into the semantic host cache. As a result, the topology adaptation process performed thereafter can adapt to the dynamically changing node vectors of each node's existing neighbors.

38.5.4 Selective One-Hop Node Vector Replication

To allow ESS to quickly locate the relevant semantic group for a query during biased walks (as shown in Section 38.5.5), each node maintains the node vectors of *all* of its random neighbors in memory. Note that ESS does not maintain those of its semantic neighbors. This is why we call it *selective* replication.

*Each node has a minimum degree constraint, and a typical value is 3. If a node's degree is less than or equal to the minimum constraint value, this node is identified as a poorly connected node.

When a random connection is lost, either because the random neighbor node leaves the system or due to topology adaptation, the node vector for this neighbor gets flushed from memory. A node periodically checks the replicated node vectors with each random neighbor in case a neighbor node might add or remove documents. This allows nodes to adapt to dynamically changing node vectors (due to dynamically changing of documents) and keep replicated node vectors up-to-date and consistent.

38.5.5 Search Protocol

The combination of the topology adaptation algorithm (whereby relevant nodes are organized into semantic groups and high capacity nodes have more neighbors [i.e., high degree]) and selective one-hop node vector replication (whereby nodes maintain the node vectors of their random neighbors) have paved the way for ESS's content-based, capacity-aware search protocol.

Given a query, the search protocol is performed as follows. First, ESS uses *biased* walks, rather than random walks, to forward the query through *random links*. During biased walks, each node along the route looks up its locally stored documents for those satisfying the query; each document is evaluated using Equation (38.1) and a relevance score is computed. If the relevance score is higher than or equal to a certain relevance threshold, this document is identified as a relevant document for this query. If any such a relevant document is identified, then the node (say, X) is called the *semantic group target node*, where the query terminates biased walks and starts flooding. Otherwise, X selects a neighbor (say, Y) from its *random neighbors* whose node vector is *most* relevant to the query vector according to the Equation (38.3), and forwards the query to Y. The biased walks are repeated until a semantic group target node is identified.

The target node then floods the query along its *semantic links*: each semantic neighbor evaluates the query against its documents and then floods the query along its own semantic links. During flooding, we can allow the query to probe all the nodes within a semantic group or only a fraction of nodes by imposing a flooding radius constraint from the target node (called *controlled flooding*). The relevant documents found within the semantic group are directly reported to the target node. Note that each query contains a *MAX_RESPONSES* parameter. The target node aggregates the relevant documents, reports them directly to the query initiator node (which will present highest relevance ranking documents to the user), and decreases *MAX_RESPONSES* by the number of relevant documents. If *MAX_RESPONSES* becomes less than or equal to zero, the query is simply discarded; otherwise, the query starts biased walks from the target node again and repeats the above search process until sufficient responses are found.

During both biased walks and flooding, ESS uses bookkeeping techniques to sidestep redundant paths. In ESS, each query is assigned a unique identifier *GUID* by its initiator node. Each node keeps track of the neighbors to which it has already forwarded the query with the same *GUID*. During biased walks, if a query with the same *GUID* arrives back at a node (say, X), it is forwarded to a different random neighbor with the highest relevance score among those random neighbors to which X has not yet forwarded the query. This reduces the probability that a query traverses the same link twice. However, to ensure forward progress, if X has already sent the query to all its random neighbors, it flushes the bookkeeping state and starts reusing its random neighbors. On the other hand, during flooding, if a query with the same *GUID* arrives back at the node, the query is simply discarded. Note that ESS nodes treat query messages with the same *GUID*s differently during biased walks and flooding.

The search protocol discussed thus far does not consider node capacity heterogeneity. Now we incorporate the capacity-aware mechanism into the search protocol to make search more efficient in the system where node capacities are heterogeneous. Due to the fact that the topology adaptation algorithm takes into account the heterogeneity of node capacities *only* in random link construction, the search protocol only needs to adapt the biased walk while retaining the flooding part untouched. During biased walks, each node makes a decision how to forward a query based on *the query vector, its own capacity, the capacities of all of its random neighbors,* and *the node vectors of all of its random neighbors*. If the node (say, X) is a supernode (whose capacity is higher than a certain threshold), it forwards the query to a random neighbor whose node vector is most relevant to the query vector. Otherwise, X must check all its random neighbors and chooses the neighbor (say, Y) with the highest capacity. If Y is a supernode, X forwards the query to Y, hoping that high capacity nodes can typically provide useful information for the query. Otherwise,

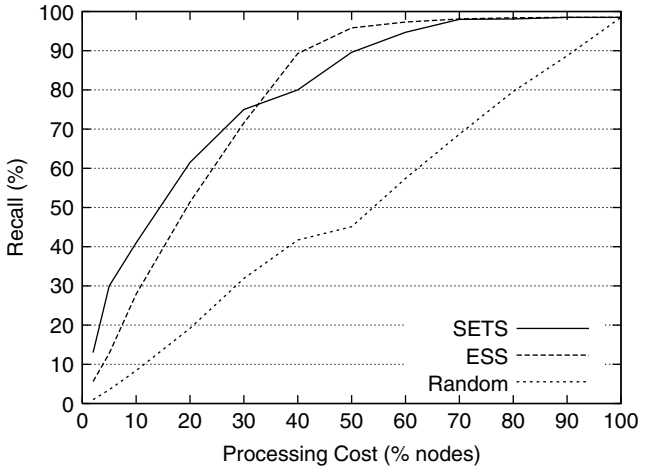

FIGURE 38.2 Recall versus processing cost for ESS, SETS, and Random.

X forwards the query to a random neighbor whose node vector is most relevant to the query vector. The biased walks are repeated until a target node if reached.

38.5.5.1 Discussion

In summary, the query flooding within a semantic group is based on the intuition that semantically associated nodes tend to be relevant to the same queries and can provide useful responses for them. The biased walk, taking advantage of the heterogeneity of node capacities and selective one-hop node vector replication, forwards a query either to a supernode neighbor with the hope that high capacity nodes can typically provide useful information for the query, or to the *most* relevant random neighbor if no such a supernode exists. Note that *biased walks direct a query along one of a node's random links, while flooding forwards the query along all of the node's semantic links*. In ESS, in addition to *MAX_RESPONSES*, each query is also bound by the *TTL* parameter. Note that flooding within semantic groups keeps us from exactly keeping track of the *TTL*. For simplification, in ESS's implementation, the *TTL* is decreased by one at each step *only* during biased walks. Once *TTL* hits zero, the query message is dropped and no longer forwarded.

38.5.6 Experimental Results

We present part of experimental results here and refer readers to Zhu and Hu[31] for more results.

The data used in the experiments is *TREC-1,2-AP*.[1] The TREC corpus is a standard benchmark widely used in the IR community. *TREC-1,2-AP* contains AP Newswire documents in TREC CDs 1 and 2. The queries used in the experiments are from TREC-3 ad hoc topics (151–200). The query vector was derived from the *title* field using VSM. These 50 queries each come with a query-relevant judgment file that contains a set of manually identified relevant documents.

Figure 38.2 plots evaluation results comparing the performance of ESS against SETS and Random.* The vertical axis is the recall, and the horizontal axis is the query processing cost in the fraction of nodes involved in a query.

Several observations can be drawn from this figure:

1. ESS and SETS outperform Random substantially, achieving higher recall at smaller query processing cost.
2. Compared to ESS, SETS achieves higher recall when exploring less than 30 percent of the nodes. This is explained by the fact that SETS takes advantage of knowing the global C (=256) topic

*Random represents the random walk technique.

TABLE 38.1 Recall Improvements with Respect to Query Processing Cost Made by ESS(1000+heter) on SETS(full)

	Processing Cost (% nodes)						
	2	*5*	*10*	*20*	*30*	*40*	*≥ 50*
ESS(1000+heter) : SETS(full)	63.8	8.3	16.1	17.9	13.3	18.5	≤ 7.4

Note: ESS(1000+heter) represents ESS, which uses an appropriate node vector size of 1000 and considers heterogeneity; "full" represents the full node vector size.

segments and therefore can quickly and precisely locate the most relevant topic segments to look up relevant documents. ESS instead has to use biased walks to locate a target node and then floods the query within the corresponding semantic group for relevant documents. If the target node is not a right one (which actually does not contain relevant documents, although some of its documents have a relevance score high enough to be deemed relevant), some irrelevant nodes are unavoidably probed. The overhead of locating a right target node hurts the performance of ESS, especially when probing *only* a small fraction of nodes. However, ESS still achieves about 71.6 percent recall by probing *only* 30 percent of the nodes.

3. ESS outperforms SETS when exploring more than 30 percent of the network. It achieves 89.3 percent recall by visiting only 40 percent of the nodes, while SETS achieves 80 percent recall in this case. We give the following explanations. First, the overhead of locating a right target node (and thus the semantic group) is amortized by exploring more nodes. Second, the nature of ESS's topology adaptation connects the *most* relevant nodes through *direct* semantic links, and it ensures that a query probes the *most* relevant nodes first along semantic links. However, SETS does not distinguish the relevance between nodes within a topic segment, and local links do not necessarily reflect that the *most* relevant nodes have *direct* connections. Therefore, some irrelevant nodes within topic segments are unavoidably visited when flooding the query within topic segments.
4. When exploring the whole network, the recall achieved by all three systems is 98.5 percent. This is because queries are short on average, with only 3.5 terms in the experiments. Some relevant documents could not be identified because their relevance scores computed using Equation (38.1) are 0. During query evaluation, they are mistakenly deemed irrelevant due to such a low relevance score. In other words, with such short queries, the maximum recall achieved by a centralized IR system is 98.5 percent.

Table 38.1 summarizes the recall improvements made by ESS(1000+heter) on SETS(full), which does not consider capacity heterogeneity in its design. The node capacities are based on a Gnutella-like profile that was derived from the measured bandwidth distribution for Gnutella.[23] Table 38.1 shows that, with an appropriate node vector size and capacity-aware mechanism, ESS outperforms SETS.

38.6 Case Study on Structured P2P Systems

We present an efficient, LSH-based semantic search system[32] built on top of DHTs. Leveraging the state-of-the-art IR algorithms such as VSM, this system aims to provide efficient semantic indexing and retrieval capabilities for structured P2P systems.

38.6.1 Overview

Figure 38.3 illustrates the system architecture. To support semantics-based access, we must add two major components into an existing P2P system: (1) *a registry of semantic extractors* and (2) *semantic indexing and locating utility*.

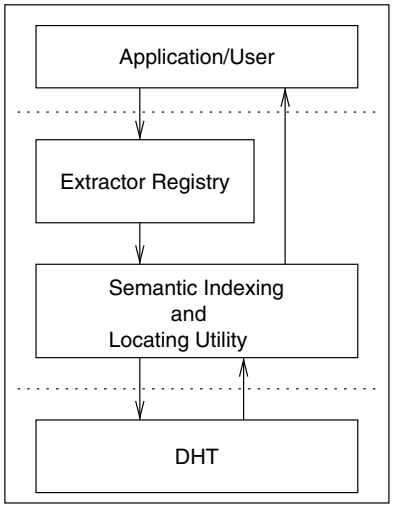

FIGURE 38.3 Major components of the system architecture.

The functionality of semantic indexing is to index each object automatically according to its semantic vector (SV) whenever an object is created or modified. The functionality of semantic locating is to find similar documents for a given query.

The index table is fully distributed. When an object is created or modified, its SV is extracted. The system then hashes the SV to an integer number called *semID*. The DHT uses this *semID* as a key to put an index entry (a pointer to the original object) into the P2P system. Note that the original locations of documents are not affected. Given a query, the system generates a *semID* based on the query's semantic vector. The semantic locating utility then uses the *semID* to locate the indices of similar documents stored in the P2P systems.

The key here is to make sure that two semantically close documents (which have similar semantic vectors) will be hashed to the same *semID* so that the underlying DHT can locate the indices. However, this is not possible in many traditional hashing functions that try to be uniformly random. As a result, two documents that are similar but slightly different (e.g., different versions of the same document) will generate different hashing results. Our system, on the contrary, relies on a very special class of hashing functions called Locality-Sensitive Hashing. If two documents are similar, then it is likely that they will generate the same hashing result. Moreover, the higher the similarity between the two files, the higher the probability that the hashing results are the same.

However, LSH cannot guarantee that two similar documents will always have the same hashing result. To increase the probability, we use a group of n LSH functions to generate n *semID*s (n is a small number, about 5 to 20). If the probability of generating a matching result from a single LSH function is p, then the probability of generating at least one matching result from n LSH functions will be $1 - (1 - p)^n$. The locating utility then uses the resulting n *semID*s to search the DHT. Our initial results indicate that with n set to about 10 to 20, our system can find almost 100 percent of semantically close documents. As a result, our system is very efficient: instead of sending the query to tens of thousands of nodes in the system, we only need to send it to n nodes.

38.6.1.1 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 document and outputs the corresponding semantic vector (SV) of that document.

A SV is a vector of file-type specific features extracted from the file content. For example, the VSM extracts the term frequency information from text documents, and Welsh et al.[29] have derived frequency, amplitude, and tempo feature vectors from music data.

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 *document* and *query* as a semantic vector where each dimension is associated with a distinct term (or keyword).

Semantically close documents or queries are considered to have similar SVs. The similarity between documents or queries can be measured as the cosine of the angle[4] or the Jaccard set similarity measure between their vector representations.

Whenever a user or application on a node X wants to store a document D in the system, the semantic extractor registry on X is responsible for deriving an SV for D. The resulting SV is then used to produce a small number of *semIDs* (semantic identifiers) as the DHT keys for D. As a result, the document D can be indexed into the DHT according to its semantic presentation, that is, with the resulting *semIDs* as the DHT keys. Note that a document in our system has two kinds of DHT keys: (1) *docID*, produced by SHA-1 hash of its content or name (which is commonly used in current P2P systems for file storage and retrieval); and (2) *semID*, derived from its semantic vector (which is used for semantic indexing, as discussed later).

38.6.1.2 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 document or query automatically according to its semantic vector whenever a document or query is created or modified. The functionality of semantic locating is to locate semantically close documents for a given query.

Given a document, the semantic indexing and locating utility hashes its SV into a small number of *semID*s using locality sensitive hashing functions (LSH). By having these *semID*s as the DHT *keys* and *docID* and *SV* as the *objects* in the DHT's interface of *put(key, object)*, it indexes the document into the underlying DHT in the form of tuple of $< semID, docID, SV >$.

When locating semantically close files for a given query, the semantic indexing and locating utility first hashes the query's SV into a set of *semID*s. It then interacts with the DHT to retrieve the indices of those files that satisfy the query, by having these *semID*s as the DHT *keys* in the DHT's interface of *get(key)*. The result of a successful query will return a list of *docID*s that satisfy the query.

The semantic indexing and locating utility also generates materialized views of query results, and allows users to reuse these materialized views as regular objects to save the expensive processing cost of popular queries. We refer readers to Zhu et al.[32] for more details.

38.6.2 LSH-Based Semantic Indexing

The objective of semantic indexing is to cluster the indices of semantically close files or queries to the same peer nodes with high probability. Without loss of generality, our focus here is on documents. Queries can also apply the same indexing procedure.

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

1. For each vector component t of A, convert it into a 64-bit integer* by taking the first 64 bits of t'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 Algorithm 38.1).

*Because we evaluate our system on the Pastry simulator of 64-bit identifier space, we here convert t into a 64-bit integer.

Algorithm 38.1 Semantic indexing procedure using n groups of m LSH functions.

Require: $g[1..n]$, each of which has m hash functions $h[1..m]$
1: Convert A into A', which is a set of integers
2: **for** $j = 1$ to n **do**
3: $semID[j] = 0$
4: **for** each $h[i] \in g[j]$ **do**
5: $semID[j] \wedge = h[i](A')$ /* $\wedge$ is a XOR operation */
6: **end for**
7: **end for**
8: **for** each $semID[j]$ **do**
9: insert the index $<semID[j], docID, A>$ into the DHT by having $semID[j]$ as the DHT key
10: **end for**

Note that for two SVs A and B, their similarity is $p = sim(A, B) = sim(A', B')$. This is because 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 document with a *docID*, as shown in Algorithm 38.1. A *semID* is produced by XORing m 64-bit integers that are produced by applying a group of m hash functions on A'. Thus, applying n such groups of hash functions on A' yields n *semID*s. By having the resulting *semID*s as the DHT keys, the file is indexed into the DHT in the form of $< semID, docID, A >$. We expect that such semantic indexing could have the indices of semantically close files hashed to the same peer nodes with *high* probability.

Probability Analyses. What is the probability achieved by the procedure as shown in Algorithm 38.1? We here offer probability analyses. Consider a group of $g = \{h_1, h_2, \ldots, h_m\}$ of m hash functions chosen uniformly at random from a family of locality sensitive hash functions. Then the probability that two SVs A and B are hashed to the same 64-bit integer for all m hash functions is $\mathbf{Pr}[g(A) = g(B)] = p^m$ (where $p = sim(A, B) = sim(A', B')$). Now, for n such groups $g_1, g_2, \ldots, g_n$ of hash functions, the probability that A and B cannot produce the same integer for all m hash functions $\in g_i$ is $1 - p^m$. And the probability that this happens for all n groups is $(1 - p^m)^n$. So, the probability that A and B can produce the same integer for all m hash functions of *at least* one of n groups is $1 - (1 - p^m)^n$. That is, the probability that A and B can produce the same *semID* for at least one of n groups is $1 - (1 - p^m)^n$.

For example, if two SVs A and B have a similarity $p = 0.7$, the probability of hashing these two SVs to at least one same *semID* is 0.975 ($m = 5, n = 20$). Semantically close files are considered to have similar SVs. We ensure that the indices of semantically close files could be hashed to the same *semIDs* with very high probability (nearly 100 percent) by carefully choosing the values of m and n.

Note that each node in a P2P system is responsible for a portion of the DHT's identifier space. Close identifiers could be mapped into the same node. So even if two SVs A and B cannot be hashed to one same identifier *semID* by applying n such groups of hash functions, it is still possible that the semantic indices of these two corresponding files might be hashed to the same node if both SVs are hashed into some *semID*s that are close together in the identifier space. This implies that the probability of hashing these two SVs to at least one same node is greater than or equal to $1 - (1 - p^m)^n$.

As a result, we can improve the performance of our probabilistic approach by adjusting the parameters n and m. Ideally, we would like the probability $1 - (1 - p^m)^n$ to approach 100 percent. By assuming that semantically close files have a relatively high similarity value (e.g., ≥ 0.7), the probability is now dependent on n and m. Recall that the probability is 97.5 percent when p, n, and m are chosen to be 0.7, 20, and 5, respectively. If we increase n to 30 while keeping p and m fixed, the probability is 99.6 percent; if we reduce m to 1 while keeping p and n fixed, the probability is nearly 100 percent. Therefore, either a relatively big n or a relatively small m would dramatically improve the performance of our probabilistic approach. However, a big n would *increase the load of indexing and querying as well as storage cost*, and a small m might *cluster the indices of those files that are not very semantically close (with low similarity) to the same nodes*

with non-negligible probability. Another interesting fact is that we could use a small m if in a system those files with a relatively low similarity ($p \leq 0.5$) are also regarded as semantically close files. For example, the probability is nearly 100 percent if p, n, and m are 0.3, 20, and 1, respectively. In summary, all these problems need further exploration in our future work.

38.6.3 LSH-Based Semantic Locating

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

Given the semantic indexing scheme described previously, we show that this goal can be easily achieved. For example, let V_q be a query Q's semantic vector. Suppose Q wants to locate those documents whose SVs are similar to A (with certain similarity degree). The semantic locating procedure (as shown in Algorithm 38.2) produces n *semID*s from V_q for Q using the same set of hash functions (used in the semantic indexing procedure). So, if a document D satisfies such a query Q, it will be retrieved by Q with very high probability. Note that the SVs of document D and query Q could be hashed to the same *semIDs* with high probability (i.e., $1 - (1 - p^m)^n$). Thus, by having these *semID*s as the DHT *keys* in the DHT's interface of *get(key)*, Q is able to retrieve semantically close documents from the peer nodes that are responsible for these *semIDs*. n is very small (e.g., 20) in the system, which implies that a query can be answered by consulting only a small number n of nodes.

As shown in Algorithm 38.2, upon a request, each destination peer (at most n) locally checks the list of tuples $<semID, docID, SV>$ and finds the *docID*s such that their associated SVs are similar to the query's SV with certain similarity threshold, and sends the list of *docID*s to the requesting node. Then, the requesting node 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 using *data clustering* techniques such as k-means clustering.[12] It should be pointed out that we do not use LSH to perform local matching because LSH could hash the indices of dissimilar files into the same *semIDs* with some probability. Each destination peer uses k-means to cluster the tuples into collections according to the semantic vector until the variance inside a collection falls below a 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.

Algorithm 38.2 Semantic locating procedure using n groups of m LSH functions.

Require: $g[1..n]$, each of which has m hash functions $h[1..m]$
1: Convert V_q into V_q', which is a set of integers
2: **for** $j = 1$ to n **do**
3: $semID[j] = 0$
4: **for** each $h[i] \in g[j]$ **do**
5: $semID[j] \wedge = h[i](V_q')$ /* $\wedge$ is a XOR operation */
6: **end for**
7: **end for**
8: **for** each $semID[j]$ **do**
9: Send the request to the destination node which is responsible for $semID[j]$
10: **end for**
11: Get replies from all the destination nodes
12: Merge the *docID*s that satisfy the query from all replies
13: Create a materialized view of the query result asynchronously
14: Index the query according to its semantic vector asynchronously

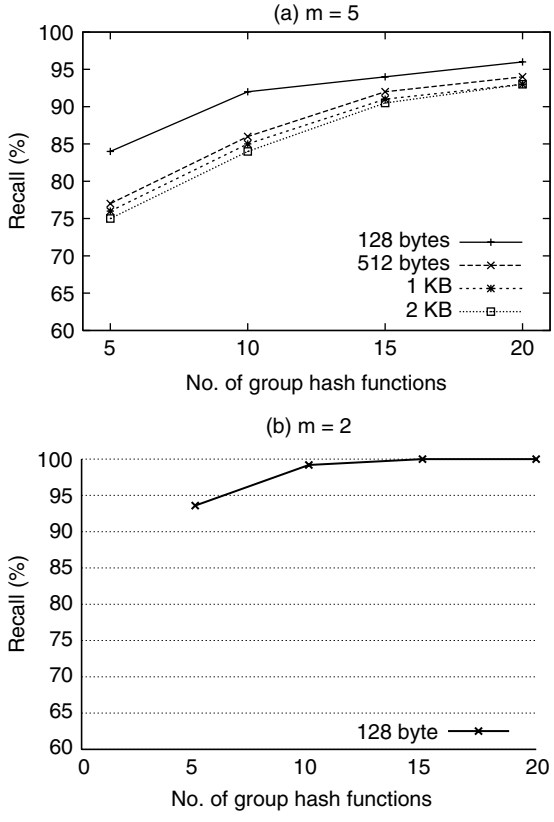

FIGURE 38.4 Performance of semantic locating, where n is the number of LSH function groups and m is the number of LSH functions in each group.

38.6.4 Experimental Results

We present part of experimental results here and refer readers to Zhu et al.[32] for more details.

The results reported here are based on a P2P file system built on top of Pastry.[22] The data used in the experiments consists of 205 unique C++ program files from a CVS repository. Each of these 205 program files consists of three different versions, on average. Hence, there are 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.[17] 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.[17] We started the experiment with an empty P2P file system and indexed each file into the system by applying the semantic indexing procedure. Then we issued a set of queries to locate different versions for each unique C++ program file, because here we consider different versions of a program file semantically close due to similar fingerprints.

Figure 38.4(a) shows the recall for different minimum chunk size limits,* including 128 bytes, 512 bytes, 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 recall. As expected, the recall increases with the number n, increasing from 5 to 20. Moreover, as the minimum chunk size varies from 128 bytes to 2 KB, the recall 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

*When dividing a file into chunks, we impose a minimum chunk size limit like in LBFS.[17]

is 128 bytes and n is 20, our semantic locating approach was still unable to find all the files. This is because some files are very small; even a minimum chunk size limit of 128 bytes 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 percent probability). Further, a small m could dramatically improve the recall according to $(1-(1-p^m)^n)$. Figure 38.4(b) shows the result for a minimum chunk size limit of 128 bytes with $m = 2$. When n is 15, the recall is 100 percent.

38.6.5 Top Term Optimization

By realizing the fact that in a document, a small number (e.g., 30) of top terms are much more important than other terms,[31] we propose a top term-based optimization for the basic LSH-based semantic indexing and locating approach. The intuition behind this optimization is that, according to the similarity computing Equation 38.1, top terms with heavy weight tend to contribute most in similarity score in comparison to those terms with light weight in a document. Therefore, we can represent a document by a *compact* term vector, which consists of only those top terms. We believe this could reduce the cost of both LSH computation in semantic indexing and location and storage (as shown by Zhu et al.[32]). Furthermore, top term-based optimization might help to identify the most relevant documents for the user's query, thus preventing the system from returning too many documents (most of them may be irrelevant) beyond the user's capability to deal with.

38.7 Summary

This chapter addressed the advantages of constructing a P2P search system and reviewed current search systems built on top of both unstructured and structured P2P systems. It also discussed in detail two search systems built on top of unstructured P2P networks and structured P2P networks, respectively.

Structured P2P systems are adept at exact-match lookups: given a key, the system can locate the corresponding document with only $O(\log N)$ hops. However, as discussed previously, extending exact-match lookups to support keyword/semantic search[14,20,27,32] on DHTs is nontrivial. The main problem facing these search techniques is the high maintenance cost in both overlay structure and document indices due to node churn in P2P networks. To counter this problem, we expect to construct the search engine using a subset of P2P nodes that are stable and have good connectivity.

Unstructured P2P systems, on the other hand, automatically support arbitrarily complex queries. In addition, node churn causes little problem for them. However, the main problem facing the search systems built on top of unstructured P2P networks is the search inefficiency — a query might probe a very large fraction of nodes to be answered. As a result, a number of search techniques[3,7,8,15,25,31] have been proposed to improve search efficiency on unstructured P2P systems.

References

1. Text retrieval conference (trec). http://trec.nist.org.
2. L.A. Barroso, J. Dean, and U. Holzle. Web search for a planet: the google cluster architecture. *IEEE Micro,* 23(2):22–28, 2003.
3. M. Bawa, G. Manku, and P. Raghavan. SETS: Search enhanced by topic segmentation. In *Proceedings of the 26th Annual International ACM SIGIR Conference,* pages 306–313, Toronto, Canada, July 2003.
4. Michael W. Berry, Zlatko Drmac, and Elizabeth R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review,* 41(2):335–362, 1999.
5. Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences,* 60(3):630–659, 2000.
6. Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC),* pages 380–388, Quebec, Canada, May 2002.

7. E. Cohen, H. Kaplan, and A. Fiat. Associative search in peer to peer networks: harnessing latent semantics. In *Proceedings of IEEE INFOCOM,* Volume 2, pages 1261–1271, San Francisco, CA, April 2003.
8. Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS),* pages 23–32, Vienna, Austria, July 2002.
9. K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM,* pages 381–394, Karlsruhe, Germany, August 2003.
10. 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 (CIDR),* Asilomar, CA, January 2003.
11. Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC),* pages 604–613, Dallas, TX, May 1998.
12. A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: a review. *ACM Computing Surveys,* 31(3):264–323, 1999.
13. S. Lawrence and C.L. Giles. Accessibility of information on the web. *Nature,* 400:107–109, 1999.
14. J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems(IPTPS),* pages 207–215, Berkeley, CA, February 2003.
15. Qin Lv, Pei Cao, and Edith Cohen. Search and replication in unstructured peer-to-peer networks. In *Proceedings of 16th ACM Annual International Conference on Supercomputing (ICS),* pages 84–95, New York, NY, June 2002.
16. Mandar Mitra, Amit Singhal, and Chris Buckley. Improving automatic query expansion. In *Proceedings of ACM SIGIR,* pages 206–214, Melbourne, Australia, 1998.
17. Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP),* pages 174–187, Banff, Canada, October 2001.
18. C.D. Prete, J.T. MacArthur, Richard L. Villars, I.L. Nathan Redmond, and D. Reinsel. Industry developments and models, disruptive innovation in enterprise computing: storage. In *IDC,* February 2003.
19. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM,* pages 161–172, San Diego, CA, August 2001.
20. P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference (Middleware),* pages 21–40, Rio de Janeiro, Brazil, June 2003.
21. Sean C. Rhea and John Kubiatowicz. Probabilistic location and routing. In *Proceedings of IEEE INFOCOM,* Volume 3, pages 1248–1257, New York, NY, June 2002.
22. Antony Rowstron and Peter 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),* pages 329–350, Heidelberg, Germany, November 2001.
23. Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking(MMCN),* San Jose, CA, January 2002.
24. H. Schutze and C. Silverstein. A comparison of projections for efficient document clustering. In *Proceedings of ACM SIGIR,* pages 74–81, Philadelphia, PA, July 1997.
25. Kunwadee Spripanidkulchai, Bruce Maggs, and Hui Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *Proceedings of IEEE INFOCOM,* Volume 3, pages 2166–2176, San Francisco, CA, March 2003.

26. 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, August 2001.
27. Chunqiang Tang and Sandhya Dwarkada. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *Proceedings of First Symposium on Networked Systems Design and Implementation (NSDI),* pages 211–224, San Francisco, CA, March 2004.
28. Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of ACM SIGCOMM,* pages 175–186, Karlsruhe, Germany, August 2003.
29. M. Welsh, N. Borisov, J. Hill, R. von Behren, and A. Woo. Querying Large Collections of Music for Similarity. Technical Report UCB/CSD00-1096, U.C. Berkeley, November 1999.
30. B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-Tolerance Wide-Area Location and Routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, April 2001.
31. Yingwu Zhu and Yiming Hu. ESS: Efficient Semantic Search on Gnutella-Like P2P Systems. Technical report, Department of ECECS, University of Cincinnati, March 2004.
32. Yingwu Zhu, Honghao Wang, and Yiming Hu. Integrating semantics-based access mechanisms with P2P file systems. In *Proceedings of the 3rd International Conference on Peer-to-Peer Computing,* pages 118–125, Linkping, Sweden, September 2003.