Searching Techniques in Peer-to-Peer Networks

Xiuqi Li and Jie Wu
Department of Computer Science and Engineering,
Florida Atlantic University
Boca Raton, FL 33431

Abstract

This chapter provides a survey of major searching techniques in peer-to-peer (P2P) networks. We first introduce the concept of P2P networks and the methods for classifying different P2P networks. Next, we discuss various searching techniques in unstructured P2P systems, strictly structured P2P systems, and loosely structured P2P systems. The strengths and weaknesses of these techniques are highlighted. Searching in unstructured P2Ps covers both blind search schemes and informed search schemes. Blind searches include iterative deepening, k-walker random walk, modified random BFS, and two-level k-walker random walk. Informed searches include local indices, directed BFS, intelligent search, routing indices, attenuated bloom filter, adaptive probabilistic search, and dominating set based search. The discussion of searching in strictly structured P2Ps focuses on hierarchical Distributed Hash Table (DHT) P2Ps and non-DHT P2Ps. Searching in non-hierarchical DHT P2Ps is briefly overviewed. The presentation of the hierarchical DHT P2Ps pays more attention to Kelips and Coral, whereas that of searching in non-DHT P2Ps focuses on SkipNet and TerraDir. The description of searching in loosely structured P2Ps focuses on Freenet. We conclude this chapter by summarizing open problems in searching the P2P networks.

Keywords: data management, peer-to-peer networks, routing, searching

Abstract

References

1. Introduction

There has been a growing interest in peer-to-peer networks since the initial success of some very popular file-sharing applications such as Napster and Gnutella [15]. A peer-to-peer (P2P) network is a distributed system in which *peers* employ distributed resources to perform a critical function in a decentralized fashion. Nodes in a P2P network normally play *equal roles*, therefore, these nodes are also called peers. A typical P2P network often includes computers in unrelated administrative domains. These P2P participants join or leave the P2P system frequently, hence, P2P networks are *dynamic* in nature. P2P networks are overlay networks, where nodes are end systems in the Internet and maintain information about a set of other nodes (called neighbors) in the P2P layer. These nodes form a virtual overlay network on top of the Internet. Each link in a P2P overlay corresponds to a sequence of physical links in the underlying network. Examples of P2P applications are distributed file-sharing systems, event notification services, and chat services [1] [3] [4] [5].

P2P networks offer the following benefits [1] [3]:

- They do not require any special administration or financial arrangements.
- They are self-organized and adaptive. Peers may come and go freely. P2P systems handle these events automatically.
- They can gather and harness the tremendous computation and storage resources on computers across the Internet.
- They are distributed and decentralized. Therefore, they are potentially fault-tolerant and load-balanced.

P2P networks can be classified based on the control over data location and network topology. There are three categories: *unstructured*, *loosely structured*, and *highly structured* [7]. In an unstructured P2P network such as Gnutella [15], no rule exists which defines where data is stored and the network topology is arbitrary. In a loosely structured network such as Freenet [34] and Symphony [31], the overlay structure and the data location are not precisely determined. In Freenet, both the overlay topology and the data location are determined based on hints. The network topology eventually evolves into some intended structure. In Symphony, the overlay topology is determined probabilistically but the data location is defined precisely. In a highly structured P2P network such as Chord [16], both the network architecture and the data placement are precisely specified. The neighbors of a node are well-defined. The data is stored in a well-defined location.

P2P networks can also be classified into *centralized* and *decentralized* [7] [11] [12]. In a centralized P2P such as Napster [53], a central directory of object location, ID assignment, etc. is maintained in a single location. Peers find the locations of desired files by querying the central directory server. Such P2Ps do not scale well and the central directory server causes single point of failure. Decentralized P2Ps adopt a distributed directory structure. These systems can be further divided into *purely decentralized* and *hybrid* [11] [12]. The difference between them lies in the role peers play. In purely decentralized systems such as Gnutella and Chord, peers are totally equal. In hybrid systems, some peers called *dominating nodes* [2] or *superpeers* [25] serve the search request of other regular peers. Peers in a P2P system are often heterogeneous in computation power, stability, and connectivity. Purely decentralized systems can not take advantage of this heterogeneity while hybrid systems can. However, dominating nodes and superpeers have to be carefully selected to avoid single points of failure and service bottlenecks.

P2P systems can also be classified into *hierarchical* and *non-hierarchical* based on whether the overlay structure is a hierarchy or not. Most purely decentralized systems have flat overlays and are non-hierarchical systems. All hybrid systems and few purely decentralized systems such as

Kelips [23], are hierarchical systems. Non-hierarchical systems offer load-balance and high-resilience. Hierarchical systems provide good scalability, opportunity to take advantage of node heterogeneity, and high routing efficiency.

There are many research issues in P2P computing. This chapter focuses on searching techniques in P2P networks. Searching means locating desired data. Most existing P2P systems support the simple object lookup by key or identifier. Some existing P2P systems can handle more complex keyword queries, which find documents containing keywords in queries. More than one copy of an object may exist in a P2P system. There may be more than one document that contains desired keywords. Some P2P systems are interested in a single data item; others are interested in all data items or as many data items as possible that satisfy a given condition. Most searching techniques are forwarding-based. Starting with the requesting node, a query is forwarded (or routed) node to node until the node which has the desired data (or a pointer to the desired data) is reached. To forward query messages, each node must keep information about some other nodes called neighbors. The information of these neighbors constitutes the routing table of a node.

The desired features of searching algorithms in P2P systems include high-quality query results, minimal routing state maintained per node, high routing efficiency, load balance, resilience to node failures, and support of complex queries. The quality of query results is application dependent. Generally, it is measured by the number of results and relevance. The routing state refers to the number of neighbors each node maintains. The routing efficiency is generally measured by the number of overlay hops per query. In some systems, it is also evaluated using the number of messages per query. Different searching techniques make different trade-offs between these desired characteristics.

Searching in highly structured systems follows the well-defined neighboring links. For this reason, highly structured P2P systems provide guarantees on finding existing data and bounded data lookup efficiency in terms of the number of overlay hops; however, the strict network structure imposes high overhead for handling frequent node join-leave. Unstructured P2P systems are extremely resilient to node join-leave, because no special network structure needs to be maintained. Searching in unstructured networks is often based on flooding or its variation because there is no control over data storage. The searching strategies in unstructured P2P systems are either blind search or informed search. In a blind search such as iterative deepening [6], no node has information about the location of the desired data. In an informed search such as routing indices [8], each node keeps some metadata about the data location. To restrict the total bandwidth consumption, data queries in unstructured P2P systems may be terminated prematurely before the desired existing data is found; therefore, the query may not return the desired data even if the data actually exists in the system. An unstructured P2P network can not offer bounded routing efficiency due to lack of structure. Searching in a loosely structured system depends on the overlay structure and how the data is stored. In Freenet, searching is directed by the hints used for the overlay construction and the data storage. In Symphony, the data location is precisely defined but the overlay structure is probabilistically formed. Searching in Symphony is guided by reducing the numerical distance from the querying source to the destination node where the desired data is located. The loosely structured systems can offer a balanced trade-off if they are properly designed.

This chapter provides a survey of state-of-the-art searching schemes in different types of P2P systems. The survey focuses on searching schemes in unstructured P2Ps. The chapter is organized as follows. In section 2, searching in various unstructured systems will be explored. In section 3, searching in strictly structured systems will be investigated. The discussion in this section focuses on hierarchical DHT P2Ps and non-DHT P2Ps. Non-hierarchical DHT P2Ps are briefly

overviewed since a survey of searching in such systems has been done in [1]. Searching in loosely structured systems will be examined in section 4. A summary will be given in section 5.

2. Searching in unstructured P2Ps

In an unstructured P2P system, no rule exists that strictly defines where data is stored and which nodes are neighbors of each other. To find a specific data item, early work such as the original Gnutella [15] used flooding, which is the Breadth First Search (BFS) of the overlay network graph with depth limit D. D refers to the system-wide maximum TTL of a message in terms of overlay hops. In this approach, the *querying node* sends the query request to all its neighbors. Each neighbor processes the query and returns the result if the data is found. This neighbor then forwards the query request further to all its neighbors except the querying node. This procedure continues until the depth limit D is reached. Flooding tries to find the maximum number of results within the ring that is centered at the querying node and has the radius: D-overlay-hops. However, it generates a large number of messages (many of them are duplicate messages) and does not scale well.

Many alternative schemes have been proposed to address the problems of the original flooding. These works include iterative deepening [6], k-walker random walk [7], modified random BFS [10], two-level k-walker random walk [52], directed BFS [6], intelligent search [10], local indices based search [6], routing indices based search [8], attenuated bloom filter based search [9], adaptive probabilistic search [11], and dominating set based search [2]. They can be classified as BFS based or Depth First Search (DFS) based. The routing indices based search and the attenuated bloom filter based search are variations of DFS. All the others are variations of BFS. In the iterative deepening and local indices, a query is forwarded to all neighbors of a forwarding node. In all other schemes, a query is forwarded to a subset of neighbors of a forwarding node.

The searching schemes in unstructured P2P systems can also be classified as deterministic or probabilistic. In a deterministic approach, the query forwarding is deterministic. In a probabilistic approach, the query forwarding is probabilistic, random, or is based on ranking. The iterative deepening, local indices based search, and the attenuated bloom filter based search are deterministic. The others are probabilistic.

Another way to categorize searching schemes in unstructured P2P systems is regular-grained or coarse-grained. In a regular-grained approach, all nodes participate in query forwarding. In a coarse-grained scheme, the query forwarding is performed by only a subset of nodes in the entire network. Dominating set based search is coarse-grained because the query forwarding is performed only by the dominating nodes in the CDS (Connected Dominating Set). All the others are regular-grained.

Another taxonomy is blind search or informed search [11] [12]. In a blind search, nodes do not keep information about data location. In an informed search, nodes store some metadata that facilitates the search. Blind searches include iterative deepening, k-walker random walk, modified random BFS, and two-level k-walker random walk. All the others are informed search.

2.1 Iterative deepening

In [6], Yang and Garcia-Molina borrowed the idea of iterative deepening from artificial intelligence and used it in P2P searching. This method is also called *expanding ring*. In this technique, the querying node periodically issues a sequence of BFS searches with increasing depth limits $D_1 < D_2 < \ldots < D_i$. The query is terminated when the query result is satisfied or when the maximum depth limit D has been reached. In the latter case, the query result may not be

satisfied. All nodes use the same sequence of depth limits called *policy P* and the same time period *W* between two consecutive BFS searches.

For example, assume that $P = \{3, 5, 8\}$, $W = 6$ seconds. The query node *S* first sends a BFS search with depth limit 3 to all its neighbors via a *query message*. This BFS search message will reach all nodes within 3-hops distance from *S*. These nodes will process this BFS message and store (*freeze*) that message for a time period (> *W)* when they receive it. If any desired data is located on these nodes, the data will be sent back to *S*. If the query is satisfied within W (= 6) seconds following the first BFS search, *S* will terminate the query and will not continue. Otherwise, *S* will initiate the second BFS search with depth limit 5 via a *resend message*. The resend message carries the same query ID as in the corresponding query message. Any node within 2-hops distance from *S* will simply forward the resend message to all its neighbors after receiving it. The nodes at 3-hops distance from *S* will drop the resend message and then unfreeze the stored query message with the matching query ID. "Unfreeze" means forwarding the respective stored query message with a new depth limit 2 (? 5 ? 3) to all its neighbors. This unfrozen query message will be processed similarly to the query message in the first BFS search. If the resend message with maximum depth limit 8 is sent by the querying node, nodes within 8-hops distance from *S* will not store (freeze) this query message. The querying node will not issue another resend message with a larger depth limit.

Iterative deepening is tailored to applications where the initial number of data items returned by a query is important. However, it does not intend to reduce duplicate messages and the query processing is slow.

2.2 *k*-walker random walk and related schemes

In the *standard random walk* algorithm, the querying node forwards the query message to one randomly selected neighbor. This neighbor randomly chooses one of its neighbors and forwards the query message to that neighbor. This procedure continues until the data is found. Consider the query message as a walker. The query message is forwarded in the network the same way a walker randomly walks on the network of streets. The standard random walk algorithm uses just one walker. This can greatly reduce the message overhead but causes longer searching delay.

In the *k*-walker random walk algorithm [7], *k* walkers are deployed by the querying node. That is, the querying node forwards *k* copies of the query message to *k* randomly selected neighbors. Each query message takes its own random walk. Each walker periodically "talks" with the querying node to decide whether that walker should terminate. Nodes can also use soft states to forward different walkers for the same query to different neighbors. *k*-walker random walk algorithm attempts to reduce the routing delay. On average, the total number of nodes reached by *k* random walkers in *H* hops is the same as the number of nodes reached by one walker in *kH* hops. Therefore, the routing delay is expected to be *k* times smaller.

A similar scheme is the *two-level random walk* [52]. In this scheme, the querying node deploys k_1 random walkers with the TTL being l_1. When the TTL l_1 expires, each walker forges k_2 random walkers with the TTL being l_2. All nodes on the walkers' paths process the query. Given the same number of walkers, this scheme generates less duplicate messages but has longer searching delays than the *k*-walker random walk.

Another similar approach, called the *modified random BFS*, was proposed in [10]. The querying node forwards the query to a randomly selected subset of its neighbors. On receiving a query message, each neighbor forwards the query to a randomly selected subset of its neighbors

(excluding the querying node). This procedure continues until the query stop condition is satisfied. No comparison to the *k*-walker random walk was given in [10]. It is expected that this approach visits more nodes and has a higher query success rate than the *k*-walker random walk.

The works in [7] [41] also address the data replication issue in unstructured P2P systems. The question studied is: assuming the fixed amount of total storage space in the P2P system, what is the optimal number of copies for each object in terms of the average search overhead per successful query? Three replication strategies were analyzed: *uniform*, *proportional*, and *square-root replication*. In the uniform replication, the same number of copies is created for each object regardless of the query distribution. In the proportional replication, the number of copies for each object is proportional to its query distribution. The higher the query rate of an object , the higher is the number of copies for that object. In the square-root replication, the number of copies per object is proportional to the square-root of the query rate. The performance measures are the average search size (i.e. the average number of nodes probed) and the utilization rate of a copy (i.e. the rate of queries that a copy serves). The search size reflects the query efficiency. The utilization rate indicates the load balance. The *k*-walker random walk is used as the searching scheme in the evaluation.

The analysis and simulation results show that uniform replication and proportional replication achieve the same average search size and this search size is larger than that of the square-root replication. As for the utilization rate, the proportional replication has the same rate for all objects; the uniform replication has the rate proportional to the query rate and the square-root replication has a varying utilization rate per object. However, the square-root replication has much smaller variances than uniform and proportional replication in the two performance measures. In summary, the square-root replication has the best query efficiency and the proportional replication achieves the best load balance. In practice, the square-root replication is implemented by replicating copies proportional to the number of sites probed.

The work in [7] [41] also studies where to replicate an object. Three approaches are considered and evaluated using *k*-walker random walk: *owner replication*, *path replication*, and *random replication*. All three schemes replicate the found object when a query is successful. The owner replication replicates an object only at the requesting node. The path replication creates copies of an object on all nodes on the path from the providing node to the requesting node. The random replication places copies on the *p* randomly selected nodes that were visited by the *k* walkers. The path replication implements the square-root replication. The random replication has slightly less overall search traffic than the path replication, because path replication intends to create object copies on the nodes that are topologically along the same path. Both the path replication and the random replication have less overall search traffic than the owner replication.

2.3 Directed BFS and intelligent search

The basic idea of directed BFS approach [6] is that the query node sends the query message to a subset of its neighbors that will quickly return many high-quality results. These neighbors then forward the query message to all their neighbors just as in BFS.

To choose "good" neighbors, a node keeps track of simple statistics on its neighbors, for example, the number of query results returned through that neighbor, and the network latency of that neighbor. Based on these statistics, the best neighbors can be intelligently selected using the following heuristics:

- The highest number of query results returned previously
- The least hop-count in the previously returned messages (i.e. the closest neighbors)
- The highest message count (i.e. the most stable neighbors)

- The shortest message queue (i.e. the least busy neighbors)

By directing the query message to just a subset of neighbors, directed BFS can reduce the routing cost in terms of the number of routing messages. By choosing good neighbors, this technique can maintain the quality of query results and lower the query response time. However, in this scheme only the querying node intelligently selects neighbors to forward a query. All other nodes involved in a query processing still broadcast the query to all their neighbors as in BFS. Therefore, the message duplication is not greatly reduced.

A similar approach called *intelligent search* was presented in [10]. The query type considered in the work is the *keyword query*: a search for documents that contain desired keywords listed in a query. A query is represented using a keyword vector. This technique consists of four components: a search mechanism, a profile mechanism, a peer ranking mechanism, and a query-similarity function.

When the querying node initiates a query, it does not broadcast the query to all its neighbors. Instead, it evaluates the past performance of all its neighbors and propagates the query only to a subset of its neighbors that have answered similar queries before and therefore will most likely answer the current query. On receiving a query message, a neighbor looks at its local datastore. If the neighbor has the desired documents, it returns them to the querying node and terminates. Otherwise, the neighbor forwards the query to a subset of its own neighbors that have answered similar queries before. The query forwarding stops when the maximum TTL is reached.

The *cosine similarity model* is used to compute the query similarity. Based on this model, the similarity between two queries is the cosine of the angle between their query vectors. To determine whether a neighbor answered similar past queries, each node keeps a profile for each of its neighbors. The profile for a neighbor contains the most recent queries that were answered by that neighbor. The profile is created and updated using two schemes. In one scheme, each peer continuously monitors the query and query response message. Queries answered by a neighbor are stored in the profile for that neighbor. In the second scheme, the peer that replies to a query message broadcasts this information to all its neighbors.

Neighbors are ranked to facilitate the selection. The rank of a neighbor P_i of the peer P_j in terms of the query q is determined by the following formula:

$$R_{P_j}(P_i, q) = \sum_{q_l \in A_i} (Q_{sim}(q_l, q))^{\gamma}$$

In the formula, A_i denotes the set of queries among the K most similar ones that were answered by peer P_i; γ is a configurable parameter used to add more weight to more similar queries. The ranking formula aggregates the similarities of K most similar past queries answered by a neighbor.

2.4 Local indices based search

The *local indices* in [6] intends to get the same number of query results as scoped-flooding with less number of nodes processing a query. In local indices, each node keeps indices of data on all nodes within k-hop distance from it. Therefore, each node can directly answer queries for any data in its local indices without resorting to other nodes. All nodes use the same policy P on the list of depths at which the query should be processed. The nodes whose depths are listed in P check their local indices for the queried data and return the query result if the sought data is found. These nodes also forward the query message to all their neighbors if their depths are not the maximum depth limit. All other nodes whose depths are not listed in P just forward the query message to all their neighbors once receiving it and do not check their local indices. For example,

assume that $P = \{0, 3, 6\}$. To route a query, the querying node processes the query because its depth: 0 (i.e. the depth from itself is 0) is listed in P. The querying node then forwards the query message to all its neighbors at depth 1. Because their depth 1 is not listed in *P*, these nodes will not process the query. They will simply forward the query message to all their neighbors at depth 2. For the same reason, all nodes at depth 2 will simply forward the query message to all their neighbors at depth 3. All nodes at depth 3 will process the query because their depth is listed in *P*. These nodes then forward the query to their neighbors at depth 4. This procedure continues until the query message is forwarded to all nodes at depth 6. These nodes will process the query. However, they will not forward the query because their depth is the maximum depth in *P*. At this point, the query is terminated even if the query result is not satisfied. Note that all nodes in a P2P system organized using local indices play equal roles.

The local indices are updated when a node joins, leaves, or modifies its data. A node *Y* joins the network by sending a join message with a TTL of *r*. This join message contains the metadata (indices) about the data collection in *Y*. All nodes within *r*-hop distance from *Y* will receive this join message. If a node *X* receives the join message from *Y*, it replies with another join message that includes the metadata over its own data collection. *X* sends this replied join message directly to *Y* over a temporary connection. Then both *X* and *Y* add each other's metadata into their own local indices.

A new node *Y* may add a new path of length *k* or less between two other nodes *A* and *B*. These two nodes can discover this new path in a number of ways without introducing additional messages. One way to achieve this is through periodic ping-pong messages. Nodes constantly send ping messages to all nodes within a depth *D*. Every node replies with a pong message. If *A* receives a pong message from *B* which is at most *k* hops away and *A* does not contain indices about *B*'s data collection, then *A* learns that there is a new path between *A* and *B*. *A* will inform *B* about its data collection by sending a join message directly to *B*. *B* will reply directly to *A* with another join message containing the indices of its own data collection.

When a node *Z* gracefully leaves the network or fails, other nodes will detect this event after a timeout. If these nodes index *Z*'s data collection, they will remove those index entries. When the data collection on a node *Z* is modified, *Z* will send a short update message with a TTL of *r* to all its neighbors. This update message includes information about all affected data elements and how they are affected: inserted, deleted or updated. Any node that receives such a message and contains index entries for those affected elements will update their local indices accordingly.

The local indices approach is similar to iterative deepening. Both broadcast the query message based on a list of depths; however, in iterative deepening, all nodes within the maximum depth limit process the query. In local indices, only nodes whose depths are listed in the policy *P* process the query. In addition, the iterative deepening approach spreads the query message iteratively with increasing TTL; the local indices approach spreads the query message once with the maximum TTL.

2.5 Routing indices based search

Routing indices [8] is similar to directed BFS and intelligent search in that all of them use the information about neighbors to guide the search. Directed BFS only applies this information to selecting neighbors of the querying source (i.e. the first hop from the querying source.) The rest of the search process is just as that of BFS. Both intelligent search and routing indices guide the entire search process. They differ in the information kept for neighbors. Intelligent search uses information about past queries that have been answered by neighbors. Routing indices stores information about the topics of documents and the number of documents stored in neighbors.

Routing indices considers content queries, queries based on the file content instead of file name or file identifier. One example of such a content query is: a request for documents that contain the word "networks". A query includes a set of subject topics. Documents may belong to more than one topic category. Document topics are independent. Each node maintains a local index of its own document database based on the keywords contained in these documents.

The goal of a *Routing Index* (RI) is to facilitate a node to select the "best" neighbors to forward queries. A RI is a distributed data structure. Given a content query, the algorithms on this data structure compute the top *m* best neighbors. The goodness of a neighbor is application dependent. In general, a good neighbor is the one through which many documents can be quickly found.

A routing index is organized based on the single-hop routes and document topics. There is one index entry per route (i.e. per neighbor) per topic. An RI index entry, *(networks,B)*, at node *A* stores information about documents in the topic: *networks* that may be found through the route (*A -> B*). This entry gives *hints on the potential query result* if *A* forwards the query to *B* (i.e. the route *A -> B* is chosen). Hence the name Routing Index. A routing index entry is very different from a regular index entry. If *(networks, B)* were the regular index entry, it would mean that node *B stores* documents in the topic: *networks*. By organizing the index based on neighbors (routes) instead of destinations (indexed data locations), the storage space can be reduced.

Three types of RIs, *compound RI, hop-count RI, and exponentially aggregated RI,* are proposed. They differ in RI index entry structures. A compound RI (CRI) stores information about the number of documents in each interesting topic that might be found if a query is forwarded to a single-hop neighbor. A sample CRI at a node *B* is shown in Table 1. Each row in the table describes the number of documents along a specific path and the number of documents on each interesting topic along that path. For example, the first row in the table indicates that if *B* forwards the query to *A*, 1000 documents may be found. Among those documents, 100 are DB documents, 200 are network documents, 400 are theory documents, and there are no language documents.

Path	#docs	Documents in topics			
		Database (DB)	Networks (N)	Theory (T)	Languages (L)
A	1000	100	200	400	0
E	300	60	0	200	100
F	800	0	100	160	200

Table 1. An example of a compound RI at node *B*.

The goodness of a neighbor for a query in CRI is the number of desired documents that may be found through that neighbor. This can be estimated by the following formula:

$$ND \cdot \prod_i \frac{CRI(t_i)}{ND}$$

In the formula, t_i refers to the subject topic that appears in both the query and the CRI table; $CRI(t_i)$ denotes the value in the intersection of the row for a path and the column for the topic t_i; *ND* represents the value at the column *#docs* for the path considered. Use the CRI example for node *B* in Table 1. Assume that *B* receives a query for documents on "networks" and "theory". The goodness of each neighbor for the query is:

A: 1000 x (200/1000) x (400/1000) = 80.
E: 300 x (0/300) x (200/300) = 0.

F: 800 x (100/800) x (160/800) = 20.

Therefore, *B* will select *A* to forward the query because its goodness score is the highest.

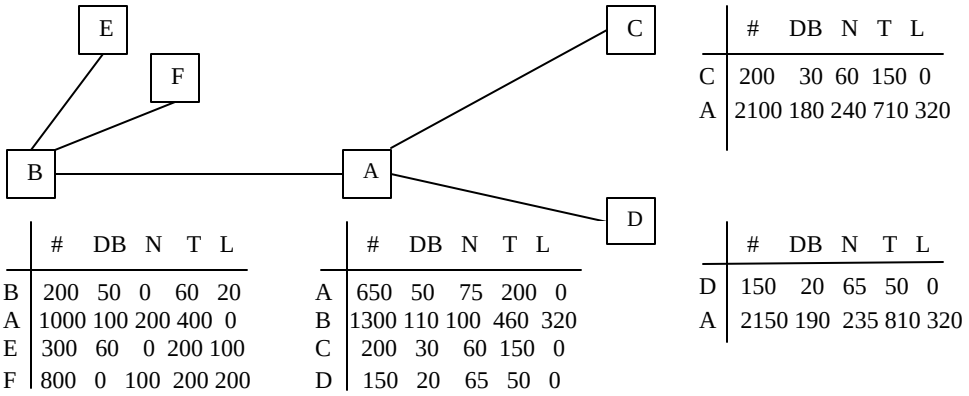


Figure 1. A partial P2P with CRI indices.

Figure 1 shows a partial P2P network and some CRI indices. An additional row is added into the CRI at each node to summarize the local indices in that node. For example, the summary at node *B* indicates that there are 200 documents at *B*; 50 of them are related to database, 60 of them are about theory, and 20 of them are about languages. *B* does not store documents about networks. The CRIs at node *B*, *A*, *C*, and *D* show that node *B* can access 200 network documents via *A*. 75 of them are at *A*, 60 at *C*, and 65 at *D*.

The following shows an example of searching using routing indices. Suppose that the node *B* initiates a query for the documents about "networks" and "theory". *B* first looks up its local database for the desired documents. If not enough documents are found, it calculates the goodness scores of all its neighbors: *A*: 80; *E*: 0; *F*: 20. *A* is then chosen as the best neighbor to forward the query. After receiving the query, *A* first checks its local database and returns all desired documents to *B*. If the query result is not satisfied, *A* will then calculate the goodness scores of its neighbors *C*, *D* (*B* is excluded): *C*: 45, *D*: 23. *A* then selects *C* as the best neighbor to forward the query. *C* then processes the query and returns all desired data along the query path. *C* does not have any other neighbor to forward the query. If the query stop condition is not satisfied, *C* will return the query back to *A*. *A* then forwards the query to its second best neighbor *D*. This process continues until the desired number of documents is found.

The CRIs are expanded as follows. When a new connection is established between nodes *A* and *D*, *A* will add up its RI vectors (rows) and then sends this aggregated RI vector to *D*. In the mean time, *D* also sums up its RI vectors (excluding *A*'s entry if it exists), and sends the aggregated RI to *A*. When either party receives the other's aggregated RI, it will create a new entry in its RI for the other party. After this, both *A* and *D* inform their other neighbors about this change in a similar fashion. The CRI entry deletion and update are handled similarly. RI entry aggregation reduces the bandwidth overhead.

The compound RI does not consider the number of hops required to reach documents of a specific topic. However, we can modify the CRI to incorporate the hop count. We can store a CRI for each hop up to a maximum hop limit *H* at each node. *H* is called the *horizon* of a RI. This modified CRI is called *hop-count Routing Indices*. The hop-count RI contains information about the non-cumulative number of documents that may be found along a path at 1-hop distance, at 2-hop distance, …, at *H*-hop distance. The goodness of a neighbor with respect to a query in the hop-count RI is the number of desired documents per message. It considers both the document

counts and the number of messages to reach those documents. The goodness score is computed using the regular-tree cost model.

The limitation of the hop-count RI is that it does not have information about documents at hop-distance beyond the horizon. The *exponentially aggregated RI (ERI)* solves this problem at the cost of some potential loss in accuracy. The ERI entries store the result of applying the regular-tree cost formula to a corresponding hop-count RI for the topics of interest.

2.6 Attenuated bloom filter based search

The attenuated bloom filter based search [9] assumes that each stored document has many replicas spread over the P2P network; documents are queried by names. It intends to quickly find replicas close to the query source with high probability. This is achieved by approximately summarizing the documents that likely exist in nearby nodes. However, the approach alone fails to find replicas far away from the query source.

Bloom filters [50] are often used to approximately and efficiently summarize elements in a set. A bloom filter is a bit-string of length *m* that is associated with a family of independent hash functions. Each hash function takes as input any set element and outputs an integer in [0,m). To generate a representation of a set using bloom filters, every set element is hashed using all hash functions. Any bit in the bloom filter whose position matches a hash function result is set to 1. To determine whether an element is in the set described by a bloom filter, that element is hashed using the same family of hash functions. If any matching bit is not set to 1, the element is definitely not in the set. If all matching bits in the bloom filter are set to 1, the element is *probably* in the set. If the element indeed is not in the set, this is called a *false positive*.

Attenuated Bloom Filters are extensions to bloom filters. An attenuated bloom filter of depth *d* is an array of *d* regular bloom filters of the same length *w*. A level is assigned to each regular bloom filter in the array. Level 1 is assigned to the first bloom filter. Level 2 is assigned to the second bloom filter. The higher levels are considered to be attenuated with respect to the lower levels. Each node stores an attenuated bloom filter for each neighbor. The *ith* bloom filter in an attenuated bloom filter (depth: *d*; $i = d$) for a neighbor *B* at a node *A* summarizes the set of documents that will probably be found through *B* on all nodes i-hops away from *A*. Figure 2 illustrates an attenuated bloom filter for neighbor *C* at node *B*. "File3" and "File4" are available at 2-hops distance from *B* through *C*. They are hashed to {0, 5, 6} and {2, 5, 8} respectively. Therefore, the second bloom filter contains 1 at bits 0, 2, 5, 6, 8.

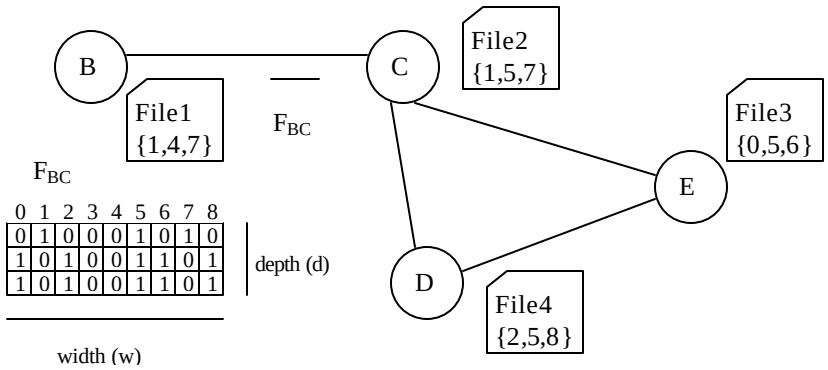


Figure 2. An example of an attenuated bloom filter.

To route a query for a file, the querying node hashes the file name using the family of hash functions. Then the querying node checks level-1 of its attenuated bloom filters. If level-1 of an attenuated bloom filter for a neighbor has 1s at all matching positions, the file will probably be found on that neighbor (1-hop distance from the query source). We call such a neighbor a candidate. The querying node then forwards the query to the closest one among all candidates. If no such candidate can be found, the querying node will check the next higher level (level-2) of all its attenuated bloom filters similarly to checking level-1. If no candidate can be found after all levels have been checked at the query source, this indicates that definitely no nearby replica exists. On receiving the query, a neighbor of the querying node looks up its local data store. If the data is found, it will be returned to the query source. If not, this neighbor will check its attenuated bloom filters similarly. During the query processing, if a false positive is found after *d* (the depth of the attenuated bloom filter) unsuccessful hops, the attenuated bloom filter based search terminates with a failure. No back tracking is allowed.

To ease the filter update operation, for any two neighboring nodes *A* and *B*, node *A* keeps a copy of the attenuated bloom filter at *B* for the link *B* -> *A*. Node *B* also keeps a copy of the attenuated bloom filter at *A* for the link *A* -> *B*. If a new document is inserted at node *A*, it calculates the changed bits in the attenuated bloom filters of its own and of its neighbors. *A* then sends the changes to the corresponding neighbors. When *A*'s neighbor *B* receives such a message, *B* will attenuate the changed bits one level and check changes in the attenuated bloom filters which its neighbors maintain. *B* will inform its neighbors about the changes as well. Thus, each update is spread outward from the update source. The duplicate update messages can be suppressed by either the source node or the destination node with the help of update message IDs.

The attenuated bloom filter approach can be combined with any structured approach to optimize the searching performance. We can use the attenuated bloom filters to try locating nearby replicas. If no nearby replica exists, we switch to the structured approach to continue the lookup. The hop-count RI is similar to the attenuated bloom filter approach. Both summarize the documents at some distance from the querying source. There are two differences between them. One is that the attenuated bloom filter is a probabilistic approach while the hop-count RI is a deterministic approach if omitting the document change. The other is that the attenuated bloom filter provides information about a specific file while the hop-count RI provides the number of documents on each document category but not a specific file.

2.7 Adaptive probabilistic search

In the *Adaptive Probabilistic Search (APS)* [11] [12], it is assumed that the storage of objects and their copies in the network follows a replication distribution. The number of query requests for each object follows a query distribution. The search process does not affect object placement and the P2P overlay topology.

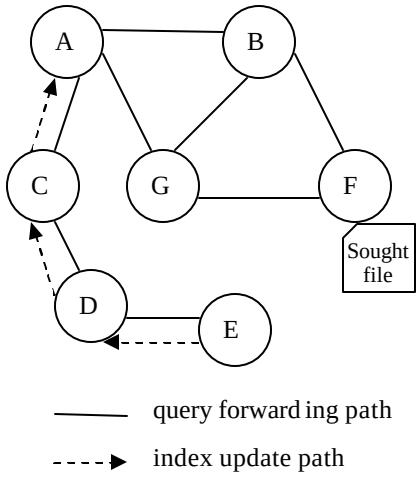
The APS is based on *k*-walker random walk and *probabilistic* (not *random*) forwarding. The querying node simultaneously deploys *k* walkers. On receiving the query, each node looks up its local repository for the desired object. If the object is found, the walker stops successfully. Otherwise, the walker continues. The node forwards the query to the best neighbor that has the highest probability value. The probability values are computed based on the results of the past queries and are updated based on the result of the current query. The query processing continues until all *k* walkers terminate either successfully or fail (in which case the TTL limit is reached).

To select neighbors probabilistically, each node keeps a local index about its neighbors. There is one index entry for each object which the node has requested or forwarded requests for through each neighbor. The value of an index entry for an object and a neighbor represents the relative

probability of that neighbor being selected for forwarding a query for that object. The higher the index entry value the higher the probability. Initially, all index values are assigned the same value. Then, the index values are updated as follows. When the querying node forwards a query, it makes some guess about the success of all the walkers. The guess is made based on the ratio of the successful walkers in the past. If it assumes that all walkers will succeed (*optimistic approach*), the querying node pro-actively increases the index values associated with the chosen neighbors and the queried object. Otherwise (*pessimistic approach*), the querying node pro-actively decreases the index values. Using the guess determined by the querying node, every node on the query path updates the index values similarly when forwarding the query.

The index values are also updated when the guess for a walker is wrong. Specifically, if an optimistic guess is made and a walker terminates with a failure, then the index values for the requested object along that walker's path are decreased. The last node on the path sends an update message to the preceding node. On receiving the message, the preceding node decreases the index value for that walker and forwards the update message to the next node on the reverse path. This update procedure continues on the reverse path until the querying node receives an update message and decreases the index value for that walker. If the pessimistic approach is employed and a walker terminates successfully, the index values for the requested object on the walker's path are increased. The update procedure is similar. To remember a walker's path, each node appends its ID in the query message during query forwarding and maintains a soft state for the forwarded query. If a walker *A* passes by a node which another walker *B* stopped by before, the walker *A* terminates unsuccessfully. The duplicate message was discarded.

Figure 3 illustrates how the search process works. Peer *A* issues a query for an object stored on peer *F*. Two walkers are deployed. Peer *A* made an optimistic guess. The initial values of all index entries for this object are 30. One walker w_1 takes the path *A* -> *B* -> *F*. The other one w_2 takes the path: *A* -> *C* -> *D* -> *E*. During the search, each node except the last node on the query paths increases the index value(s) for this object and the chosen neighbor(s) by 10. Since the optimistic approach is employed and w_2 fails, the index values on the query path for w_2 will be decreased by 20 so that the final index values are smaller than the initial index values. When the subsequent request for the same object is initiated at or forwarded to *A*, the neighbor *B* will be chosen with the probability 4/9 (40/(20+30+40)), *C* with the probability 2/9, and *G* with the probability 3/9.



Indices	Initially	At walker termination	After index updates
A ✍ B	30	40	40
B ✍ F	30	40	40
A ✍ C	30	40	20
C ✍ D	30	40	20
D ✍ E	30	40	20
A ✍ G	30	30	30

Figure 3. An example of adaptive probabilistic search.

Compared to the *k*-walker random walk, the APS approach has the same asymptotic performance in terms of the message overhead. However, by forwarding queries probabilistically to most promising neighbor(s) based on the learned knowledge, the APS approach surpasses the *k*-walker random walk in the query success rate and the number of discovered objects.

Two performance optimizations of the APS were also proposed in [11] [12]. The APS uses the same guess for all objects. This imprecision causes more messages. The *swapping-APS* (*s*-APS) therefore constantly observes the ratio of successful walkers for each object and swaps to a better update policy accordingly. The *weighted-APS* (*w*-APS) includes the location of objects in the probabilistic selection of neighbors. A distance function is embedded in the stored path of the query and is used in the index update. When the pessimistic guess is made for a walker and the walker succeeds, the index values for neighbors closer to the discovered object are increased more than those for distant neighbors.

2.8 Dominating set basedsearch

The *dominating set based search* scheme was proposed in [2]. In this approach, routing indices are stored in a selected set of nodes that form a connected dominating set (CDS). A CDS in a P2P network is a subset of nodes which are connected through direct overlay links. All other nodes that are not in the CDS can be reached from some node in the CDS in one-hop. Searching is performed through a random walk on the dominating nodes in the CDS.

The construction of the CDS uses solely the local information: a node's 1-hop and 2-hop neighbors. The construction consists of two processes: *marking* followed by *reduction.* The marking process marks each node in the P2P system as either a dominating node or a non-dominating node. The marker *T* represents a dominating node while the marker *F* represents a non-dominating node. A node is marked using *T* if two of its neighbors are not directly connected (i.e. these two neighbors are not neighbors of each other). At the end of the marking process, all nodes with marker *T* form the CDS. To reduce the size of the CDS, two reduction rules are applied during the reduction process. Each node in the CDS is assigned a 1-hop ranking value. This ranking value is the sum of the number of documents on a node and the number of documents of the node's neighbor that has the most documents. The first reduction rule specifies that if the neighbors of a node *A* in the CDS are a proper subset of neighbors of another node *B* in the CDS and the node *A* has a smaller 1-hop ranking value than node *B*, then remove node *A* from the CDS. The second reduction rule states that a node *C* is removed from the CDS if the following three conditions are satisfied: 1) Two neighbors *A* and *B* of the node *C* are also dominating nodes. 2) The neighbor set of *C* is a proper subset of the union of the neighbor sets of *A* and *B*. 3) The node *C* has a 1-hop ranking value that is smaller than the values of both *A* and *B*.

Searching is conducted on the CDS as follows. If the querying source is not a dominating node, the source forwards the query to its dominating neighbor with the highest 1-hop ranking value. If the querying source is a dominating node, it forwards the query to its dominating neighbor with the highest 1-hop ranking value. This querying source also forwards the query to a non-dominating neighbor if that neighbor has the most documents among all neighbors of the querying source. On receiving a query request, a dominating node looks up its local database for the searched document and performs the query forwarding similarly to a querying source that is a dominating node. On receiving a query request, a non-dominating node only looks up the local database and does not forward the query any further. All found documents are returned from the hosting nodes to the querying source along the reverse query paths. The query stops when the TTL limit is reached or a node is visited the second time.

The dominating set based approach intends to get the most number of documents by forwarding queries primarily on dominating nodes which are well-connected and have many documents themselves or whose neighbors have many documents. The construction of the CDS does not incur more overlay links, as often occurs in superpeer approaches to be discussed in Section 3.2.The cost of creating and maintaining the CDS is lower than that of routing indices.

3. Searching in strictly structured P2Ps

In a strictly structured system, the neighbor relationship between peers and data locations is strictly defined. Searching in such systems is therefore determined by the particular network architecture. Among the strictly structured systems, some implement a distributed hash table (DHT) using different data structures. Others do not provide a DHT interface. Some DHT P2P systems have flat overlay structures; others have hierarchical overlay structures.

A DHT is a hash table whose table entries are distributed among different peers located in arbitrary locations. Each data item is hashed to a unique numeric key. Each node is also hashed to a unique ID in the same key space. Each node is responsible for a certain number of keys. This means that the responsible node stores the key and the data item with that key or a pointer to the data item with that key. Keys are mapped to their responsible nodes. The searching algorithms support two basic operations: *lookup(key)* and *put(key)*. *lookup(k)* is used to find the location of the node that is responsible for the key *k*. *put(k)* is used to store a data item (or a pointer to the data item) with the key *k* in the node responsible for *k*. In a distributed storage application using a DHT, a node must publish the files that are originally stored on it before these files can be retrieved by other nodes. A file is published using *put(k)*.

In this section, searching in non-hierarchical (flat) DHT P2Ps is briefly overviewed. Then searching in hierarchical DHT P2Ps and non-DHT P2Ps are discussed in detail. More about non-hierarchical DHT P2Ps can be found in a comprehensive survey in [1].

3.1 Searching in non-hierarchical DHT P2Ps

Different non-hierarchical DHT P2Ps use different flat data structures to implement the DHT. These flat data structures include ring, mesh, hypercube, and other special graphs such as de Bruijn graph. Chord [16] uses a ring data structure. Node IDs form a ring. Each node keeps a finger table that contains the IP addresses of nodes that are half of the ID ring away from it, one-fourth of the ID ring away, one-eighth of the ID ring away, ..., until its immediate successor. A key is mapped to a node whose ID is the largest number which does not exceed that key. During the searching for *lookup(k)*, a node *A* forwards the query for *k* to *successor(k)*, which is another node in *A*'s finger table with the highest ID that is not larger than *k*. In this way, the query for *k* is forwarded through the successor list until the node responsible for *k* is reached. The finger table speeds up the lookup operation. In case of the failure of *successor(k)*, a node forwards the query to its immediate successor node. Chord achieves $O(\log N)$ routing efficiency at the cost of $O(\log N)$ routing state per node. *N* refers to the total number of nodes in the system. The work in [22] extends Chord by adding different kinds of reverse edges into Chord so that the modified Chord is resilient to routing attacks.

Pastry [17] uses a tree-based data structure which can be considered as a generalization of a hypercube. The node ID is 128-digit in base 2^b. *b* is typically 4. Each node *A* keeps a leaf set *L*. *L* consists of the set of $|L|/2$ nodes whose IDs are closest to and smaller than *A*'s ID and the set of $|L|/2$ nodes whose IDs are closest to and larger than *A*'s ID. This leaf set guarantees the correctness of routing. To shorten the routing latency, each pastry node also keeps a routing table

of pointers to other nodes in the ID space. Each node keeps $(2^b ? 1)$ entries for each prefix of its node ID. An entry for a prefix of length i stores the location of some node whose ID shares that prefix and whose $(i ? 1)^{th}$ digit is different.

The searching in Pastry is done as follows. Given a query for the key k, a node A forwards the query to a node whose ID is numerically closest to k among all nodes known to A. The node A first tries to find a node in its leaf set. If such node does not exist, the node A tries to find a node in its routing table whose ID shares a longer prefix with k than A. If such node does not exist either, the node A forwards the query to a node whose ID has the same shared prefix as A but is numerically closer to k than A. Network proximity can be considered using heuristics during query forwarding in Pastry. Each Pastry node maintains $O(\log^N)$ routing state to achieve the routing latency $O(\log^N)$. The algorithms in Tapestry [18] and Kademlia [19] are similar to Pastry.

A d-dimensional toroidal space is used to implement the DHT in CAN [20]. The space is divided into a number of zones. Each zone is a hyper-rectangle and is taken care of by a node. The zone boundaries identify the node responsible for that zone. A key k is hashed to a point p in the d-dimensional space. The node whose zone covers p stores the hash table entry for k. Each node's routing table consists of all its neighbors in the d-dimensional space. A node A is considered as a neighbor of another node B if B's zone shares a $(d\text{-}1)$-dimensional hyperplane with A's zone. Given a query for the data item with key k, a node forwards the query to another node in its routing table whose zone is closest to the zone of the node responsible for the key k. Ties are broken arbitrarily. Each CAN node maintains $O(d)$ states to achieve $O(d\sqrt[d]{N})$ routing efficiency, where N refers to the total number of nodes in the P2P.

The systems Koorde [21], Viceroy [47], and Cycloid [48] have overlays with constant degrees. Koorde embeds a de Bruijn graph on the Chord ring for forwarding lookup requests. A routing efficiency of $O(\log^N)$ can be achieved with $O(1)$ state per node. The overlay of Viceroy is an approximate butterfly network. The node ID space is [0, 1). The butterfly level parameter of a node is selected according to the estimated network size. Viceroy also achieves $O(\log^N)$ routing efficiency with $O(1)$ neighbors per node. Cycloid integrates Chord and Pastry and imitates the cube-connected-cycles (CCC) graph routing. It has a routing efficiency of $O(d)$ with a routing state per node of $O(1)$. The simulation results in [48] show that Cycloid performs better than Koorde and Viceroy in large-scale and dynamic P2P systems.

3.2 Searching in hierarchical DHT P2Ps

All hierarchical DHT P2Ps organize peers into different groups or clusters. Each group forms its own overlay. All groups together form the entire hierarchical overlay. Typically the overlay hierarchies are two-tier or three-tier. They differ mainly in the number of groups in each tier, the overlay structure formed by each group, and whether or not peers are distinguished as regular peers and *superpeers/dominating nodes*. Superpeers/dominating nodes generally contribute more computing resources, are more stable, and take more responsibility in routing than regular peers. The discussion in this subsection focuses on Kelips and Coral.

3.2.1 Kelips

Kelips [23] is composed of k virtual *affinity groups* with group IDs in [0, k-1]. The IP address and port number of a node n is hashed to a group ID of the group to which the node n belongs. The

consistent hashing function SHA-1 provides a good balance of group members with high probability. Each file name is mapped to a group using the same SHA-1 function. Inside a group, a file is stored in a randomly chosen group member, called the file's *homenode*. Thus Kelips offers load balance in the same group and among different groups.

Each node *n* in an affinity group *g* keeps in the memory the following routing state:

(1) View of the belonging affinity group *g*:
This is the information about the set of nodes in the same group. The data includes the round-trip time estimate, the heartbeat count, etc.

(2) Contacts of all other affinity groups:
This is the information about a small constant number of nodes in all other groups. The data for each contact is the same as that of an intra-group node.

(3) Filetuples:
This is the intra-group index about the set of files whose homenodes are in the same affinity group. A file tuple consists of a file name and the IP address of the file's homenode. A heartbeat count is also associated with a file tuple.

The total number of routing table entries per node is $N / k ? c ? (k ? 1) ? F / k$, where N refers to the total number of nodes, c: the number of contacts per group, F: total number of files in the system, and k: the number of affinity groups. Assume that F is proportional to N and c is fixed. With the optimal k, the complexity of the routing state is $O(\sqrt{N})$.

To look up a file *f*, the querying node A in the group G hashes the file to the file's belonging group G'. If G' is the same as G, the query is resolved by checking the node A's local data store and local intra-group data index. Otherwise, A forwards the query to the topologically closest contact in group G'. On receiving a query request, the contact in the group G' searches its local data store and local intra-group data index. The IP address of *f*'s homenode is then returned to the querying node directly. In case of a file lookup failure, the querying node retries using different contacts in the group G', a random walk in the group G', or a random walk in the group G. The query is processed in $O(1)$ time with $O(1)$ message complexity.

To insert a file *f*, the *origin node* hashes the file name to the belonging group G. After looking up the routing table, the origin node sends an insert request to the topologically closest contact in the group G. A node in the group G is randomly chosen by this contact to be the homenode of the file *f*. This contact forwards the insert request to the chosen homenode. The file is then transferred from the origin node to the homenode. A new file tuple for the file *f* is created and added to the states of other nodes in group G. The failure of a file insertion is handled similarly to a file lookup failure. The file insertion is also done in $O(1)$ time with $O(1)$ message overhead.

All existing routing states are periodically updated using the spatially weighted gossip scheme within a group and across groups. Any timed-out entries are deleted. An update such as the heartbeat count for a file tuple starts at the responsible node. This node gossips the update for a number of fixed time intervals. During each time interval, the update message is multicast to a small constant number of gossip target nodes. The target nodes are chosen using a weighted scheme based on the round-trip time estimates. The preferences are given to those that are topologically closer in the network.

When a new node joins Kelips, it contacts a well-known introducer node (or group). The new node then uses the introducer's routing table to create its own routing table. The new node then announces its presence through gossiping. Contacts may be replaced either proactively or

reactively taking into account node distance and accessibility. Currently, a proactive approach is used to replace the farthest contact.

3.2.2 Coral and related schemes

Coral in [24] is an indexing scheme. It does not dictate how to store or replicate data items. The objectives of Coral are to avoid hot spots and to find nearby data without querying distant nodes. A distributed sloppy hash table was proposed to eliminate hot spots. In DHT, a key is associated with a single value which is a data item or a pointer to a data item. In a DSHT, a key is associated with a number of values which are pointers to replicas of data items. DSHT provides the interface: *put(key, value)* and *get(key)*. *put(key,value)* stores a value under a key; *get(key)* returns a subset of values under a key. There is a quota on the number of values associated with a particular key stored per node. When this quota is exceeded, the additional values are distributed across multiple nodes on the lookup path.

Specifically, when a file replica is stored locally on a node *A*, the node *A* hashes the file name to a key *k* and inserts a pointer *nodeaddr* (*A*'s address) to that file into the DSHT by calling *put(k,nodeaddr)*. During the processing of *put(k,nodeaddr)*, the node *A* finds the first node whose list of values under the key *k* is full or the first node that is closest to key *k*. If a node with a full-list is found, the node *A* goes back one hop on the lookup path. This previous node appends the pointer *nodeaddr* together with a timestamp to the end of its list under the key *k*. To query for a list of values for a key *k*, *get(k)* is forwarded in the identifier space until the first node storing a list for the key *k* is found. The requesting node can then download data from the list of nodes obtained. The unique "spill-over" scheme in Coral inserts pointers along the lookup path for popular keys. The hot spots are removed because load is balanced during pointer insertion and retrieval and data downloading.

To find nearby data without going through distant nodes, Coral organizes nodes into a hierarchy of clusters and puts nearby nodes in the same cluster. Coral consists of three levels of clusters. Each cluster is a DSHT. In the lowest-level, Level 2, there are many clusters that cover peers located in the same region and have the cluster diameter (round-trip time) 30msecs. In the next higher level, Level 1, there are multiple clusters that cover peers located in the same continent and have the cluster diameter 100msecs. The highest level, Level 0, is a single cluster for the entire planet and the cluster diameter is infinite. Each cluster is identified by a cluster id. Coral's hierarchy is built on top of Chord. Each cluster is a Chord ring that is composed of a different set of peers. The cluster at Level 0 is the original Chord ring. Each node belongs to one cluster at each level and has the same node id in all clusters to which it belongs.

A node inserts a key/value pair into Coral by performing a *put* on all of its clusters. To retrieve a key *k*, the querying node *A* first looks in its lowest level cluster. If the query fails in this level, the node *B* in the same cluster whose id is closest to the key *k* is reached. The node *B* returns its routing information in Level 1 to *A*. The node *A* then continues the search on its Level-1 cluster starting with the closest Level-1 node *C* in *B*'s routing table. If the query fails again, *A* will continue the search in the global cluster beginning with the closest Level-0 node *E* in *C*'s routing table. The query latency is therefore reduced by resolving a query from nearby nodes to distant nodes. The query hop count is still $O(\log^{N})$, where *N* is the total number of nodes in the system.

A node only joins *acceptable* clusters. A cluster is acceptable to a node if its latency (round-trip time) to 90% of the nodes in the cluster is below the cluster diameter. If a node can not find such clusters, it forms its own cluster. A node first joins a lowest level cluster. Then the node inserts itself to its higher-level cluster under the hash key of the IP addresses of its gateway routers.

When a node switches to a new cluster, its information is still kept in the old cluster. When old neighbors contact this node, it replies with the new cluster information. These members in the old cluster found the new cluster with more nodes and the same diameter. They will then switch to this larger cluster. The cluster split is implemented by guiding the split toward two directions. Some node in the cluster *c* is chosen as the *cluster center*. The nodes that are close to *c* form one cluster. The nodes far away from *c* form another cluster.

The HIERAS in [38] is similar to Coral. They differ in three aspects. Firstly, HIERAS supports DHT while Coral supports DSHT. Secondly, a HIERAS node joins the P2P hierarchy from the top level to the lowest level while a Coral node joins the hierarchy in an opposite way. Thirdly, HIERAS employs distributed binning to determine nodes in each Chord ring while Coral uses ping-pong messages to get latencies for determining peers in the same cluster (a Chord ring).

Another work similar to Coral was proposed in [48]. The overlay is also a hierarchy of Chord-like rings. The hierarchy emulates the nodes' real-world organization. Each Chord ring corresponds to an administrative domain. It requires that each node knows its own position in the hierarchy, and two nodes are able to compute their common ancestor in the hierarchy. The overlay hierarchy is formed in a bottom-up manner. All nodes in each leaf domain form their own overlay, a Chord ring. The overlay for a domain in the next higher-level is formed by merging the overlays for its child domains. The merging of two Chord rings is conducted as follows. Each node keeps all neighbors in its original Chord ring. In addition, each node *A* in one Chord ring adds another node *B* in the other Chord ring into its neighbor set if the following two conditions are satisfied: 1) *B* is the closest node which is at least 2^k away from *A* in the node ID space, where $0 ? k ? m$ (Node IDs are *m*-bit numbers). 2) *B* is closer to *A* than *A*'s immediate successor in *A*'s original Chord ring. The query routing in [48] is performed from the bottom level to the higher level in the hierarchy similarly to Coral.

3.2.3 Other hierarchical DHT P2Ps

In Kelips and Coral, all peers play equal roles in routing. The differences among peers, such as processing power and storage capacity, are not considered. The work in [25] takes into account peer heterogeneity such as CPU power and storage capacity. The nodes with more contributed resources are called *superpeers*. Otherwise, they are called *peers*. A superpeer may be demoted to a peer. A peer may also become a superpeer. The system architecture consists of two rings: an outer ring and an inner ring. The outer ring is a Chord ring and consists of all peers and all superpeers. The inner ring consists of only superpeers. Each superpeer is responsible for an arc in the outer ring. Each superpeer *sp* maintains a peer table and a superpeer table. The peer table contains the node ID and address of each peer in the *sp*'s managed arc. The superpeer table stores the node ID and managed arc range of each superpeer. The routing state is in the order of $O(\log^N)$, where *N* is the total number of nodes in the system.

To look up a document with the key *k*, the querying node first sends the query to its superpeer. If the key *k* is in the superpeer's managed arc, this superpeer locates and returns the successor of *k* to the querying node. Otherwise, the superpeer checks its superpeer table and forwards the query to another superpeer whose arc includes *k*. This second superpeer then looks up its peer table and returns the successor of *k* to the querying node. The lookup cost is $O(1)$.

To support the superpeer selection, the system uses a *volunteer service* to keep track of resources each node is willing and able to contribute to the system. Each new node registers its resources with the volunteer service. The volunteer service is provided as a black box.

A new node first joins the outer ring just as in Chord [16] and obtains its superpeer from its immediate neighbor. The new node then informs its superpeer to add a new entry for itself into the peer table. Unless selected as a superpeer later, this new node remains as a peer and stays in the outer ring in its life time. The peer failure is detected through periodic keep-alive messages between peer neighbors. The neighbor peer detecting a peer failure notifies its superpeer to remove the corresponding entry from the peer table. Superpeer failures are detected similarly. In case of a superpeer failure, the load of the failed superpeer may be taken over by newly created superpeers or existing neighbor superpeers. The actual load failover scheme is determined by the arc range of the failed superpeer. All changes to the inner ring topology are distributed to all superpeers.

The work in [26] also considers peer heterogeneity. However, the criterion for the superpeer selection is different. The selection in [26] primarily considers nodes with longer uptime and better connection, secondarily CPU power and network bandwidth. The hierarchy in [26] is also somewhat different. It contains 2 tiers. Peers form disjoint groups in the lower tier based on the network latency. Each group has its own overlay structure like Chord or CAN. A small number of peers in each group are chosen as superpeers for that group. All superpeers in the system form a separate overlay: a Chord ring in the top tier. Each "node" in the top-tier ring refers to all superpeers in a group and is represented by a vector. Given a query for the key *k*, the querying node first tries to look up the key in the lower tier. If the key is not found, the querying node sends the query to one of the superpeers in its group. This superpeer routes the query on the top-tier overlay towards the group that is responsible for the key *k*. After passing one or more superpeers, the query reaches one superpeer in the responsible group. This superpeer routes the query to the node closest to *k* in its own group in a similar way to the routing in a regular Chord ring.

KaZaA [27] also employs a two-tier hierarchy. It chooses nodes with fastest Internet connections and best CPU power as *supernodes*. A supernode indexes the files in its managed groups. In the literature, it is not clear what type of structure is formed by supernodes. In Brocade [28], all peers in the system form an overlay. Some peers in this overlay that have significant processing power, minimal number of IP hops to the wide-area network, and high-bandwidth outgoing links are chosen as supernodes. Each supernode acts as a landmark node for a network domain. Each supernode keeps a list of nodes in its managed domain. All supernodes form a Tapestry overlay on top of the base overlay. During query routing, the supernode of the querying source determines whether the query can be resolved in the local domain or not. If not, the supernode will route the query on the supernode overlay to the supernode of the node responsible for the sought key.

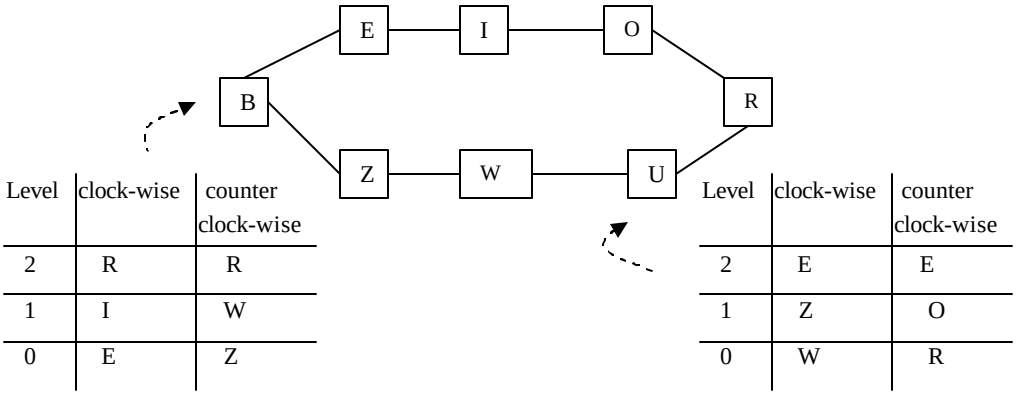
3.3 Searching in non-DHT P2Ps

The non-DHT P2Ps try to solve the problems of DHT P2Ps by avoiding hashing. Hashing does not keep data locality and is not amenable to range queries. This section introduces three kinds of non-DHT P2Ps: SkipNet [29], SkipGraph[30], and TerraDir [32]. SkipNet is designed for storing data close to users. SkipGraph is intended for supporting range queries. TerraDir is targeted for hierarchical name searches. Searching in such systems follows the specified neighboring relationships between nodes.

3.3.1 SkipNet and SkipGraph

DHTs balance load among different nodes. However, hashing destroys data locality. The work in [29] introduces *content locality* and *path locality*. Content locality refers to the fact that a data item is stored close to its users, and the nodes in a given organization store their data items inside

the same organization. Path locality means that routing between the querying node and the node responsible for the queried data are within their organization if these two nodes belong to the same organization. The overlay SkipNet in [29] supports these two data localities by using a hierarchical naming structure.



Level	clock-wise	counter clock-wise
2	R	R
1	I	W
0	E	Z

Level	clock-wise	counter clock-wise
2	E	E
1	Z	O
0	W	R

Figure 4. The peer name order and sample routing tables.

The SkipNet is based on the SkipList. A SkipList is a sorted linked list where some nodes have pointers that skip over varying numbers of list elements in the increasing sort order. In a perfect SkipList, all elements that have pointers skipping 2^h elements form the level *h*. The highest level of an element is called its height. In a probabilistic SkipList, node heights are determined probabilistically. A SkipList may also be considered as a hierarchy of sorted linked lists that are increasingly sparse.

The SkipNet is a modification of a SkipList. The data in the SkipNet are peer names (name IDs). The linked list is changed to a doubly-linked ring for path locality. All SkipNet nodes have the same $2\log^N$ number of pointers where *N* denotes the number of peers in the P2P. All pointers of a peer constitute its routing table. Figure 4 shows the peer name order and the routing tables for the peer *A*. The corresponding perfect SkipNet is shown in Figure 5. All peers are part of the *root ring* at level 0. The root ring is divided into two disjoint rings at level 1. The pointer of each peer at level 1 traverses 2 peers. Each ring at level 1 is divided into 2 disjoint rings at level 2. The pointers at level 2 traverse 4 peers. This procedure continues until at level 3, each peer forms a ring containing just itself. The pointers at level 3 traverse 8 (i.e. all) peers.

To ease efficient node insertions and deletions, the probabilistic SkipNet is used in practice. In such probabilistic design, each ring at level *i* is still split into two rings at level *i*+1. However, the peers in the two rings at level *i*+1 are randomly and uniformly selected from the peers in the corresponding ring at level *i*. With such probabilistic design, a pointer at level *i* traverses an *expected* 2^i number of peers. The routing efficiency is $O(\log^N)$ with high probability where *N* is the number of peers in the P2P. SkipNet generates a random binary bit vector for each peer. These random bit vectors are used to determine the random ring memberships of peers. A ring at level *i* consists of all peers whose random vectors have the same *i*-bit prefix. For example, the vectors for *A* and *D* are 000 and 001 respectively. Both *A* and *T* are in the same ring 0 at level 1 and the same ring 00 at level 2. The random bit vector is also used as the *numeric ID* of a peer.

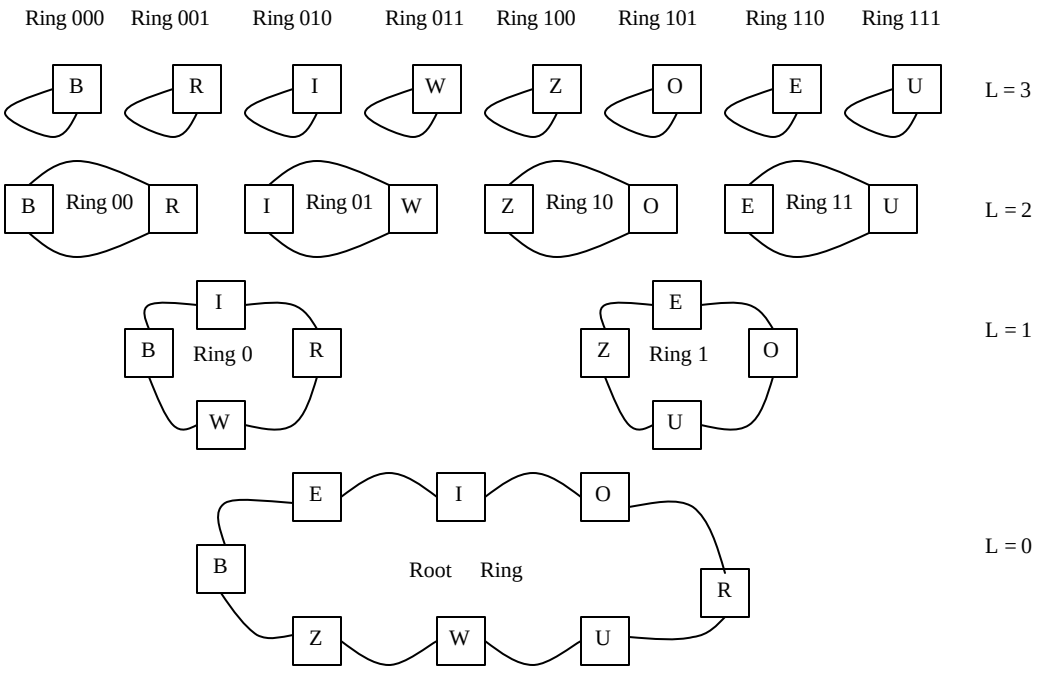


Figure 5. A sample of perfect SkipNet.

A file is stored in the node whose name ID is closest to the file name. To provide content locality, the node name is used as the prefix of the file name. For example, a file *cert9i.html* stored in the node *education.oracle.com* can be named *education.oracle.com/cert9i.html*. To search for a file named *fname*, the querying node *A* forwards the query along its highest level pointers until the node *B* whose name ID is closest to but is not greater than *fname* is reached. The node *B* continues the search along its next lower level pointers until the node *C* whose name ID is closest to but is not greater than *fname* is reached. This procedure continues until the node *E* whose name ID is closest to *fname* among all levels is reached. If the node *E* stores the sought file, the query succeeds. Otherwise, the query fails. To provide path locality, the DNS name with reversed components is used as the prefix of the file name. For example, nodes in the domain *oracle.com* can be named *com.oracle.node1*, *com.oracle.node2*, etc. In summary, searching by name ID visits nodes whose name IDs share a non-decreasing prefix of the desired file name.

In SkipNet, searching can also be done by numeric ID. It is similar to searching by name ID. However, the querying node starts the search from the lowest-level (Level 0). In Level 0, the search stops at the node whose numeric ID matches the first bit in the desired numeric ID. This node then continues the search in its level-1 ring and stops at the node with the first two matching bits. This procedure continues until the longest prefix is found in a ring at Level *h*. The search continues in this ring and terminates at the node that is numerically closest to the desired numeric ID.

SkipNet supports *constrained load balance*: loads are balanced among peers in a constrained range such as an organization. This is implemented by dividing the file name into two parts: a prefix and a suffix. The prefix specifies the domain where load balance should occur. The suffix is hashed uniformly to the peers in that domain.

DHTs do not support range queries very well because hashing destroys the ordering on hash keys such as file names. The overlay SkipGraph in [30] is tailored for range queries. SkipGraph is very similar to SkipNet. There are three differences. First, SkipNet is designed for providing data locality while SkipGraph is designed for supporting range queries. Secondly, each node in SkipNet is a computer and the node name is the computer name while each node in a SkipGraph is a resource and the node name is the resource name. Thirdly, SkipNet is a hierarchy of doubly-linked sorted rings while SkipGraph is a hierarchy of doubly-linked sorted list. Searching for a specific resource in SkipGraph is similar to searching by node name in SkipNet. A range query is resolved by first locating the range boundary and then traversing the linked list in the lowest level.

3.3.2 TerraDir

TerraDir [32] [33] is a general distributed directory service for searching data by hierarchical names like Unix file names. The hierarchical name space consists of meta-information about the data stored in the P2P system. The TerraDir directory structure is a rooted graph. Each node in this graph has one single canonical name and may have other names. All canonical names form a rooted tree. They are used to avoid cycles in wildcard queries and also used for failure recovery. Users can query the data using any node name. Each TerraDir directory node has a single owner. The owner is a peer that permanently maintains information for a TerraDir directory node. The owner is in charge of the replication of the owned node. Only the owner can make modifications to the owned node. Many directory nodes may be owned by the same owner. An owner keeps the following information (*state*) for each owned directory node: a label, a set of incoming edges, a set of outgoing edges, a set of attributes, a record, and some bookkeeping information. The incoming and outgoing edges contain the information about the peers that own or replicate the parent nodes and children nodes respectively. The attributes are meta-data about the node and are represented using (type, value) pairs. The record is the actual data represented by this node. The bookkeeping information is used for failure recovery. Each peer also permanently maintains the meta-data for all nodes replicated on it.

To reduce the routing latency, the owner of each node on the query path caches the partial query path from that node to the sought node. The querying peer (i.e. the owner of the starting node) caches the entire query path. The cache entry for a cached node includes information about that node, its parent and children, its owning peer, and a digest of the nodes permanently hosted by its owning peer. The node owner replicates an owned node in randomly selected peers. The number of replicas of an owned node is $k * h$, where k is a configurable constant and h is the level of that node in the TerraDir directory tree. Level 1 consists of all leaf nodes. Level 2 consists of all parents nodes of the leaf nodes. With this replication scheme, the average replication overhead per node is a constant. The network addresses of peers that have replicas of a node *A* are also part of the state which *A*'s parent maintains for *A*.

The searching in TerraDir is conducted as follows. Assume that a peer *A* is forwarding a query towards the peer that owns, replicates, or caches the target node *t*. Peer *A* proceeds in the following order:

1) It generates a list *L* of prefixes of node names it knows. *L* includes the target *t*, names of the nodes *A* owns, replicates, or caches. The entire node name is also considered as a prefix of that node name.
2) It sorts all elements in *L* in the increasing order of the distance (on the namespace tree) between the prefixes and the target *t*. This sorted list is called a candidate list.
3) It searches the candidate list for the first prefix whose owning peer or replicating peer *B* is known to *A*. This best prefix is closest to the target *t*.
4) It forwards the query to *B*.

The peer failures are handled as follows. If the peer storing the best prefix fails, the next best prefix in the candidate list is tried. If all peers storing the prefixes in the candidate list fail, the query is retried on a replica of the current node that is available and has not yet been visited. If such a replica does not exist, the query is retried on a replica of the directory root that is available and has not yet been visited. If no such replica exists, the query fails.

4. Searching in loosely structured P2Ps

In loosely structured P2Ps, the overlay structure is not strictly specified. It is either formed based on hints or formed probabilistically. In Freenet [34] and Phenix [40], the overlay evolves into the intended structure based on hints or preferences. In Symphony [31] and the work in [35], the overlay is constructed probabilistically. Searching in loosely structured P2P systems depends on the overlay structure and how the data is stored. In Freenet, data is stored based on the hints used for the overlay construction. Therefore, searching in Freenet is also based on hints. In Phenix [40], the overlay is constructed independent of the application. The data location is determined by applications using the Phenix. Therefore searching in Phenix is application dependent. In Symphony [35], the data location is clearly specified but the neighboring relationship is probabilistically defined. Searching in Symphony is guided by reducing the numerical distance from the querying source to the node that stores the desired data.

4.1 Freenet

Freenet [34] is one loosely structured decentralized P2P designed for protecting the anonymity of data sources. It supports the DHT interface. Each node maintains a local datastore and a dynamic routing table. The routing table of a node contains addresses of some other nodes and the keys possibly stored on these nodes. Because of the storage capacity, both the datastore and the routing table are managed using the LRU algorithm.

The query routing in Freenet is similar to DFS. Given a query for a file with a key k, the querying node A first looks up its local datastore. If the file is in the local data store, the query is resolved. Otherwise, A forwards the query to the node B in its routing table whose key is nearest to k. On receiving the query, B performs the similar computation. If the file is not stored on B, then B forwards the query to the neighbor in its routing table which has the nearest key to k. This forwarding procedure continues until the query terminates. During query routing, some node may not forward the query to the neighbor with the nearest key because that neighbor is down or a loop may be detected. In such cases, this node tries the neighbor with the second nearest key. If the node can not forward to all its neighbors, the node reports a failure back to its upstream node. This upstream node will try its second best choice. A TTL limit is specified to restrict the number of messages in query routing. When the file is found, the file is returned to the querying node hop by hop along the reverse of the query path. Each node except the last one on the query path caches the found file and creates an entry in the routing table for the key k.

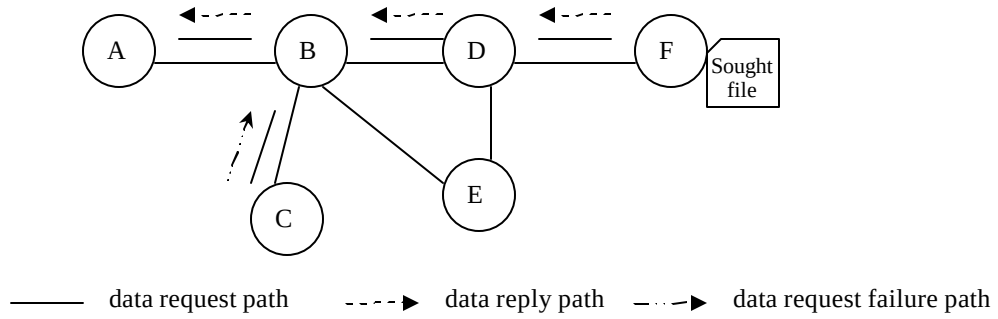


Figure 6. A sample of querying processing in Freenet.

To provide anonymity, each node except the last one on the query path can change the reply message and claim itself or another node as the data source. Figure 6 depicts a querying routing example. Node *A* starts the query for the file *f* with the key *k* stored on node *F*. It first sends the query to node *B* with the nearest key. Node *B* then forwards to its best neighbor *C*. *C* reports the failure back to node *B* because *C* does not have any other neighbor. *B* then forwards the query to its second best neighbor *D*. *D* forwards the query to its best neighbor *F*. The file is found in *F*. *F* then returns the file to *A* through the path *F -> D -> B -> A*. After this query, nodes *A*, *B*, *D* all have the file *f* in their datastores and entries for the key *k* in their routing tables.

To insert a file *f* with key *k*, the inserting node first issues a query for the key *k* to avoid duplicate keys. This search request for *k* is processed as a regular query. If an existing file with key *k* is found, the file is returned to the inserting node. All nodes except the last one on the search path cache the file and add an entry for the key k in their routing tables. The inserting node will retry the insertion with a different key. If the search fails, the last node on the search path sends a "no-collision" message back to the inserting node. The inserting node then sends an insert message to its neighbor on the path established by the initial search for duplicate keys. On receiving the insert message, the neighbor adds the new file into its own datastore and a new entry for the new file into its routing table. The neighbor then forwards the insert message to the next node on the initial search path. The insert message is normally propagated along the initial search path. If some node on the search path fails, the upstream node forwards the insert message to the neighbor with the second nearest key, etc. The insertion terminates successfully when a TTL limit is reached and fails when the message is backtracked to the original inserter. To provide data anonymity, any node on the insert path may change the insert message and claim itself or another node as the data source. Freenet does not support persistent data storage. Unpopular files with few requests are purged from the datastores by the LRU algorithm.

Freenet's query routing improves over time as more queries are answered and more files are inserted. Nodes tend to locate collections of similar keys. Once a node is listed in the routing table under a key, it will receive many requests for keys similar to that key. As the node builds up its routing table, it gets more informed about the locations of similar keys. Nodes also tend to store files with clusters of keys. Forwarding a successful query causes a node to keep a copy of the sought file with key *k*. File insertion normally follows the query path. Therefore files with similar keys are inserted along the same query path. The nodes on the query path accumulate files of similar keys in their datastores. Consequently, the routing tables of these nodes also contain entries with similar keys.

A new node joins the Freenet by sending messages to some existing nodes discovered through out-of-band means. The new node announces its presence using a random walk. The key

associated with the new node is the *XOR* of the random seeds generated by all nodes on the random walk. A cryptographic protocol is used during the random walk for verifying the truth of each random seed.

4.2 Searching the power-law graph overlay

In a power-law graph, node degrees follow a power-law distribution [14]. This means that the probability of a node with the degree k, P_k, is

$$P_k \propto ck^{-\alpha}$$

where, α is a positive integer, c is a constant. In many networks, α is close to 2. Freenet [34] and Gnutella [15] tend to evolve into a power-law graph with a power-law exponent close to 2. An efficient strategy for searching power-law overlays is proposed in [14]. The scheme tries to utilize high-degree nodes. First, each node forwards the query to a neighbor with a higher degree. This rule continues until the node with the highest degree is reached. After the highest-degree node is accessed, the node of approximately second highest degree will be selected. Following this procedure, the searching algorithm approximately visits nodes with degrees in the decreasing order across the entire graph.

The approach in [40] makes a power-law overlay "organically" emerge by guiding the node-join process through preferences. The new node prefers connecting to existing nodes with high degrees. Specifically, when a new node joins the system, it gets a list of live nodes using a rendezvous mechanism. Then, the new node divides this node list into two sets: *random* neighbors and *friend* neighbors. Next, the new node sends a TTL-1 ping message to each friend. Each friend returns its own neighbors in a pong message to the new node. Each friend also forwards the ping message to its own neighbors. On receiving the ping message, the friend's neighbors add the new node to their own *special lists*. The new node's friends and the friends' neighbors form a *candidate list* of the new node. The candidate list is then sorted based on the decreasing order of the number of node appearances. The top c number of nodes in this list is selected as the *preferred list* of the new node. The new node then creates connections to all nodes in its random list and preferred list. The new node also periodically contacts the neighbors in its preferred list for possible backward connections. A preferred neighbor increases its counter each time it is contacted by the new node. If the counter reaches a constant value r, the preferred neighbor decreases the counter by r and adds a backward connection to the new node. The work in [40] is intended for a generic overlay topology independent of the application. Searching on the resulting power-law overlay can be any scheme suitable for the specific application.

4.3 Searching the small-world model overlay

A small-world graph is a graph in which each node has many local connections and a few random long-range connections. The diameter of a small-world graph is $O((\log^N)^2)$ [49]. Symphony [31] employs the small-world graph model to implement the DHT. A data item with a key k is mapped to a node whose ID is numerically closest to k. Each node has two *short-distance links* to its immediate predecessor and successor in the ID ring and m *long-distance links* to other distant nodes in the ID ring. These distant nodes are chosen probabilistically. Specifically, when selecting a long-distance neighbor, a node A draws a random number r based on the probabilistic distribution function: $p_N(r) \propto 1/(r \ln^N)$, where r is in $[1/N, 1]$ and N is the current number of nodes in the P2P. Then the node A finds another node B that is responsible for the number r. This node B is selected as one long-distance neighbor of node A. m is determined experimentally. N is estimated based on the sum of segment lengths managed by a set of distinct nodes.

Symphony has a unidirectional routing protocol and a bidirectional routing protocol. In the unidirectional routing protocol, each node on the query path forwards the query to one of its immediate or distant neighbors that is *clockwise* closest to the sought key. In the bidirectional routing protocol, the query is forwarded to one immediate or distant neighbor that has the *absolutely shortest* distance (*clockwise or counterclockwise*) from the responsible node. The *look-ahead approach* is proposed to reduce the query latency even further. In this approach, each node looks ahead at its neighbors' neighbors during query forwarding. For example, in the 1-lookahead approach, each node forwards a query to the neighbor whose neighbor is closest to the sought key.

In Freenet, the cache replacement policy LRU for datastores can destroy the key clustering in both the datastores and the routing tables when the number of files stored is huge. The work in [35] solves this problem by incorporating a small-world model in the datastore cache replacement policy. Routing tables are tailored by datastores. Integrating a small-world model in the datastore cache replacement policy makes the routing tables emulate a small-world model overlay. Specifically, each new node selects a random seed from the key space. When a new data item with a new key k comes in and the datastore is full, the node first compares the new key with the existing key k' in its datastore that is farthest from its seed. If the new key k is closer to the seed than k', then the node removes the file with key k', stores the new file with key k, and adds a new entry for k in the routing table. (This is intended for emulating short links to close neighbors in a small-world model.) Otherwise, the node probabilistically removes k', caches k, and adds a new routing table entry for the new key k. (This is designed for emulating a small number of random long links to distant neighbors in a small-world model.) This scheme enforces clustering of keys around the random seed at each node's datastore and routing table.

5. Conclusion

This chapter discusses various searching techniques in peer-to-peer networks (P2Ps). First, the concept of peer-to-peer networks is introduced and different schemes are classified. Next, searching strategies in unstructured P2P systems, strictly structured P2P systems, and loosely structured P2P systems are presented. Strengths and weaknesses of these approaches are addressed.

Clearly, significant progress has been made in the P2P research field. However, there are still many issues left unresolved. First, good benchmarks need to be developed to evaluate the actual performance of various techniques. The work in [36] [37] made such an initial attempt. Secondly, schemes amenable to complex queries supporting relevance ranking, aggregates, or SQL are needed to satisfy the practical requirements of P2P users [3]. An initial work in [51] addressed the top-K query, the query supporting relevance ranking of query results. It uses a global index and ranks the result using the term frequency and inverse document frequency. This scheme is not amenable to large-scale systems. Thirdly, security issues have not been addressed by most current searching techniques. An initial work in [45] adds security by preferring to forward queries to friends obtained through third-party services such as instant messenger service. Fourthly, P2P systems are dynamic in nature. Unfortunately existing searching techniques can not handle concurrent node join-leave gracefully. Fifthly, good strategies are needed to form overlays that consider the underlying network proximity. Some initial effort has been made in Coral, Hieras, and other approaches in [42] [43] [44]. Both Coral and Hieras consider network proximity in the initial construction of overlays. Coral uses ping-pong messages to estimate round-trip times between nodes. Hieras employs distributed binning to estimate the proximity between nodes. The works in [42] [43] [44] try to modify the existing overlay to match the underlying network. The modification is conducted by deleting inefficient overlay links and adding efficient ones. Sixthly, almost all existing techniques are forwarding-based techniques. Recently, a study on non-

forwarding techniques [39] was done. More effort is required to develop good non-forwarding techniques and to compare non-forwarding techniques to various forwarding techniques.

Acknowledgements

This work was supported in part by US National Science Foundation grants CCR 9900646, CCR 0329741, ANI 0073736, and EIA 0130806.

References

1. H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in P2P systems", *Communications of ACM*, Vol.46, No.2, 2003.
2. C. Yang and J. Wu, "A dominating-set-based routing in peer-to-peer networks," *Proc. of the 2nd International Workshop on Grid and Cooperative Computing Workshop (GCC'03)*, 2003.
3. N. Daswani, H. Garcia-Molina, and B. Yang, "Open problems in data-sharing peer-to-peer systems", *Proc. of the 9th International Conference on Database Theory (ICDT'03)*, 2003.
4. D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rolins, and Z. Xu, "Peer-to-peer computing", *HP Lab technical report*, HPL-2002-57, 2002.
5. D. Barkai, "Technologies for sharing and collaborating on the net", *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.
6. B. Yang, and H. Garcia-Molina, "Improving search in peer-to-peer networks", *Proc. of the 22nd IEEE International Conference on Distributed Computing (IEEE ICDCS'02)*, 2002.
7. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks", *Proc. of the 16th ACM International Conference on Supercomputing (ACM ICS'02)*, 2002.
8. A. Crespo, and H. Garcia-Molina, "Routing indices for peer-to-peer systems", *Proc. of the 22nd International Conference on Distributed Computing (IEEE ICDCS'02)*, 2002.
9. S. C. Rhea, and J. Kubiatowicz, "Probabilistic location and routing", *Proc. of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02)*, 2002.
10. V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-yazti, "A local search mechanism for peer-to-peer networks", *Proc. of the 11th ACM Conference on Information and Knowledge Management (ACM CIKM'02)*, 2002.
11. D. Tsoumakos, and N. Roussopoulos, "Adaptive probabilistic search in peer-to-peer networks", *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.
12. D. Tsoumakos, and N. Roussopoulos, "Adaptive probabilistic search in peer-to-peer networks", *technical report*, CS-TR-4451, 2003.
13. D. Tsoumakos, and N. Roussopoulos, "A comparison of peer-to-peer search methods", *Proc. of 2003 International Workshop on the Web and Databases*, 2003.
14. L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman, "Search in power-law networks", *Physical Review*, Vol. 64, 046135, 2001.
15. http://www.gnutella.com
16. I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications", *Proc. of the 2001 ACM Annual Conference of the Special Interest Group on Data Communication (ACM SIGCOMM'01),* 2001.
17. A. Rowstron, and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", *Proc. of the 18th IFIP/ACM International Conference of Distributed Systems Platforms*, 2001. www.cs.rice.edu/CS/systems?Pastry.

18. K. Hildrum, J. Kubiatowicz, S. Rao, and B. Y. Zhao, "Distributed object location in a dynamic network", *Proc. of 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.
19. P. Maymounkov, and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric." *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Springer-Verlag version, 2002.
20. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network", *Proc. of the 2001 ACM Annual Conference of the Special Interest Group on Data Communication (ACM SIGCOMM'01)*, 2001
21. M. F. Kaashoek, and D. R. Karger, "Koorde: a simple degree-optimal distributed hash table", *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.
22. D. Xuan, S Chellappan, and M. Krishnamoorthy, "RChord: an enhanced chord system resilient to routing attacks", *Proc. of the 2003 International Conference in Computer Networks and Mobile Computing*, 2003.
23. I. Gupta, K. Birman, P. Linga, A. Demers, and R. V. Renesse, "Kelips: building an efficient and stable P2P DHT through increased memory and background overhead", *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003
24. M. J. Freedman, and D. Mazieres, "Sloppy hashing and self-organized clulsters", *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003
25. A. T. Mizrak, Y. Cheng, V. Kumar, and S. Savage, "Structured superpeers: leveraging heterogeneity to provide constant-time lookup", *Proc. of the IEEE Workshop on Internet Applications (WIAPP'03)*, 2003.
26. L. Garces-Erice, E.W.Biersack, P.A. Felber, K.W. Ross, and G. Urvoy-Keller, "Hierarchical peer-to-peer systems", *Parallel Processing Letters*,Vol.13, 2003.
27. KaZaA, http://www.kazaa.com
28. B.Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiatowicz, "Brocade: landmark routing on overlay networks", *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.
29. N. J.A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "SkipNet: a scalable overlay network with practical locality properties", *Proc. of 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, 2003.
30. J. Aspnes, and G. Shah, "Skip Graphs", *technical report*, Yale University, 2003.
31. G. S. Manku, M. Bawa, and P. Raghavan, "Symphony: Distributed hashing in a small world", *Proc. of 4th USENIX Symposium on Internet Technology and Systems (USITS'03)*, 2003
32. B. Silaghi, B. Bhattacharjee, and P. Keleher, "Query routing in the TerraDir Distributed Directory", *Proc. of SPIE ITCom'02*, 2002.
33. B. Bhattacharjee, P. Keleher, B. Silaghi, "The design of TerraDir", *technical report*, CS-TR-4299, University of Maryland, College Park, 2001.
34. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system", *Proc. of ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.
35. H. Zhang, A. Goel, and R. Govindan, "Using the small-world model to improve freenet performance", *Proc. of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'03)*, 2003.
36. S. Rhea, T. Roscoe, and J. Kubiatowicz, "Structured peer-to-peer overlays need application-driven benchmarks", *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.
37. T. Lin, H. Wang, "Search performance analysis in peer-to-peer networks", *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.
38. Z. Xu, R. Min, and Y. Hu, "HIERAS: a DHT based hierarchical P2P routing algorithm", *Proc. of the 32nd International Conference on Parallel Processing (ICPP'03)*, 2003.

39. B. Yang, P. Vinograd, and H. Garcia-Molina, "Evaluating GUESS and Non-forwarding peer-to-peer search", *Proc. of the 24th IEEE International Conference on Distributed Computing Systems (IEEE ICDCS'04)*, 2004
40. R. H. Wouhaybi, and A. T. Campbell, "Phenix: supporting resilient low-diameter peer-to-peer topologies", *Proc. of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, 2004.
41. E. Cohen, and S. Shenker, "Replication strategies in unstructured peer-to-peer networks", *Proc. of the ACM Annual Conference of the Special Interest Group on Data Communication (ACM SIGCOMM'02)*, 2002.
42. Y. Liu, X. Liu, and L. Xiao, "Location-aware topology matching in P2P systems", *Proc. of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, 2004.
43. S. Ren, L. Guo, S. Jiang, and X. Zhang, "SAT-Match: a self-adaptive topology matching method to achieve low lookup latency in structured P2P overlay networks", *Proc. of the 18thIEEE International Parallel & Distributed Processing Symposium (IPDPS'04)*, 2004
44. Y. Liu, L. Xiao, and L. M. Ni, "Building a scalable bipartite P2P overlay network", *Proc. of the 18thIEEE International Parallel & Distributed Processing Symposium (IPDPS'04)*, 2004.
45. S. Marti, P. Ganesan, and H. Garcia-Molina, "DHT routing using social links", *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, 2004.
46. D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: a scalable and dynamic emulation of the butterfly", *Proc. of Principles of Distributed Computing (PODC) 2002*, 2002.
47. H. Shen, C.-Z. Xu, and G. Chen, "Cycloid: a constant-degree and lookup-efficient P2P overlay network", *Proc. of the 18thIEEE International Parallel & Distributed Processing Symposium (IPDPS'04)*, 2004.
48. P. Ganesan, K. Gummadi, and H. Garcia-Molina, "Canon in G major: designing DHTs with hierarchical structure", *Proc. of the 24th IEEE International Conference on Distributed Computing Systems (IEEE ICDCS'04)*, 2004.
49. J. Kleinberg, "The small-world phenomenon: an algorithmic perspective", *Proc. of the 32nd ACM Symposium on Theory of Computing*, 2000.
50. B. Bloom, "Space/time trade-offs in hash coding with allowable errors", Communications of ACM, Vol. 13(7), 1970.
51. F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen, "PlanetP: using gossiping to build content addressable peer-to-peer information sharing communities", *Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.
52. I. Jawhar and J. Wu, "A Two-Level Random Walk Search Protocol for Peer-to-Peer Networks", *Proc. of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics*, 2004.
53. http://www.napster.com