Routing Indices For Peer-to-Peer Systems

Arturo Crespo, Hector Garcia-Molina

Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA
{crespo,hector}@db.Stanford.edu

Abstract

Finding information in a peer-to-peer system currently requires either a costly and vulnerable central index, or flooding the network with queries. In this paper we introduce the concept of Routing Indices (RIs), which allow nodes to forward queries to neighbors that are more likely to have answers. If a node cannot answer a query, it forwards the query to a subset of its neighbors, based on its local RI, rather than by selecting neighbors at random or by flooding the network by forwarding the query to all neighbors. We present three RI schemes: the compound, the hop-count, and the exponential routing indices. We evaluate their performance via simulations, and find that RIs can improve performance by one or two orders of magnitude vs. a flooding-based system, and by up to 100% vs. a random forwarding system. We also discuss the tradeoffs between the different RI schemes and highlight the effects of key design variables on system performance.

Keywords: Peer-to-peer systems, Routing Index, Query Processing, Query Forwarding, Approximate Indices, Content Discovery, Distributed Search Mechanisms

1 Introduction

Peer-to-peer systems (P2P) have grown dramatically in recent years. In a P2P system, distributed computing nodes of equal roles or capabilities exchange information directly with each other. These systems represent an incredible wealth of information allowing users to exchange documents (Freenet [Wora]), music files (Napster [Word], Gnutella [Worb]), and even computer cycles (Seti-at-home [Worc]). A key part of a P2P system is document discovery. Our goal is to help users find documents with content of interest across potential P2P sources efficiently.

There are many mechanisms for searching in a P2P system, each with their own advantages and disadvantages. These solutions can be classified in three broad categories: mechanisms without an index (non-index search), mechanisms with specialized index nodes (centralized search), and mechanisms with indices at each

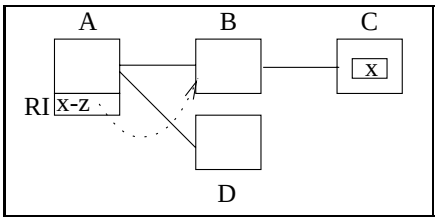

Figure 1: Routing Indices

node (distributed search). Gnutella uses a mechanism where nodes do not have an index and queries are propagated from node to node until matching documents are found. This search mechanism works by flooding the network (or a subset of it) with each query in the hope of finding a match. Although this approach is simple and robust, it has the disadvantage of the enormous cost of flooding the network every time a query is generated.

Centralized-search systems use specialized nodes that maintain an index of the documents available in the P2P system. To find a document, the user queries an index node to identify nodes having documents with the content of interest. These central indices may be built with the cooperation of the nodes (e.g., Napster nodes provide a list of available files at sign-in time) or by crawling the P2P network (as in a web search engine). The advantages of a centralized-search mechanism is efficiency (just a single message is needed to resolve a query). However, a centralized system is vulnerable to attack (e.g., index sites can be shut down by a court order or a hacker attack) and it is difficult to keep the indices up-to-date.

A distributed-index mechanism, the option we will study in detail in this paper, maintains indices at each node. These distributed indices need to be small, so instead of using traditional "destination" indices, we use *Routing Indices* (RIs) that give a "direction" towards the document, rather than its actual location. To illustrate, consider Figure 1 which shows four nodes A, B, C, and D, connected by the solid lines. The document with content "x" is located at node C, but the RI of node A points to neighbor B instead of pointing directly to C (dotted arrow). By using "routes" rather than destinations, the indices are proportional to the number of neighbors, rather than to the number of documents. We can reduce the size of RIs even further by using approximate indices, i.e., by allowing RIs to give a hint (rather than a definite answer) about the location of a document. For example, in the same figure, an entry in the RI of node A may cover documents with contents "x," "y," or "z." A request for documents with content "x" will yield a correct hint, but one for content "y" or "z" will not.

In this paper we study options for building effective RIs, and evaluate their performance. In particular, the contributions of this paper are:

- We introduce Routing Indices, an efficient way of locating content in a P2P system (Sections 3 and 4).
- We present three kinds of RIs: the compound, the hop-count, and the exponential routing indices (Sections 5, 6, and 7).

- We evaluate the performance of RIs via simulations, and find that RIs can improve performance by one or two orders of magnitude over a flooding-based system, and by 50-100% versus a random forwarding system (Section 8).

2 Related Work

The problem of indexing a P2P network is closely related to the problem of indexing a distributed database. For a survey of current approaches please refer to [Kos00]. Despite some superficial similarities, algorithms for indexing distributed databases are based on fundamental assumptions that are not applicable to P2P systems. First, distributed database algorithms assume that the nodes in a distributed database are stable and connected most of the time. This is not to say that nodes cannot fail. In fact, nodes can fail, but in general, they are expected not to fail very often, and when they do, they are repaired and returned to the system. Second, distributed database algorithms generally assume that the number of nodes is relatively small. This assumption is not applicable to a P2P system, which can have tens or even hundreds of thousands of nodes.

There are several working P2P systems currently available, each with its own "indexing" approach. Napster [Word] uses centralized indices, which, as stated before, are vulnerable to attack and are difficult to fund and scale. Gnutella [Worb] does not build indices, instead, queries flood a significant part of the network, resulting in a simple but very costly approach as just one query can expand into hundred of thousands of requests through the Gnutella network. Freenet [Wora] uses an interesting approach to indexing. Each node builds an index with the location of recently requested documents, so if they are requested again, the document can be retrieved at a very low cost. However, Freenet has two limitations: first, queries are restricted to the identifier of a specific document, and second, it takes time for a node to build an effective index. In Freenet, a newly connected node cannot do better than just blindly forward queries to its neighbors. Oceanstore [KBC+00] uses an approach similar to the one proposed in this paper (in fact, their approach can be considered a special case of a compound RI). The key difference versus our approach is that Oceanstore assumes a static network and that queries are on document identifiers, rather than on the content of the documents.

Selecting a neighbor for forwarding a query is also related to traditional routing algorithms [Tan96]. In fact, an RI can be considered a generalization of the data structures used to implement the Bellman-Ford routing algorithm [Bel57, FF62]. The major difference with our algorithms is that the standard routing algorithms are designed to transmit a packet between two nodes through the shortest route. In our case, we need to get a "packet" from one node to one or *more* nodes so we find the best answers to a query. Also, the destination of a packet is not pre-defined (as in IP routing), but instead it depends on the query contained by the packet.

IP routing to multiple destinations (multicast algorithms) has been studied extensively (see for exam-

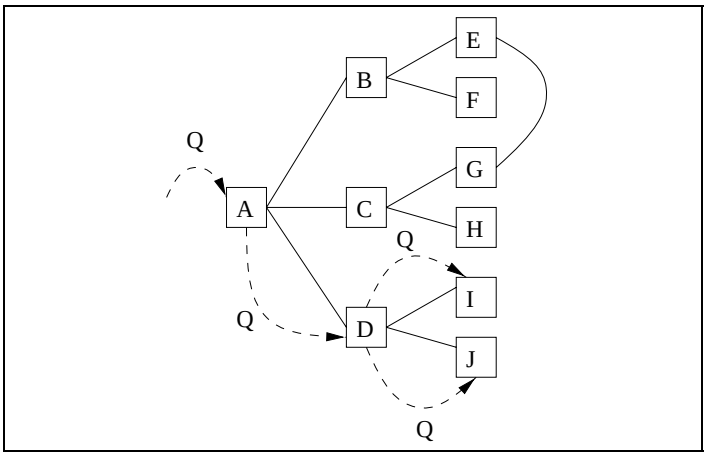

Figure 2: P2P System Model

ple [Mil98]). However, multicast algorithms require the creation of a relatively stable multicast tree. In the case of query processing in a P2P system, we cannot predict in advance what the multicast tree would look like, and even if we could, the cost of creating a tree for each query would be prohibitive.

The problem of selecting the best database to which to send a query was studied as part of the GlOSS project [GGMT00, GGM95]. However, GlOSS assumes the we are selecting among a set of databases, rather than among "paths" that lead to a set of databases.

Some recent work has empirically evaluated P2P systems. A survey and evaluation of centralized-search P2P systems can be found at [YGM01a]. An evaluation and description of the present state of Gnutella can be found at [Clia]. Finally, [YGM01b] focuses on search techniques that do not use indexes, although it also studies one type of "local area index." In such indices, a node indexes the content of nodes within "r" hops. However, these indices are not routing indices, they are traditional indices.

3 Peer-to-peer Systems

A P2P system is formed by a large number of *nodes* that can join or leave the system at any time and that have equal capabilities. Each node is connected to a relatively small set of neighbors which in turn are connected to more nodes. In Figure 2, the neighbors of node A are nodes B, C, and D. Note that there might be cycles in the network (such as the one caused by the link between E and G). Each node has a local document database that can be accessed through a local index. The local index receives content queries (e.g., a request for documents containing the words "database systems," a request for documents containing a picture of the sun, etc.) and returns pointers to the documents with the requested content.

3.1 Query Processing in a Distributed-Search P2P System

In a distributed-search P2P system, users submit queries to any node along with a stop condition (e.g., the desired number of results). A node receiving a query, first evaluates the query against its own database, returns to the user pointers to any results, and, if the stop condition has not been reached, the node selects one or more of its neighbors and forwards the query to them (along with some state information). In turn, each of the neighbors evaluates the query in a similar fashion, returns result pointers to the user and forwards the query to neighbors.

To illustrate, consider Figure 2. Node A initially receives a query. Node A checks for local results and sends those results to the requesting node. Then, assuming that the stop condition has not been satisfied, node A selects node D as the best neighbor to handle the query and forwards the query to it (dashed arrow). Note that for nodes to be able to verify if the stop condition has been reached, we need to include the number of results found so far as state information in each query-forwarding message. Then D processes the query and selects I as the best neighbor to continue handling the query. Let us assume now that I has processed the query, but not enough results have been found to reach the stop condition. In this case, I returns the query to D which forwards the query to the next best neighbor (J in this case).

Queries can be forwarded to the best neighbors in parallel or sequentially. In the example above, D can send the query to I, wait for results, and then send the query to J; or it can send the query to I and J simultaneously. A parallel approach yields better response time, but generates higher traffic and may waste resources (e.g., if the stop condition is reached after sending the query to I, there is no need to send it to J). In this paper, we focus on a sequential forwarding of the queries.

4 Routing indices

In this section we present an example of how the *compound RI* (CRI) works. Later, in Section 5 we define RIs in general and present two other RIs: the exponential RI and the hop-count RI. in Section 6.

The objective of a Routing Index (RI) is to allow a node to select the "best" neighbors to send a query to. A RI is a data structure (and associated algorithms) that, given a query, returns a list of neighbors, ranked according to their *goodness* for the query. The notion of goodness may vary but in general it should reflect the number of documents in "nearby" nodes.

As a running example, we will use a P2P system for retrieval of text documents with the network depicted on the right side of Figure 3. For simplicity, this network does not have cycles (we will discuss cycles in Section 7). In this system, documents are on zero or more "topics," and queries request documents on particular topics. Each node has a local index for quickly finding local documents when a query is received. Nodes also have a CRI containing (i) the number of documents along each path and (ii) the number of documents on each topic of interest In Figure 3 we show an example of a CRI for node A with three

Path	Number of documents	Documents with topics: Databases	Networks	Theory	Languages
B	100	20	0	10	30
C	1000	0	300	0	50
D	200	100	0	100	150

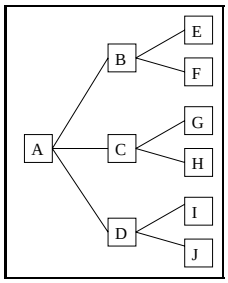

Figure 3: A Sample Compound RI

neighbors (paths): B, C, and D. For simplicity, we assume that there are only four topics of interest: databases, networks, theory, and languages. In the figure, we can see that we can access 1000 documents through C (i.e., there are 1000 documents in C, G and H) and that of those documents, 300 are about "networks" and 50 are about "languages."

The RI may be "coarser" than the local indices maintained at nodes. For example, node A could maintain a more detailed local index in which each document is further classified into sub-categories. By keeping a *summary* of the detailed index, we achieved a more compact RI at the cost of introducing "errors" when user queries are based on the subcategories. Specifically, the summarizing of the local index may introduce overcounts or undercounts in the RI. For example, a summarization that groups several subtopics into a single topic (e.g., "indices", "recovery", and "SQL" into "databases") may introduce overcounts on the number of documents available. In fact, a query for documents on "SQL" will be converted into a query for documents on "databases," making us believe that there are many documents on "SQL" whereas in reality there may be few or even none. Summarization can also introduce undercounts. For example, if the summarization uses a frequency threshold (e.g., throws away topics with very few documents), then we may believe that there are no documents on a topic when there are in fact a few.

Given the index, we need now to compute the "goodness" of each node for a query. For CRIs we will use the number of documents that may be found in a path as a measure of goodness. To compute the number of documents, we will use the estimators in [GGMT94a, GGMT94b]. Given that our focus is not on the estimators but on the use and maintenance of RIs, throughout the paper we will use a simplified model where queries are conjunction of subject topics, documents can have more than one topic, and document topics are independent. Thus, we can estimate the number of results in a path as: $NumberOfDocuments \times \prod_i \frac{CRI(s_i)}{NumberOfDocuments}$ where $CRI(s_i)$ is the value for the cell at the column for topic s_i and at the row for a neighbor.

To illustrate, consider a query for documents on "databases" and "languages" received by A. We estimate the number of results as $\frac{20}{100} \times \frac{30}{100} \times 100 = 6$ at B, $\frac{0}{100} \times \frac{0}{100} \times 100 = 0$ at C, and $\frac{100}{200} \times \frac{150}{200} \times 200 = 75$ at D. Therefore, the "goodness" of path B will be 6, of path C will be 0, and of path D will be 75. Note that these numbers are just estimates and they are subject to overcounts and/or undercounts. In particular,

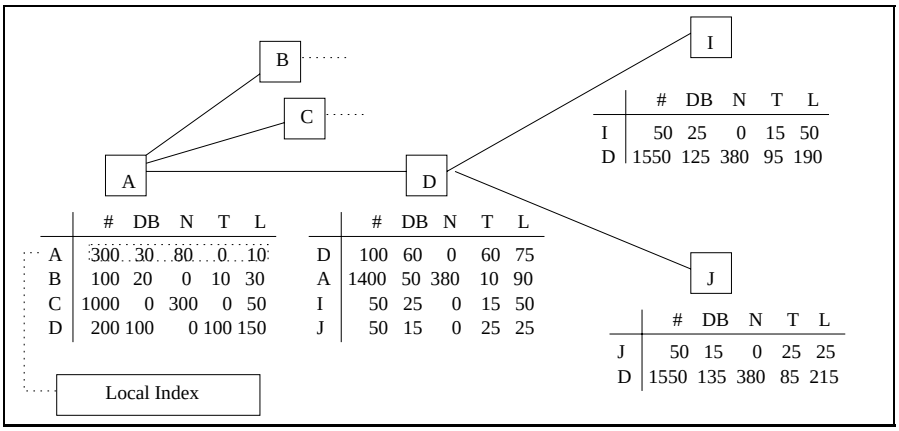

Figure 4: Routing Indices

if there is a strong correlation between the topics "databases" and "languages," then path B may have as many as 20 documents matching the query for topics "databases" and "languages" On the other hand, if there is a strong negative correlation between the topics "databases" and "languages," then there may be no documents in path B on either topic.

A limitation of using CRIs is that they do not take into account the difference in cost due to the number of "hops" necessary to reach a document. For example, the documents along path B may all be just one hop away, while the documents along path C may be scattered in a long chain of nodes and finding them would require many messages. Later, we will introduce more sophisticated RIs that do not have this limitation.

In the rest of the section we describe how compound RIs are used, created, and maintained.

4.1 Using Routing Indices

In this subsection we show how RIs, and in particular compound RIs, can improve the performance of query processing in a P2P system. Consider the P2P system described in Figure 3. In Figure 4 we present part of the P2P network with RIs attached to each node. For compactness, we are representing the four topics of interest: database, network, theory, and languages with the letters DB, N, T, and L respectively. In the example we are assuming that the first row of each RI contains the summary of the local index. (This summary can be obtained, e.g., by consolidating subtopics into the main topics, or perhaps by using clustering on a local keyword index to generate topics for each of its documents.) In particular, the summary of A's local index shows that A has 300 documents: 30 about databases, 80 about networks, none about theory, and 10 about languages. The rest of the rows represent a compound RI. In the example, the RI shows that node A can access 100 database documents through D (60 in D, 25 in I, and 15 in J).

When A receives from a client a query for documents about "databases" and "languages," it attempts

to use the local database to answer the query. If not enough answers are found, it computes the goodness of each path as explained earlier. In this case, the goodness of B, C, and D is 6, 0, and 75 respectively, so A selects D as the best neighbor to forward the query to. In turn, D returns all local results to the client of A and, if not enough results are found, computes the goodness of I and J (25 and 7.5). Since I has the highest goodness, D forwards the query to I. In turn, I returns local results, but it cannot forward the query any further as it does not have any peers besides D, so (if more results are needed) it returns the query to D which forwards it to its best next neighbor J. Even though the network in the example is very small, a query with a stop condition of 50 documents will generate 9 messages when using flooding; but only 3 messages if we use the RI. Even if we send the query serially in a depth-first fashion to neighbors ranked randomly, we will have 3 messages in the best case and 9 messages in the worst case. The savings in the number of messages when using RIs are the result of forwarding the query only to the nodes that have a high potential of having results.

The storage space required by an RI in a node is modest as we are only storing index information for each neighbor. Furthermore, the storage space per neighbor can be adjusted by increasing or decreasing the level of summarization of the index. Specifically, if s is the counter size in bytes, c is the number of categories, N the number of nodes, and b the branching factor (i.e., number of neighbors), then a centralized index would require $c \times (t+1) \times N$ bytes, while each node of a distributed system would need $c \times (t+1) \times b$ bytes. Thus, the total for the entire distributed system is $c \times (t+1) \times b \times N$ bytes. Although the RIs require more storage space overall than a centralized index, the cost of the storage space is shared among the network nodes.

4.2 Creating Routing Indices

Let us now turn our attention to how RIs are created. Returning to our running example, let us assume that initially there is no connection between A and D. The initial state of the system is shown by the solid lines of Figure 5a. When the $A-D$ connection is established, node A informs node D of all the documents that can be accessed through node A. Specifically, node A *aggregates* its RI and sends it to D. In our example, the aggregation is done by adding all the vectors in the RI. (We will describe additional aggregation procedures in Section 5.) Thus, A sends D a vector saying that it has access to 1400 documents $(300 + 100 + 1000)$, of which 50 are on databases $(30 + 20 + 0)$, 380 on networks $(80 + 0 + 300)$, 10 on theory $(0 + 10 + 0)$, and 90 on languages $(10 + 30 + 50)$. A does not need to send more information as D does not need to know the precise location of the documents, but only that they can be accessed through A. After D receives the aggregated RI from A, it adds an additional row to its RI with A's identifier and A's aggregated RI (as shown in Figure 5b). Note that by aggregating RIs we reduce both the amount of information transmitted and the storage space used. Similarly, D, aggregates its RI (excluding the row for A if it is already in the RI) and sends its aggregated RI to A. Note that the RI creation process at A and D can be done in parallel.

After A and D update their RIs, they need to inform their other neighbors that now they have access to more documents. Thus, D sends an aggregate of its RI to I (excluding I's row) and to J (excluding J's row)

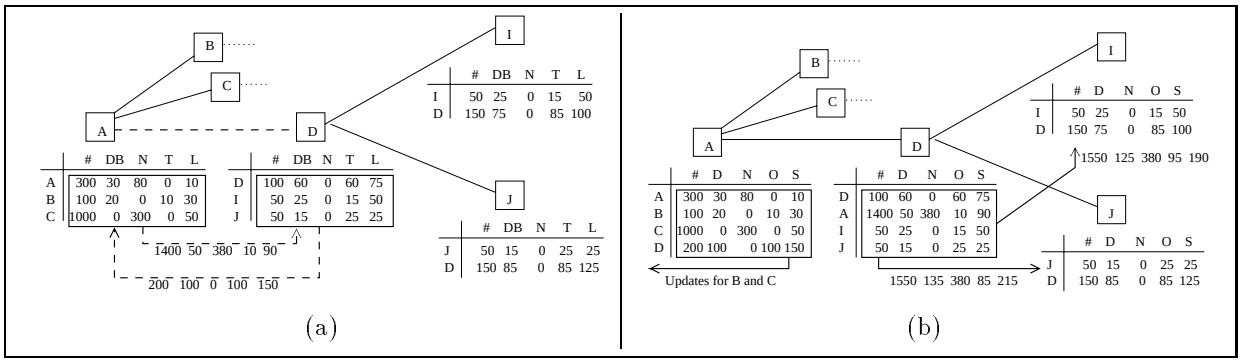

Figure 5: Creating a Routing Index

as shown in Figure 5b. Then I and J update their RI by replacing the row for D with the new information (not shown in the figure). If I and J were connected to nodes other than D, they would have to send an update to those nodes too.

4.3 Maintaining Routing Indices

The process of maintaining RIs is identical to the process used for creating them. To illustrate, let us suppose now that client I introduces two new documents about "languages" in its database. To update the RIs of its neighbors, I summarizes its new local index, aggregates all the rows of its compound RI (excluding the row for D), and sends this information to D. Then D replaces the old row for I with the received aggregated RI. In turn, D computes and sends new aggregates to A and J. When receiving the update, A and J update their RIs and compute new aggregates for their neighbors, and so on. For efficiency, we may delay exporting an update for a short time so we can batch several updates, thus, trading RI freshness for a reduced update cost. We can also choose not to send updates when the difference between the old and the new value is not significant. By not sending minor updates, we can again trade reduced update cost for accuracy of the RI.

Finally, a special but frequent update case occurs when a node disconnects from the network. To illustrate, let us suppose that I disconnects from the network. Node D detects the disconnection and updates its RI by removing the row for I. Then D informs its neighbors of the change on the number of documents it can access by sending new aggregates of its RI to them. In turn, the neighbors of D update their RIs and propagate the new information to their neighbors. Note, that we did not need I's participation (or the participation of any other neighbor) in the disconnection process. Not requiring the participation of a disconnecting node is an important feature in a P2P system where nodes can come and go at will.

Observe that the creation and update of RIs can be expensive operations as an update sent to a neighbor prompts it to send an update to its neighbors, which in turn prompt them to update their own neighbors, and so on. By using RIs, we trade the higher cost of maintaining the RIs for a lower cost when processing

```
RI Creation/Update Algorithm
Input: I
Output: An updated RI (as a side effect)

// Creation phase
RI[0] = Summary(I)
For each neighbor j:
        Send(j, RI[0])

// Update phase
OldRI = RI
While (true):
        Wait for a batch of aggregated RIs to arrive and/or
           for the local index to change
        For each aggregated RI (if any), A_k, received from neighbor k:
                RI[k] = A_k
        If local index changed:
                RI[0] = Summary(I)
        If OldRI is different enough than RI:
                OldRI = RI
                For each neighbor j in RI:
                        A = Aggregate(RI[(0...j-1, j+1..)])
                        Send(j, A)
```

Figure 6: RI Creation/Update algorithm

queries. In some systems this trade-off may be worth it, while in others it may not. Our experiments in Section 8 suggest that in many systems, our approach does reduce the total cost, and therefore it is worth building RIs. In the experimental section, we will show that for ratios of updates to queries as high as 1 update for every 30 queries, our approach reduces the total cost.

5 Algorithms and Data Structures for Routing Indices

In this section, we formalize the algorithms and data structures for RIs. We present the general algorithms for RI creation, update, and usage and we explore alternatives for the main components of an RI.

5.1 Algorithm for creating and updating RIs

We present the algorithm for creating and updating RIs in Figure 6. This algorithm is started by a node when it joins the P2P system and remains running until the node disconnects. The input to the algorithm is the local index (I). The algorithm uses the following functions: $Summary()$, which summarizes the local

```
Answer Query
Input:   A query Q, a client C, and a node F
Output: Query answers, Query forwards

Send local answers for Q to C
If StopCondition():
      return
Rank = [ ]
For each neighbor j except F:
      Rank.Append([j, Estimator(RI[j])])
      Sort tuples in Rank by second attribute
for t in Rank:
      Send query to neighbor t[0]
      If StopCondition():
            return
```

Figure 7: RI Usage Algorithm

index I; $Aggregate(RI)$, which aggregates the RI; and $Send(i, A)$, which sends A, an aggregated RI, to neighbor i. In the algorithm, we are assuming that RI is an array where each row i stores the aggregated RI of neighbor i. Row 0 is reserved for the summary of the local index.

The Creation/Update algorithm is divided in two phases. In the first phase, the creation phase, a newly connected node sends a summary of its local index to its new neighbors. In the second phase, the update phase, the node waits for update messages or for changes on its local index. As stated before, the node may wait for a set of changes to batch them together or it may ignore updates that are not significant in order to reduce cost. After updating its RI, the node sends a new aggregate RIs to all its neighbors. The aggregated RIs contain all rows except the row from the neighbor to which the aggregated RI is sent.

5.2 Algorithm for Answering Queries

We present the algorithm for answering queries in Figure 7. This algorithm is run every time a new query is received (either submitted by a user or forwarded by another node). The input of the algorithm is a query Q, a pointer to the client C that issued the query, and a pointer to the neighbor F that forwarded the query (if any).

The algorithm starts by attempting to answer the query using the local database. If not enough local results are obtained to reach the stop condition, then the algorithm ranks all the neighbors by using the $Estimator(Q, RI[i])$ function (this function takes a query Q and a row $RI[i]$ and computes the goodness of neighbor i with respect to that query). Then the node sends the query to each neighbor sequentially, checking if the stop condition was reached whenever each query returns.

	1 Hop					2 Hops				
Neighbor	#	DB	N	T	L	#	DB	N	T	L
X	60	13	2	5	10	20	10	10	4	17
Y	30	0	3	15	12	50	31	0	15	20
Z	5	2	0	3	3	70	10	40	20	50

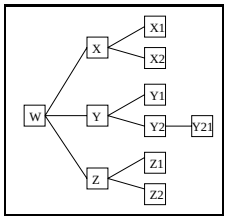

Figure 8: A sample Hop-count RI for node W

6 Alternative Routing Indices

6.1 Hop-count Routing Indices

In this subsection, we present an alternative data structure for an RI: a hop-count RI. The main limitation of the compound RI is that it does not take into account the number of "hops" (query forwardings) required to find documents. In the hop-count RI we stored aggregated RIs for each "hop" up to a maximum number of hops. We call this number the *horizon* of the RI. We show in Figure 8 a sample hop-count RI with a horizon of 2 hops. The node with this hop-count RI has three neighbors: X, Y, and Z. With one hop via neighbor X, the node can find 60 documents, out of which 4 are about databases, 2 about networks, 5 about theory, and 10 about systems. The node can also find 20 more documents through X with 2 hops (i.e., at X's neighbors). Note, that we do not have information beyond the horizon with this kind of RI.

The estimator of a hop-count RI needs a cost model to compute the goodness of a neighbor. For example, neighbor X may be preferable over neighbor Y for a query on topic "DB," as through X we would find 13 results with one hop, while it would require two hops to find that many results through Y. On the other hand, we can find more results (31) when going through Y.

If we define cost in terms of number of messages (we will expand on the notion of cost in Section 8), then we can define the goodness of a neighbor as the ratio between the number of documents available through that neighbor and the number of messages required to get those documents. So a neighbor that allows us to find 3 documents per message is better than a neigbor that allows us to find 1 document per message.

A simple model that allows us to compute this ratio is the regular-tree cost model. The construction of this model assumes that document results are uniformly distributed across the network and that the network is a regular tree with fanout F (i.e., all nodes are connected to exactly $F + 1$ nodes, except the ones at the leaves which are connected to only one node). Under these assumptions, it takes one message for a client to find all documents at the root of the tree (zero hops), $1 + F$ messages to get all documents at zero or one hops (one initial message for the root and one message for each of the F neighbors of the root), $1 + F + F^2$ to get all documents at zero, one, or two hops, and so on. Therefore, if we know that there are 10 documents in a node, 20 two hops away, and 100 three hops away, it would require 1 message

to get the first 10 documents, F additional messages to get the 20 documents, and F^2 additional messages to get the last 100 documents. Given this, we can compute the number of documents per message by dividing the expected number of result documents at each hop by the number of messages needed to find them. Formally, we define the goodness ($goodness_{hc}$) of $Neighbor_i$ with respect to query Q for hop-count RI as: $goodness_{hc}(Neighbor_i, Q) = \sum_{j=0..h} \frac{goodness(N_i[j],Q)}{F^{j-1}}$, where h is the horizon of the hop-count RI, $goodness()$ is the estimator for CRI, and $N_i[j]$ is the RI entry for j hops through $Neighbor_i$.

In our example, if we assume $F = 3$, the goodness of X for a query about "DB" documents would be $13+10/3 = 16.33$ and for Y would be $0+31/3 = 10.33$, so we would prefer X over Y. These values are based on the assumption that the network topology is a regular tree and that documents are uniformly distributed (the regular-tree cost model). If this assumption is not true, we can still use the formula above as a heuristic.

Let us turn now to the problem of how to create and update hop-count RIs. A node that needs to notify its neighbors of an update in its database first builds the aggregated RI in the same fashion as with the compound RI. Then it shifts the columns to the right, so the entries for 1 hop become the entries for 2 hops, the entries for 2 hops become those for 3 hops, and so on. The entries in the last column of the original RI are discarded and the summary of the local index is placed as the first column of the new table (i.e. as the one-hop entry). This new aggregated RI is sent to all neighbors which in turn update their RIs.

6.2 Exponentially aggregated RI

The hop-count RI is effective in taking into account the number of hops. However, this benefit comes at a higher storage and transmission cost than the compound RI. Moreover, in Section 8.2 we will see that the hop-count RI performance is negatevely affected by the lack of information beyond the horizon (a hybrid CRI-HRI overcomes this disadvantage, but it still does not solve the storage and transmission cost problem). In this subsection we present an alternative index structure, the exponential aggregated RI, that overcomes these shortcomings at the cost of some *potential* loss in accuracy.

The exponentially aggregated RI stores the result of applying the regular-tree cost formula to a hop-count RI. Specifically, each entry of the ERI for node N contains a value computed as: $\sum_{j=1..th} \frac{goodness(N[j],T)}{F^{j-1}}$, where th is the height and F the fanout of the assumed regular tree, $goodness()$ is the Compound RI estimator, $N[j]$ is the summary of the local index of neighbor j of N, and T is the topic of interest of the entry.

We show in Figure 9 an exponentially aggregated RI computed from our sample network of Figure 8. In the figure, we assume that the neighbors of X, Y, and Z are leaf nodes and that the fan out of the tree is 3. The entries for topic "DB" for X and Y have the values $13 + 10/3 = 16.33$ and $0 + 31/3 = 10.33$.

The exponential RI shares the assumptions of the regular-tree cost model and may not be realistic in some configurations, but it can still be used as an approximate index. There is a fundamental difference between the exponential RI and the hop-count RI. While the hop-count RI does not have any information

Path	#	DB	N	T	L
X	66.67	16.33	5.33	6.33	15.67
Y	46.67	10.33	3.00	20.00	18.67
Z	28.33	5.33	13.33	9.67	19.67

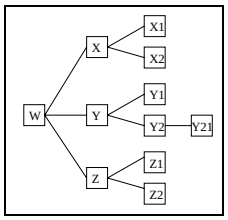

Figure 9: A sample Exponential Routing Index for Node W

beyond the horizon, with the exponential RI we can keep information for all nodes accessible from each neighbor in the RI. In fact, we will see in Section 8.2 that the exponential RI outperforms the hop-count RI in most cases.

To update exponential RIs, the node that needs to send an update to a neighbor adds up all rows (except the one associated with the neighbor to which the update vector is sent), multiplies the resulting vector by $1/F$, and adds the goodness of the summary of its local index. When the neighbor receives this update vector, it replaces the row of the sending node with the new vector and propagates the updates to its neighbors. Note that for updating exponential RIs it is essential to propagate updates only when the new and old values of the vector are "different" enough (for example by requiring that the Euclidean distance between the two vectors is greater than a certain number). If we do not do this, the exponential RI would propagate updates to all nodes in the network.

7 Cycles in the P2P Network

In this section we analyze how cycles affect the process of creating and updating RIs as well as strategies to minimize those effects. To illustrate the effect of cycles, we will use the initial setup of Figure 3, but with the network depicted in Figure 10 (with the cycle $A - B - E - G - C - A$). Let us assume that node A adds to its database two new "theory" documents and sends a new aggregate of its RI to B. Node B sends the update to its E and F neighbors, prompting E to send an update to G, which sends an update to C, which finally sends an update to A. When A receives this update, it will mistakingly assume that more "theory" documents are available via node C, but those extra documents are its own. Worse than that, the update from C will prompt A to send an update to its neighbors, informing them that they can access two more theory documents, creating an infinite loop. Even if A is able to recognize that the update from C originated from itself (and discard it), the cycle will lead to overcounts in other nodes as, for example, node C will receive an update from both A and G.

There are three general approaches for dealing with cycles: We ignore the problem alltogether and pretend that cycles never occur in the P2P network (no-op solution). We use a connection protocol that prevents cycles from forming in the P2P network (cycle avoidance solution). We allow cycles but take

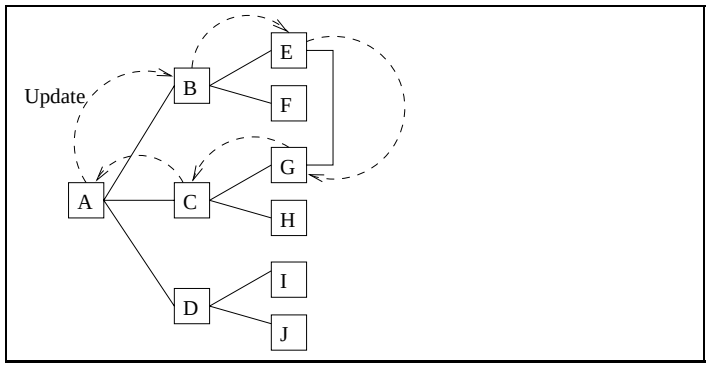

Figure 10: Cycles and Routing Indices

actions to recover from their effect (cycle detection and recovery solution). Next we examine each of the above approaches in more detail.

No-op solution: This solution only works with the hop-count RI and the exponential RI (if there are cycles, the compound RI algorithms can be trapped in an infinite loop).

In the case of the hop-count RI, cycles longer than the horizon will not affect the RI. However, shorter cycles will affect the hop-count RI but their effect will be limited if we are using the regular-tree cost model. To illustrate, consider a simple P2P system with three nodes A, B, and C. Initially A is connected to B, and B is connected to C. A has 10 documents, while B has 15, and C has 20. The initial hop-count RI of A for a given topic is shown on the left side of Figure 11. Let us assume now that C establishes a new connection with A. This new link causes a series of updates that result in the hop-count RI shown on the right side of Figure 11. Note that now A believes that it can get 10 documents at three hops from B, however, those 10 documents are the documents present in A. Similarly, the documents reachable via 4 and 5 hops are repeated counts of documents available at 1 or 2 hops. Let us now quantify the effect of those extra columns on the goodness of B. If we use the regular-tree cost model, the goodness of B, before the cycle was created, was 21.67 $(15 + 20/3)$. After the cycle is created, the goodness increases to 23.58 $(15 + 20/3 + 10/9 + 15/27 + 20/81)$. The difference between the two values is the error introduced by the cycle. This increase represents a relative error of only 9%. However, the cycle increases the cost of creating/updating the hop-count RI as updates sent by a node return to it (via the cycle), causing the node to send a new update to all its neighbors (which in turn send the update back to the node again). The cycle is broken when the update reaches the horizon of the hop-count RI.

Similarly, in the case of the exponential RI, updates are sent back to the originator. However, the effect of the cycle will be smaller and smaller every time the update is sent back (due to the exponential decay), until the difference between the old update and the new update is small enough and the algorithm stops propagating the update. To illustrate, in the scenario of Figure 11, if we stop propagating updates after

	Hops				
Neighbor	1	2	3	4	5
B	15	20	0	0	0

Initial State

	Hops				
Neighbor	1	2	3	4	5
B	15	20	10	15	20
C	20	15	10	20	15

After C-A connection

Figure 11: Effects of a cycle on the hop-count RI

there are no changes to the first decimal digit of the goodness of the neighbor, then the goodness of B will be 23.64 ($15 + 20/3 + 10/9 + 15/27 + 20/82 + 10/243 + 15/729$), resulting in a similar relative error of 9%. As in the previous case, the main effect of the cycle is the increase in cost of creating/updating the RI.

Cycle avoidance solution: In this solution we do not allow nodes to create an "update" connection to other nodes if such connection would create a cycle. The techniques for cycle avoidance have been extensively studied (see [TW99] for a survey) and we do not cover it further in this paper. The main disadvantage of this approach is that in the absence of global information we may end with a suboptimal update network.

Cycle detection and recovery: This solution detects cycles sometime *after* they are formed and, after that, takes recovery actions to eliminate (or neutralize) the effect of the cycles. In the example of Figure 10, cycles can be detected by having the originating node of a query or an update, let us say A, include a unique message identifier in the message. Any update (or query forwarding) that any other node sends as a consequence of this message will have the original message identifier. If a message with the same identifier returns to A (let us say from C), then A knows that there is a cycle and that a recovery procedure should be started. A simple recovery procedure is for A to exclude C from its RI and to ignore any update messages coming from the link with C. However, this recovery procedure may be reducing the accuracy of the RI. For example, let us assume that B modifies its local database and sends updates to its neighbors A, E, and F. In turn, E will send updates to G which will send an update to C. But, simultaneously, A will send the same update to C. C knows that the updates from A and G are the same and that it should discard one of them. How can C choose the one to discard? Unfortunately, C by itself does not have enough information to make a reasonable decision, especially if updates are batched. Thus, if C discards the update from A, it may be discarding the shortest path to B.

8 Experimental Results

In this section, we evaluate search mechanisms for P2P systems. First, we present our model of a P2P system. Then we introduce a simulation tool that allows us to evaluate different search mechanisms efficiently. We

then use our tool to study the performance of different mechanisms as well as the factors that affect their performance. We close the section with an analysis of the cases where RIs can be used effectively.

8.1 Modeling search mechanisms in a P2P system

Our goal in this subsection is to identify the elements of a typical search mechanism in a P2P system, so we can model each element and study its impact on performance. A typical P2P system is a network of nodes T where each node contains a set of documents. Users send requests consisting of a query q and a *stopCondition* to a node of the P2P system. The objective of the search mechanism is to answer those requests by obtaining a set of documents of size *stopCondition* that matches the query q. In addition, search mechanisms allow for updates such as the addition of nodes or new documents.

To process queries and updates we use the mechanisms described in the previous sections (CRIs, HRIs, ERIs). For comparison purposes, we add an additional mechanism: *No-RI*. Instead of using an RI to choose the best neighbor to forward a query, this search mechanism simply chooses a random neighbor.

To further model the elements of a search mechanism, we need to define the topology of the network, the location of document results, cycle policies, and how costs are measured.

The topology of the network defines the number of nodes and how they are connected. In our model, we consider three kinds of network topologies: a tree, a tree with added cycles, and a power-law graph. The first topology, a tree, is of interest because it does not have cycles (a good base case for our algorithms). For the second topology, we start with a tree and we add extra vertices at random (creating cycles) so we can measure the impact of cycles on the search mechanisms. The third topology, a power-law graph, is considered a good model for P2P systems and allows us to test our algorithms against a "realistic" topology [FFF99].

We model the location of document results using two distributions: uniform and an 80/20 biased distribution. Under the uniform distribution all nodes have the same probability of having each document result (nodes can have more than one document result). The second distribution assigns uniformly 80% of the document results to 20% of the nodes (and the remaining 20% of the documents to the remaining 80% of the nodes).

Modeling the cost of a search mechanism is a complex task. We can model the cost based on the resources used in the P2P system (e.g., network, storage space, or processing power) or based on the user experience (e.g., mean query response time, query throughput, or query turnaround time). In current P2P systems, the critical resource is the network [Clia] as many of the nodes are connected through links with limited bandwidth (e.g., dial-up connections, DSL, and cable). Therefore, in this paper we focus on the network and we use the number of messages generated by each algorithm as a measure of cost. This is not to say that user-based factors (such as response time) are not important, but by focusing on the network bottleneck we are also improving those factors.

Parameter Name	Description	Base Value
Network Configuration Parameters		
$NumNodes$	Number of nodes in the network	60000
T	Topology of the P2P network	tree
F	Branching factor for tree topology	4
EL	Extra links to add cycles to tree topology	10
o	Outdegree exponent for power law	-2.2088
Document Distribution Parameters		
QR	Number of Query Results available through all local indices	3125
D	Distribution of documents for query q	80/20
RI Parameters		
$Creation_{size}$	Average size of an RI creation/update message	1000 bytes
$Query_{size}$	Average size of a query message	250 bytes
$StopCondition$	Number of documents requested	10
H	Horizon for HRIs	5
A	Decay for ERs	4
c	RI Compression	0%
$minUpdate$	minimum percentage difference for updates to propagate	1%

Figure 12: Simulation Parameters

8.2 Evaluating P2P Search Mechanism Performance

In this section we experimentally compare the three proposed RIs: compound RI (CRI), hop-count RI (HRI), and exponential RI (ERI) against each other and against the No-RI search mechanism. We also explore how the performance of the RIs are affected by approximate indices, different stop conditions, document result distribution, number of document results, and network topology.

To evaluate search mechanisms we built a simulator. The simulator receives as input a P2P model and an operation that can be an update or a query. The simulator iterates over different network topologies and document result locations, and outputs the average number of messages necessary to perform the operation plus a confidence interval. All results were computed with at least a 95% confidence interval of having a relative error of 10% or less. Parameters are set to the base values presented in Figure 12 unless stated otherwise. In Appendix A, we explain in detail the operation of the simulator as well as the choice of the base values for the parameters.

Figure 13 shows the number of messages needed to process a query when using each kind of RI for two document distributions. The advantage of using RIs is obvious, we are able to reduce the number of messages by half when compared to not using RI. Among the RIs, CRI had the best performance, followed by the ERI and HRI. This difference in performance is a function of the number of nodes used to generate the index. In particular, CRI uses all nodes in the network, HRI uses nodes within a predefined a horizon, and ERI uses nodes until the exponentially decayed value of an index entry reaches a minimum value (resulting in using more nodes than HRI, but fewer than CRI). This result shows that the more nodes an RI uses to compute

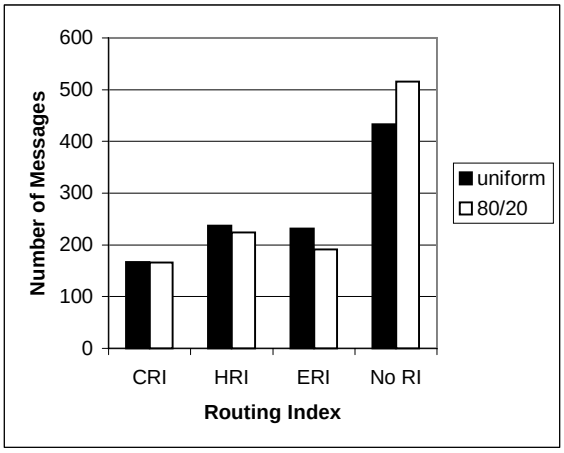

Figure 13: Comparison of CRI, HRI, and ERI

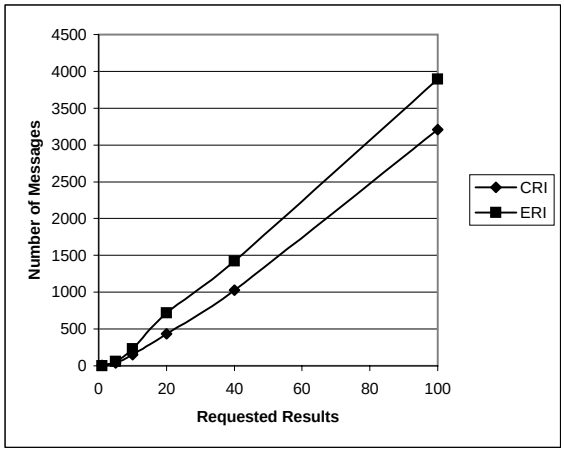

Figure 14: Number of Results

the goodness of a path, the better the RI is. However, we will see that a larger number of nodes implies a higher update cost.

In Figure 13 we also present the effect of using two document distributions, an 80/20 biased and a uniform document distribution. Surprisingly, a 80/20 biased distribution does not improve the performance of RIs much, but it degrades the performance of a No-RI search mechanism. To understand this result, we analyzed traces of our simulations. In the case of RIs under a 80/20 document distribution, the algorithm directed the queries to nodes with a high number of document results, but to reach those content-loaded nodes the queries needed to travel through several nodes that had very few or no document results. On the other hand, under a uniform document placement, the algorithms followed good paths where at each node it obtained a few results. In summary, in one case, we collected results by traveling over an almost empty path to a full node, while in the other case, we collected results by traveling through a path of a similar length where each node contributes a few documents. The overall result was that the number of messages per document result was about the same in both cases. In the case of the No-RI approach, an 80/20 document distribution penalizes performance as the search mechanism needs to visit a significant number of nodes until it finds a content-loaded node (generating a large number of messages in the process).

We also compared RIs against non-index/flooding solutions such as Gnutella. In that case, RIs reduce the number of messages by two orders of magnitude (graph not shown). However, this comparison is not completely fair as Gnutella-like systems find all results in the section of the network they explore (versus only a user-defined number of results when using RIs) and they potentially have a better response time (as queries are processed in parallel, rather than sequentially). However, finding *all* results may be an overkill for most applications as users rarely examine more than the first 10 top results returned by a search engine [Bri98]. In addition, low response times may be hard to achieve in Gnutella-like systems because of network contention created by tens of thousands of messages generated by each query.

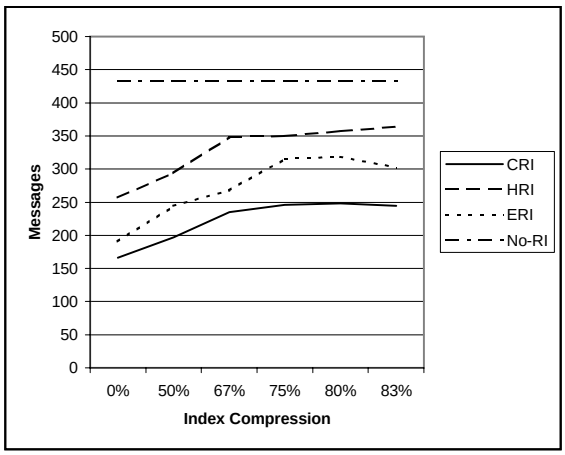

Figure 15: Effect of Overcounts

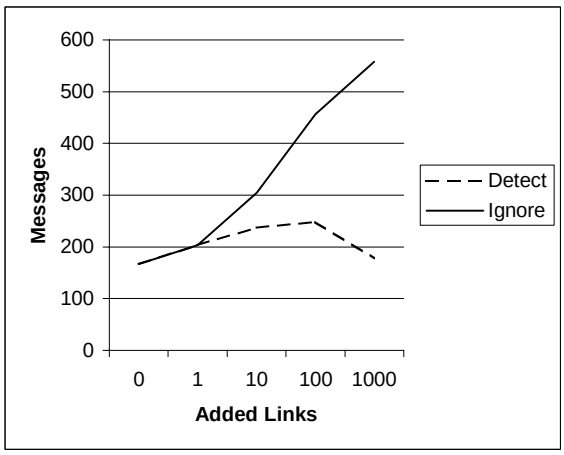

Figure 16: Effect of Cycles

We now investigate how an increase in the requested number of documents affects RIs. Figure 14 shows the number of messages generated for different requested result sizes when using CRIs, HRIs and ERIs (the performance of HRI is indistinguishable from ERI, so it is omitted from the graph). In the figure, we observe that the higher the number of requested documents, the more messages are generated from the other, so it is omitted from the graph. the An encouraging fact is the linear shape of the increase, showing that all RIs, as well as No-RI, scale well on this parameter. We also analyzed the effect of a decrease on the number of document results available (graph not shown). In that case, we obtained a very similar graph where the number of messages grows linearly with a reduction of document results.

We now investigate how errors in RIs, and particularly overcounts, affect RI performance. As discussed in Section 4, errors can occur in a variety of ways; here we select one scenario to illustrate. We assume that documents are organized into categories, and the index is a hash table of categories. Several categories may hash to the same bucket, so the count in a bucket represents the aggregate number of documents in those categories. For example, suppose there are 3 "database" documents, and 2 "network" ones. If "database" and "network" hash to the same bucket, the consolidated "database/network" bucket will have a count of 5 documents. If a query is looking for "database" documents, when using the RI, we will believe that they are 5 of them, when in reality there are only 3 (an overcount). (Instead of adding the original document counts, we could have also chosen to take the minimum of them, generating undercounts; or we could have averaged them, generating mixed errors.)

As the table size is reduced, more and more overcounts occur. In Figure 15, we show the performance of CRI, HRI and ERI, as a function of the "index compression." For example, a 50% value means that the number of hash table buckets is half the number of categories, while 83% represents a table with one-sixth the categories. (Note that the scale is not linear.) From the graph, we can see that even though there is a loss of performance because of overcounts, this loss is modest even in the case of significant reductions on

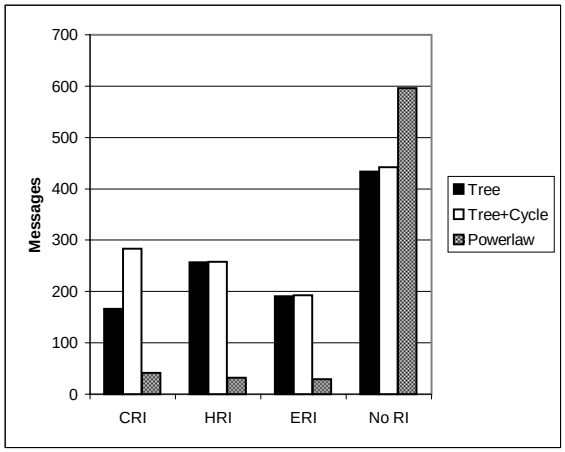

Figure 17: Network topology

the size of the index. Moreover, query processing when using RIs is still far cheaper than No-RI even if we use the highest compression levels. We conducted additional experiments for undercounts and mixed errors as well as for other error models. Those experiments had similar results to the one presented here and are omitted for brevity.

In Figure 16 we study how ERIs perform when cycles are added to a tree network. Cycles are created by adding random links to a tree network with $NumNodes - 1$ links. As expected, the number of messages increases as we add more links and cycles are created. The increase in the traffic is the result of two factors. First, there is a loss of accuracy of the RI. In the case of the "detect and recover" policy, this loss is the result of missing the best route to the results (as explained in Section 7), and in the case of the "no-op" policy the accuracy suffers because overcounts introduced in the generation of the RI. Second, during query processing the number of messages increases. In the case of the "detect and recover" policy, those extra messages are the result of return-queries messages sent by a node that detects a cycle. In the case of the no-op policy, extra messages are generated when we traverse a cycle more than once, finding document results that were already found in a previous iteration. In the figure, we observe that the increase in the number of messages is small if we use the "detect and recover" policy, but it can be significant if we choose to ignore cycles. An unexpected result is that the number of messages drops if we add a large number of links. This drop is the result of the added connectivity that additional links create, which allows shorter routes to document results. Similar performance to the one presented for ERI is shown by HRI and CRI (when using the ignore-detect policy, as CRI is not guaranteed to terminate when using the no-op policy).

In Figure 17 we study how RIs perform in different network topologies. The result of our analysis is surprising at first glance: RIs perform better in a power-law network than in a tree network. There are two reasons for this result. First, while in a tree-like network the connectivity of every node (except leaf

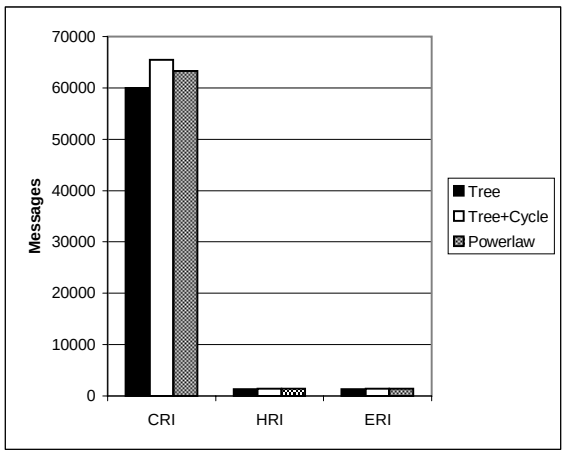

Figure 18: Updates and Network Topology

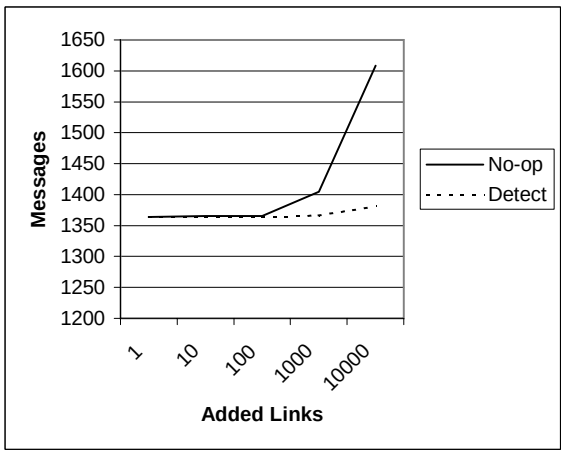

Figure 19: Updates and Cycle Policy

nodes) is the same, in a power-law network a few nodes have a significantly higher connectivity than the rest. By analyzing the traces of our simulation, we found that the query algorithms actually direct the queries towards those well-connected nodes. After getting to these highly connected nodes, a large number of results is collected without having to issue many messages. The second reason for this performance improvement is that power-law distributions generate network topologies where the average path length between two nodes is lower than in tree topologies. Lower path length improves performance as we need less messages to go from node to node. On the other hand, these same two factors hinder the performance of the No-RI approach. In a power-law network, there are very few highly connected nodes and it is not easy to find them if we just move randomly as No-RI does. As a result, the No-RI approach visits a significant number of nodes until it finally stumbles onto a highly connected node (generating a large number of messages in the process). Shorter path length also hinders No-RI as bad decisions about which neighbor to contact often result in return-query messages.

Figure 18 shows the number of messages needed to update each kind of RI for each network topology. The graph shows the cost of one batch of updates, propagated throughout the network. In the graph we can see that the cost of CRI is much higher when compared with HRI and ERI. This is the result of CRI propagating the update to all nodes, while HRI and ERI only propagate the update to a subset of the network. This result confirms that the additional information and better query performance of CRIs come with a high price tag. On the other hand, HRIs and ERIs have very low update costs and their query processing performance is very close to the one of CRIs, making them an excellent choice as the search mechanism of a P2P system. In the graph we can also see that network topology has little impact on the update performance of RIs, as there is low or no correlation between the network topology and the number of nodes that needs to be updated.

A concern about using ERIs is that their performance may be affected significantly by cycles as we may

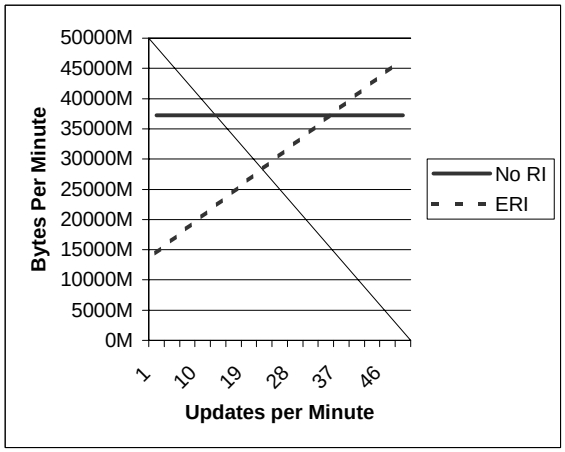

Figure 20: Updates per minute

have to loop through nodes several times until the value of the update has been exponentially decayed enough to be considered insignificant. Figure 19 shows the number of update messages generated for two different cycle policies on different tree+cycle network topologies. As in a previous experiment, we create a tree+cycle topology by starting with a tree network and adding random links to create cycles. In this experiment, we consider significant all updates that may change the current index value by more than 1%. As expected, the number of messages increases as we add more links, but in both cases the increase is modest (although the increase is more rapid when cycles are ignored). In conclusion, ERIs are not expensive to use in a network with cycles even if we propagate updates with an impact as low as 1%.

As stated earlier, building RIs is advantageous only if enough queries can make use of the RIs before we have to update the indices. To compare query and update costs, we need to count bytes transmitted, rather than simply messages, since query messages are shorter. In Figure 20 we show the number of bytes transmitted per minute, for an ERI system and for a no-RI one. We assume that the system as a whole (or is it a single node?) is processing 1032 queries per minute, and the horizontal axis shows the number of updates per minute. The number 1032 corresponds to the observed average query load by a Gnutella node run at Stanford [YGM01a]. Thus, when say 10 updates per minute are run, there are roughly 100 queries for each update. For our comparison, we assume that query messages are 70 bytes [Clib], and updates messages are 3500 bytes (hash table with 1750 buckets of 2 bytes each). As we see in the figure, as the number of updates increases, the overall communication cost for the ERI scheme increases. The cost for the no-RI scheme remains constant since there are no RIs to update. The crossover point is 36 updates per minute. That is, as long as there are fewer than 36 updates per minute, using an RI pays off. In practice we would expect the number of updates to be way below 36 per minute, especially since it is not that critical to keep indexes up to date and updates can be batched together.

9 Conclusions

In this paper we studied how to improve the efficiency of content search in a peer-to-peer system. We achieve greater efficiency by placing Routing Indices in each node. Three possible RIs: compound RIs, hop-count RIs, and exponential RIs were proposed and experimentally evaluated using simulations. From our experiments we conclude that ERIs and HRI offer significant improvements versus not using an RI, while keeping update costs low. We belive that routing indices, and in particular ERIs and HRIs, can help improve the search performance of current and future P2P systems.

References

[Bel57] R.E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[Bri98] S. Brin. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th WWW Conference*, 1998.

[Clia] Clip2.com, world-wide web: http://dss.clip2.com/gnutella.html. *Gnutella: To the Bandwidth and Beyond*.

[Clib] Clip2.com, world-wide web: http://www.clip2.com/dss_barrier.html. *Bandwidth Barriers to Gnutella Network Scalability*.

[FF62] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[FFF99] M. Faloutsos, P. Faloutsos, and C Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, 1999.

[GGM95] L. Gravano and H. Garcia-Molina. Generalizing gloss for vector-space databases and broker hierarchies. In *Proceedings of the Tweenty-first International Conference on Very Large Databases (VLDB'95)*, 1995.

[GGMT94a] L. Gravano, H. Garcia-Molina, and A. Tomasic. The effectiveness of gloss for the text-database discovery problem. In *Proceedings of the 1994 ACM International Conference on Management of Data (SIGMOD'94)*, 1994.

[GGMT94b] L. Gravano, H. Garcia-Molina, and A. Tomasic. Precision and recall of gloss estimators for database discovery. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS'94)*, 1994.

[GGMT00] L. Gravano, H. Garcia-Molina, and A. Tomasic. Gloss: Text-source discovery over the internet. *ACM Transactions on Database Systems (TODS'2000)*, 2000.

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, 2000.

[Kos00] D. Kossman. The state of the art in distributed queyr processing. *ACM Computing Survey*, September 2000.

[Mil98] C. K. Miller. *Multicast Networking and Applications.* Addison Wesley, 1998.

[PS00] C. Palmer and J. Steffan. Generating network topologies that obey power laws. In *GLOBECOM*, 2000.

[Tan96] A. S. Tanenbaum. *Computer Networks.* Prentice Hall PTR, 1996.

[TW99] A. Tanenbaum and A. Woodhull. *Operating Systems Design and Implementation.* Prentice-Hall, Inc., 1999.

[Wora] World-wide web: http://freenet.sourceforge.com. *Freenet Home Page.*

[Worb] World-wide web: http://gnutella.wego.com. *Gnutella Development Page.*

[Worc] World-wide web: http://setiathome.ssl.berkeley.edu. *Seti At Home Page.*

[Word] World-wide web: http://www.napster.com. *Napster Home Page.*

[YGM01a] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *Proceedings of the Tweenty-first International Conference on Very Large Databases (VLDB'01)*, 2001.

[YGM01b] B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. Technical report, Stanford University, October 2001. Submitted to ICDCS 2002.

Appendix A: Simulating a P2P System

In this appendix we described how our simulator works as well as the choice for the base values of the parameters.

The simulator starts by generating a network topology. Then it distributes results among the nodes, picks at random a node that will initially receive the query or update, and creates the necessary RIs. To build the RIs, we do not need to use the full-fledged algorithms presented in Section 5.1, as we know from which node the query will be issued. Instead, we use a version of the algorithm that only updates RI entries for neighbors "downstream" from the node picked as the originator of the query. Cycles are possible when creating RIs. We consider two possible cycle policies: no-op solution and detect and recover (we do not consider the cycle avoidance solution as it has the same effect as the detect and recover policy when processing queries and updates). As the name implies, in the no-op solution we do not do anything special to deal with cycles. For the detect and recover policy, nodes keep track of the queries and updates that they have processed. If a query or update reaches a node for a second time (due to a cycle) the message is not forwarded any further (breaking the cycle).

We generate errors to simulate approximate indices in two ways. In one scenario, we assume that RIs were implemented as hash tables with document counts in each bucket. We generate overcounts by consolidating buckets of the original hash tables and adding their document counts (under the motivation that by consolidating buckets we reduce the size of the index). In the second scenario, errors are generated by using a normal distribution with mean 0 and variance v (making errors positives for overcounts, negative for undercounts, and unchanged for mixed). In all cases, errors are propagated on updates, so they compound from node to node (as they would do in a real system).

In the case of a query, after building the network, distributing the documents, and building the RIs, we send a message to the selected initial node with the number of documents requested (*stopCondition* in this case). The node finds how many local results are available (if any), subtract the number of local results from the number of requested documents and if this number has not reached zero, the node sends the request sequentially to each of its neighbors in the order prescribed by the RI (with the number of requested document set to the number received minus the number of documents found). In turn, a node that receives a forwarded request does the same procedure until the number of requested documents reaches zero (we are done) or the node does not have more neighbors to which to forward the query (we cannot proceed further). In the later case, the query is returned to the neighbor that sent the query to that node and the neighbor sends the returned query to its next best neighbor according to the RI. During this process we count the three kind of messages that are generated: forwarded queries (when a node sends the query to a neighbor), returned queries (when a node sends back a query because it cannot be forwarded any more and the stop condition has not been met yet), and result messages (sent to the originator of the query telling them that a result can be accessed at a node). Note, that we are not counting the actual transmission of the answers as these messages are very application dependent and they do not depend on the performance of

the search mechanism. Similar than when building RIs, we treat cycles during query processing according to the policies prescribed by the P2P model.

In the case of an update, the initially selected node updates its RI, aggregates all rows and sends them to all of its neighbors (appropriately modified depending on the RI used). In turn, each neighbor checks if the changes in the RI are "significant enough" by comparing the percentage difference between the old and the new value against the $minUpdate$ parameter. If the changes are significant, it updates its RI, aggregates all columns and sends the aggregated vector to its neighbors. The stop condition for an update varies among the different RIs as described in the previous sections. As before, during the update process we apply the cycle policies defined in the model.

In the remaining of the appendix, we explain the choice for base values of the parameters in Figure 12. The first parameter, $NumNodes$ defines the size of the P2P network. In our simulations, we set $NumNodes$ to 60000, roughly double the maximum number of nodes found in a 2000 study of the Gnutella network [Clia]. Parameter T defines the topology of the network. To further define the three topologies in our model, we define F as the branching factor for the tree topology, EL as the number of links added to a tree to form a tree with cycles, and o as the outdegree exponent of a power-law network. In the simulator, we generate a power-law network by using the power-law out-degree generator [PS00]. We set the outdegree exponent to -2.20288 equal to the best fit for the Internet [FFF99]. The Document Distribution Parameters define how documents are distributed throughout the P2P system. For simplicity, we assume that all queries have the same number of results (QR). [YGM01a] found that about 5.2% of the nodes of the Gnutella network will have an answer for a given query, so we set this number to 3125 (5.2% of 60000 nodes). Parameter D defines the location of document results in the network. Finally, the RI Parameters define the size of RI tables and messages as well as the parameters needed for each RI algorithm.