A General Framework for Searching in Distributed Data Repositories

Spiridon Bakiras[1] Panos Kalnis[2] Thanasis Loukopoulos[3] Wee Siong Ng[2]
[1]Department of Electrical & Electronic Engineering
The University of Hong Kong
Pokfulam Road, Hong Kong
sbakiras@eee.hku.hk

[2]School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543
{kalnis, ngws}@comp.nus.edu.sg

[3] Department of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Hong Kong
luke@cs.ust.hk

Abstract

This paper proposes a general framework for searching large distributed repositories. Examples of such repositories include sites with music/video content, distributed digital libraries, distributed caching systems, etc. The framework is based on the concept of neighborhood; each client keeps a list of the most beneficial sites according to past experience, which are visited first when the client searches for some particular content. Exploration methods continuously update the neighborhoods in order to follow changes in access patterns. Depending on the application, several variations of search and exploration processes are proposed. Experimental evaluation demonstrates the benefits of the framework in different scenarios.

1. Introduction

The tremendous growth of Internet has simplified the process of publishing and sharing information. A variety of data, ranging from simple text files to entire scientific databases, and from still images to high quality sound and video streams, exists on-line. Searching in the numerous available data repositories is an essential function, which is complicated due to the unstructured nature of the Web.

Many data repositories follow a centralized architecture. Users pose their queries to the server, which searches its index and returns links to the sites that contain the results. Typical examples in this category are web search engines (e.g., Google) and music distribution systems (e.g., Napster). Changes of the source data are reflected to the central index either by periodically polling the source sites (e.g., web crawling [2]), or by allowing the sites to notify the server whenever updates occur.

Centralized systems suffer from several drawbacks: (i) they cannot follow high-frequency changes in the source data, (ii) they require expensive dedicated infrastructure (i.e., high-end server farms, fast network connections, etc.), (iii) they exhibit a single point of failure, (iv) legal reasons may prevent the accumulation of information at a central location (e.g., the case of Napster). As an alternative, several distributed systems have been proposed. In this case there is no form of centralized catalogue; instead, queries are propagated through the network and each node returns to the initiator the parts of the result that it may contain.

Several forms of distributed data repositories are already in use for a wide range of applications. An example is the Squid system [8], for cooperative web proxies. When a local miss occurs at some proxy, the proxy searches its neighbors for the missing page in order to avoid the delay of fetching the page from the corresponding server. Another case of distributed data repositories are music sharing systems (e.g., Gnutella). In this case the nodes of the repository are users' computers and the content is mp3 files. A request for a song is propagated through the network and a list of nodes that can serve it is returned.

The above systems have several differences. For example, in web caching a large proxy can serve requests from various smaller ones without forwarding any requests to them. In music sharing on the other hand, each neighbor of a node v is both incoming (i.e., sending requests to v)

and outgoing (i.e., receiving requests from v). Furthermore, the search process can be extensive (i.e., retrieving numerous nodes containing the requested result) or limited (i.e., terminating when the first result is found). It can be guided by the existence of local indexes representing the contents of other nodes (e.g., cache digests), or random.

Despite the differences, all forms of distributed search have some common characteristics. First the number of neighbors for each node is restricted in order to avoid the network and CPU overhead of processing an excessive number of requests. Second, in all cases the quality of content sharing is maximized by an appropriate choice of neighbors, so that nodes with similar access patterns (e.g., in Squid and PeerOlap) or interests (e.g., in Gnutella) are grouped together. Third, the neighborhood structure should continuously update the best neighbors of a node.

Motivated by these observations we propose a general framework for searching in distributed data repositories without centralized coordinators. The framework is based on the concept of dynamic reconfiguration, which adapts the neighborhood structure according to the evolving query patterns, and can capture all cases discussed above with appropriate parameter tuning.

The rest of the paper is organized as follows: Section 2 overviews related work on distributed data repositories. Section 3 describes the framework and the related search algorithms. Section 4 presents the case study on music-sharing scenario. Section 5 concludes the paper with a discussion on future directions.

2. Related Work

Searching in distributed data repositories is an important issue in Peer-to-Peer [1] (P2P) systems [9]. Such systems allow the sharing of resources among autonomous peers whose participation is ad-hoc and highly dynamic.

Here we are interested in P2P systems like Gnutella, where search is distributed. When a new peer P_N wishes to join the network, it first obtains the address of an arbitrary peer. A peer P broadcasts a query to all its neighbors, which propagate it recursively. If any of the visited peers contains a result, it sends it back to P via the reverse route. A peer can also broadcast exploration messages, when some of its neighbors abandon it (i.e., go off-line).

Yang and Garcia-Molina [10] observed that the Gnutella protocol could be modified in order to reduce the number of nodes that receive a query, without compromising the quality of the results. They proposed three techniques: (i) Iterative Deeping [2], where multiple search cycles are initiated with successively larger depth, until either the query is satisfied or the maximum depth d is reached. (ii) Directed BFT, where queries are propagated only to a beneficial subset of the neighbors of each node. (iii) Local Indices, where each node maintains an index over the data of all peers within r hops of itself, allowing each search to terminate after $d - r$ hops. All three techniques are orthogonal to our methods and can be employed in our framework in order to further reduce the query cost.

The network configuration in the above-mentioned systems is static. This fact introduces two major drawbacks: (i) peers with slow links become the bottleneck since numerous queries are propagated through them, and (ii) the relation between peers may become unbalanced, if a peer only requires, but refuses to provide any content.

Most of the existing P2P systems aim at content sharing. PeerOlap [3] is a P2P system for data warehousing applications. PeerOlap acts as a large distributed cache for OLAP results by exploiting underutilized peers. When a query is posed, the initiating peer decomposes it into chunks, and broadcasts the request for the chunks in a similar fashion as Gnutella. However, unlike Gnutella, PeerOlap employs a set of heuristics in order to limit the number of peers that are accessed. Missing chunks can be requested from the data warehouse. Towards minimizing the query cost, PeerOlap also supports adaptive network reconfiguration.

The previous discussion applies to ad-hoc dynamic P2P networks without any guarantee on the availability of resources. By allowing strong control over the topology of the network and the contents of each peer (e.g., special case where all peers belong to the same organization), queries can be answered within a bounded number of hops, since search is guided by a hash function. Chord [7], CAN [5] and Pastry [6] belong to this category. Such configurations are outside the scope of this paper.

3. Dynamic Reconfiguration

The general framework for dynamic neighbor reconfiguration includes: i) search in order to satisfy user requests, ii) exploration in order to identify new potential neighbors, iii) neighbor update, which is the process of selecting a new set of neighbors. In this section we illustrate each mechanism after providing some necessary definitions.

3.1. Neighbor Relations

Consider N repositories, connected through a network. We distinguish two kinds of neighboring relations between

[1] The term "P2P" has been used in the database literature to identify systems where each node may act both as a server and a client assuming static configuration [4]. Such systems are generalizations of the traditional client-server model and standard techniques can be applied. Here, "P2P" refers to dynamic systems with ad-hoc participation.

[2] In [10] the term "frozen queries" is used to indicate queries that will move one hop further at the next iteration of BFT. Throughout this paper, we use the same term with different meaning.

the repositories: *symmetric* and *asymmetric*. A relation is *symmetric* if both ends can forward requests to each other; otherwise, it is *asymmetric*. In the general case each repository v_i maintains two neighbor lists, one with the outgoing neighbors, to which it forwards its own requests (denoted by l_i), and one with the incoming neighbors, from which it receives requests (denoted by l'_i). The lists themselves contain all, or a subset of the available repositories (presumably due to limitations on the available bandwidth and processing capacity). The network is said to be consistent, if there does not exist a pair of nodes v_i, v_j such that $v_j \in l_i \wedge v_i \notin l'_i$, i.e., v_j is an outgoing neighbor of v_i, without v_i being an incoming neighbor of v_j. Depending on the list contents, three situations are of particular interest:

All to All lists: The list of outgoing (l_i) and incoming (l'_i) neighbors for each v_i contain all N repositories. Such a case happens, for instance, when the repositories are organized in a single multicast group [11]. In order to avoid unnecessary resource consumption, this category is applicable only for small values of N.

Asymmetric lists: When the lists l_i and l'_i contain different nodes, then the neighborhood relation is asymmetric. A case of special interest, which we call *pure asymmetric*, is when the capacity of the incoming list l'_i is N (i.e., every node v_i can be the outgoing neighbor of all repositories). Under the *pure asymmetric* assumption the neighbor network is always consistent, regardless of independent changes in the outgoing neighbor lists. This enables nodes to select neighbors based solely on their own criteria. An example of *pure asymmetric* lists can be found in the topmost level proxies of the Squid caching hierarchy [1], which are configured to accept requests from all low-level proxies without the opposite being true.

Symmetric lists: l_i and l'_i contain the same repositories. The case leads to symmetric relations for v_i, and is of practical interest whenever neighboring relations must assure that both ends benefit from cooperation. Notice that in the symmetric case any neighbor's (let v_j) addition in l_i or deletion from l'_i, will make the network inconsistent, unless being accompanied by the respective changes in l'_i and l_j. Therefore, neighbor configuration decisions can no longer be taken individually, but they must be the result of an "agreement" that involves a pair of repositories. As an example of symmetric relations, consider the Gnutella file sharing system, where each user (acting as a repository) is configured to send and accept queries from a fixed number of neighbors.

3.2. Search

A request arriving at a repository can either be satisfied locally, or propagated through the neighbor network until some node(s) return results, or a terminating condition is reached. A generic search algorithm is shown in Algo. 1, assuming that propagation stops at nodes that can serve the request. Notice, however, in some systems (e.g., music sharing), a node may still forward the request even if it can serve it, in order to maximize the number of the results.

```
On End-user Request Arrival
    If the request can not be satisfied locally
        Select one or more outgoing neighbors and
        forward the request /*use summary info if available*/
        Obtain results and update statistics
On Neighbor Request Arrival
    Send Reply /*either results or NOT FOUND*/
    If the request was not satisfied locally
        If propagation terminating condition is not met
            Select one or more outgoing neighbors
                and forward the request
```

Algo. 1. Search Algorithm

The algorithm distinguishes the case when i) a request arrives from an end-user, or ii) it arrives from another repository. The main parameters are the *propagation terminating criterion* and the *set of neighbors* where the request should be sent to. A common threshold in many distributed systems for the first parameter, is the maximum number of hops that a request may perform. Obviously, a large number favors extensive search at the expense of network overloading. The extent of the search process depends on whether there exists a central/alternative repository that can satisfy the request. For instance, in distributed web caching the web servers play this role, while in music sharing systems such alternatives are not available. As a result, most Squid implementations define the number of hops to be 1, i.e., only the immediate neighbors are searched before the request is sent to the web server, whereas Gnutella allows up to 7 hops, since there is no alternative redirection point.

The process of selecting the neighbors to forward a request can take various forms, from the simple $query - all$ approach to random, or history based selection. The choice of strategy depends on the general system goals. For instance, if bandwidth consumption is of lesser importance (due to the availability of a high capacity line) the $query - all$ approach may be preferable.

3.3. Exploration

Whereas search concerns the retrieval of actual content, the goal of exploration is to identify beneficial nodes that may become neighbors. As an example, consider the web caching environment where, as mentioned before, a proxy sends directly to the web server a request that cannot be served by the first degree neighbors. Unless the proxy explicitly initiates an exploration process, it cannot obtain information about the contents of distant nodes. Another type of exploration is performed in Gnutella by nodes that seek new neighbors (possibly because a previous neighbor

logged-off). In particular, such nodes issue a dummy query (called $ping$) and join the neighborhood of other nodes (with free neighbor slots) that respond.

```
On Exploration Triggering Event
        Select set of data items to query for
        Select one or more outgoing neighbors and
                                forward the exploration query
        Obtain results and update statistics
On Exploration Query Arrival
        Return statistics and summarized information
        If propagation terminating condition is not met
                Select one or more outgoing neighbors and
                                forward the exploration query
```

Algo. 2. Exploration Algorithm

Algo.2 illustrates a general pseudo-code for exploration. The process involves querying (w/o fetching) about collections of data. The neighbors that receive the query propagate it until a terminating condition is reached. Having obtained the results, the initiating node updates the statistics according to which neighbor selection is performed.

Exploration is triggered when certain events occur. The choice of events is very important since it significantly affects performance. Ideally, there should be a correlation between the exploration frequency and the frequency with which repositories change their contents.

3.4. Neighbor Updates

Search and exploration mechanisms do not depend on the kind of neighboring relations exhibited. Neighbor updates, though, are different for the asymmetric and symmetric cases. The update algorithm, in its general form, is based on the collection of statistics and the computation of a benefit function for selecting new neighbors. In practice, this requires maintaining information for both the neighboring and the non-neighboring nodes that were encountered through search and exploration.

The benefit function should capture the general goals and characteristics of the system. In distributed web proxy caching for instance, the number of retrieved pages, combined with the end-to-end latency, is a good candidate for benefit, since page size plays little role. On the other hand, in a multimedia sharing system, the file sizes must be considered, whereas in PeerOlap the dominating cost is the query processing time. Clearly, the statistics depend on the specific choice of the benefit function. Selecting appropriate events to trigger neighbor updates is also very important, since too frequent updates may be counterintuitive in some environments, while in others they can be a prerequisite. The events that trigger exploration are also applicable here. Another possible event candidate is to update whenever the statistics indicate that a non-neighboring node is more beneficial than at least one of the current neighbors.

```
On Update Triggering Event
        Sort current neighbors and nodes encountered by
                exploration according to a benefit function
        Select the most beneficial nodes and make them the
                new outgoing neighbors
                If the outgoing neighbor list is full
                        evict some existing neighbors
        Update statistics
```

Algo. 3. Neighbor Update (asymmetric case)

Algo.3 describes a generic algorithm for (pure) asymmetric relations. The update process in this case is simple since once an update event has been triggered, the node just adds the node(s) with the highest benefit in its outgoing neighbor list l_i, possibly evicting the least beneficial of the existing neighbors.

The symmetric case, shown in Algo.4, differs from the asymmetric one in the sense that neighbor reconfiguration involves invitation and eviction messages (processing an eviction message is omitted from the pseudo-code since it is straightforward). Accepting a node as an incoming neighbor involves the evaluation of a benefit function, unless the incoming list is not full.

```
On Update Triggering Event
        Sort current neighbors and nodes encountered by
                exploration according to a benefit function
        Do
                Select next most beneficial node
                If node was not in the previous outgoing list
                        Send invitation message
                        If positive reply
                                Add node in new outgoing list
                                If the outgoing neighbor list is full
                                        evict some existing neighbors
        While outgoing list not full && not all nodes considered
        Update statistics
On Neighboring Invitation Arrival
        If incoming neighbor list not full
                Send positive reply to inviting node
        Else If inviting node more beneficial than at least one
                                        neighbor in the incoming list
                Send positive reply to inviting node
                Notify evicted node
        Else Send negative reply to inviting node
```

Algo. 4. Neighbor Update (symmetric case)

We distinguish two cases: i) a node v_i that receives an invitation from v_j always accepts it, possibly by evicting the least beneficial neighbor v'_j, according to the benefit function used for outgoing neighbors; ii) the invited node v_i decides based on the potential benefit of the inviting node. Assessing the potential benefit of v_j, however, is not straightforward since v_i may have no statistics about v_j. Possible solutions to this situation are: a) the establishment of a temporary relationship in order to start exchanging search and exploration messages and gather statistics; the relationship will either become permanent or will terminate after a certain time threshold. b) the exchange of summarized information, according to which v_i can assess the potential

benefit. Notice, that we did not include this case in the algorithms, since such information is not always available.

A neighborhood update in the symmetric case may trigger other updates throughout the network. In the previous example, if v_i accepts the invitation of v_j, it will evict its least beneficial neighbor v'_j, which in turn may send an invitation to another node and so on. In general, the frequency of reorganizations should maximize the overall benefit of sharing, while at the same time avoid overloading the network with exploration and invitation/eviction messages.

4. Case Study: Adaptive Content-sharing Network

In this section, we consider the case of music sharing among end-users of Gnutella. Gnutella defines that when a node logs in, it first contacts a specialized server and retrieves a number of addresses of other nodes that are currently online. The neighborhood list is then selected from these nodes (typically each peer has 4 neighbors), and remains static until a neighbor logs off the system. Both the initial configuration and the changes are purely random, and do not take into account the music preferences of individual users. We employ our framework to allow each node to dynamically reconfigure its neighbors in order to minimize the number of search hops of future requests.

4.1. Problem Parameters

Below we discuss various parameters of the general framework for the dynamic variation of Gnutella:

Symmetric neighborhood relationships - The symmetric relationship is imposed by the fact that each user tries independently to maximize his/her own potential for locating interesting files. Asymmetric relations cannot achieve such a balance; e.g., it is possible that a node with numerous songs will be the outgoing neighbor of many other nodes (that consume its resources), while it does not get any benefit from sharing with them.

No directory information - Nodes have no information about the contents of their neighbors' libraries. Although it would be possible to develop some form of summarized information (e.g., similar to routing indices), it would involve major modifications in the Gnutella protocol.

Forced reconfiguration - Typically, Gnutella nodes remain active for a limited time period. This suggests that neighbor slots become continuously available and the update process must take this into account.

Infrequent Reconfiguration - Users' preferences for songs remain rather static. This suggests infrequent reconfiguration once the first "beneficial" neighbors are found.

Combined search and exploration process - The absence of a central repository and directory information enforces an extensive search process and there is no need for a separate exploration step. If a neighbor contains the query results, it replies to the initiator without further propagating the query (in order to limit the number of messages). All propagations terminate after 5 hops.

Different importance of results - All search results are not equally beneficial. A user will prefer to download a song from a node with high bandwidth. Moreover, the search process accumulates multiple results and presents them at the initiating peer. The larger the results list, the lesser its significance for the reconfiguration process.

Judging from the above observations, we implement neighbor update as follows. i) Each obtained result accounts for a benefit of c/TRN, where c is the bandwidth of the answering link and TRN is the total number of results. Notice that the Gnutella Ping-Pong protocol, which performs exploration, specifies that information concerning bandwidth capacity is propagated together with the query reply. ii) Periodically, each node checks the cumulative benefit of all nodes for which it keeps statistics, and includes in the new neighborhood the most beneficial ones. iii) When a new node needs to be added, an invitation message is sent. iv) The invited node always accepts an invitation evicting the least beneficial neighbor if necessary. v) Neighbor log-offs trigger the update process.

Algo.5 illustrates the pseudo-code that describes the basic functionality of the system. Function *Send_Query* corresponds to the combined search and exploration process as discussed in Section 3. In particular, the initiator node v, sends the query to its neighbors and waits for the results until a time-out period is reached. The statistics from nodes that respond are updated accordingly.

Function *Process_Query* illustrates the actions taken by a node v_i that receives the query. First, v_i checks whether it has received the same query before (through another path), in which case it discards it. In order to achieve this, each node keeps a list of recent messages. If this is the first time that v_i receives the query, then (i) if v_i contains the result, it returns it to v and does not propagate the query; otherwise (ii) if v_i does not contain the result, it propagates the query to all its neighbors, provided that the limit of hops has not been reached. When reconfiguration occurs, the list l_{new} of most beneficial nodes is computed according to the statistics. Invitation messages are sent to the ones that do not belong to the current list of neighbors. Eviction messages are sent to the neighbors that are not in l_{new}, and the reconfiguration counter is reset.

When a node v_i receives an invitation it always accepts it. If v_i has some empty slot it accommodates the inviting node; otherwise, it evicts one of its current neighbors. After the invitation is processed, v_i resets its reconfiguration counter in order to avoid updating the neighborhood in the near future (which could trigger cascading updates).

For the same reason, when a node v_j is evicted (function *Process_Eviction*) it does not attempt to replace the evicting neighbor (v) immediately. Node v_j will obtain a new neighbor if: (i) it receives an invitation from another node or, (ii) reaches its reorganization threshold. Notice that v's statistical information is reset, so that v_j will not attempt to reconnect to v in the near future.

Algorithm **Send_Query**
INPUT:(*File*: F)
for each neighbor v_i
query (F,v, v_i)
nlist=collect-query-result(F,time-out)
update the statistics of each node in l'

Algorithm **Process_Query**
INPUT:(*File*: F,Node:v (sending node), Node: v_i(current node))
if the same message has been received before return
if F stored locally **then** reply to v and return
if limit of hops has been reached **then** return
for each neighbor v_j of v_i
query (F, v_i, v_j)

Algorithm **Reconfigure**
INPUT:(*Node*: v)
l_{old}= list of (k) nodes in the old neighborhood
l_{new} = list of (k) most beneficial nodes
for each proxy v_{old} in l_{old} but not in l_{new}
send *eviction*(v, v_{old})
reset reconfiguration counter(v)

Algorithm **Process_Invitation**
INPUT:(Node: v (originator), Node: v_i (current-invited-node))
If (empty neighbor slot exists) accept invitation
Else
evict least beneficial neighbor v_j according to statistics
send *eviction*(v_i, v_j)
reset reconfiguration counter(v_i)

Algorithm **Process_Eviction**
INPUT:(Node: v (originator), Node: v_j (current-evicted-node))
reset v's statistics

Algo. 5. Basic functionality of dynamic Gnutella

4.2. Simulation Settings

In the absence of any real data about the music libraries of individual[3] users, we created a synthetic dataset, which attempts to capture the profile of the average user . We assume that the search space consists of 200,000 distinct files (songs). These songs are equally divided into k = 50 categories, which represent different music genders (e.g. pop, jazz, rock, etc.). The popularity of the songs within each category follows the Zipf's law with parameter θ = 0.9; the most popular songs from each category will be shared by many users, while the least popular ones will exist in the libraries of only a few users.

The network consists of 2,000 users, and each user maintains a number of songs that follows a Gaussian distribution with mean 200 and standard deviation 50 (i.e. there is approximately a total of 400,000 songs in the whole network). Each user has a favorite category (e.g., rock), and 50% of his songs belong to this category. The other 50% of the songs are selected from 5 other random categories (with a 10% contribution from each category). The selection of the individual songs is based on the popularity of the song inside its category (some popular songs are requested by most fans in the corresponding categories - the majority of the songs are requested by very few). The assignment of users into categories is also performed according to Zipf's law with parameter θ = 0.9.

Each user will stay on-line for a period of time, which is exponentially distributed with mean 3 hours, and then go off-line for a period of time, which is also exponentially distributed with the same mean. Therefore, there will be on average 1,000 users simultaneously on-line. When on-line, each user will issue queries with the same frequency. The category in which a query falls, matches the distribution of the user's preferences (i.e. with 50% probability the user will ask for a song from his favorite category). We set the number of songs that are requested by a query to one.

For more realistic simulations, we randomly split the users into 3 categories, according to their connection bandwidth; each user is equally likely to be connected through a 56K modem, a cable modem or a LAN. The mean value of the one-way delay between two users is governed by the slowest user, and is equal to 300ms, 150ms and 70ms, respectively. The standard deviation is set to 20ms for all cases, and values are restricted in the interval $\mu \pm 3\sigma$.

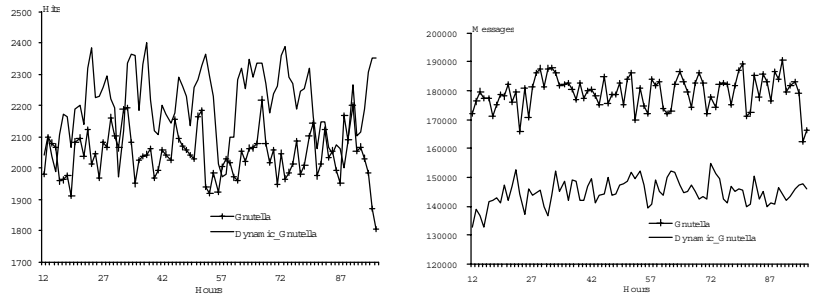
4.3. Experimental Results

Next, we describe a set of experiments which illustrate the performance gain of the dynamic version of Gnutella under various settings. In all the experiments, the maximum number of neighbors was set to 4. Unless otherwise specified, the reconfiguration threshold was set to 2 requests.

Figure 1(a) shows the total number of queries that were satisfied during each one-hour interval for a simulated period of 4 days. We present the results after the 12^{th} hour, when the system has reached its steady-state. The maximum number of hops (terminating condition) is set to 2. The dynamic approach clearly outperforms the static configuration, and it is able to satisfy more queries. This is due to the fact that, as the time evolves, new beneficial neighbors are being discovered. These nodes can not be reached in the static configuration, since the maximum number of hops that a query may traverse is limited.

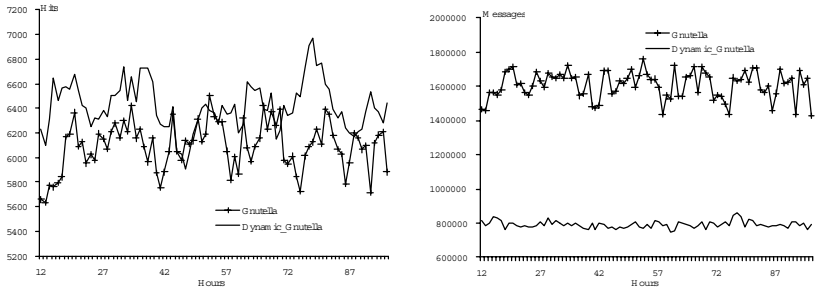
Figure 1(b) illustrates the corresponding overhead for obtaining the above results. It shows the number of messages (i.e., queries) propagated in the network per hour. The overhead of the static scheme is large because most of the

[3]Note that the work of [9] does not apply to individual users, since it models the aggregated statistics at the server side.



(a) Number of queries satisfied . (b) Query overhead.

Figure 1. The performance of the dynamic Gnutella (hops=2)



(a) Number of queries satisfied . (b) Query overhead.

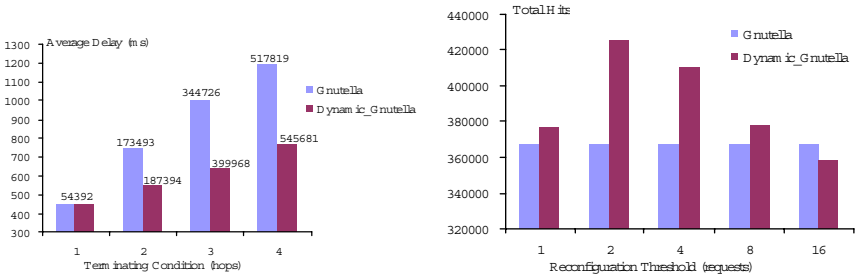
Figure 2. The performance of the dynamic Gnutella (hops=4)

nodes propagate the query further, since they are not able to satisfy it. The dynamic approach, on the other hand, groups nodes with similar content together. Thus, more queries are satisfied in the first hop and the request is not propagated. The performance gain, though, is limited since only up to 4·3 nodes are explored during each query.

The performance difference is significant if we allow the queries to propagate for a larger number of hops. Figures 2(a) and 2(b) present the total number of hits and the overhead exhibited by the systems, respectively, when the terminating condition is set to 4 hops. Each query can now reach up to $4 \cdot 3^3 = 108$ nodes; therefore, a large number of beneficial neighbors is more likely to be discovered. This fact explains the sharp improvement in terms of query overhead. In brief, the dynamic approach is able to produce more hits compared to the static configuration, while at the same time it reduces the message overhead by 50%.

The effect of neighborhood reconfiguration is illustrated in Figure 3(a). This figure shows the average delay observed from the moment a query is issued at a certain node, until the first result arrives at that node. The numbers above each column indicate the total number of results obtained. In the static approach, the delay increases significantly when searching is more extensive. This fact implies that most of the results are retrieved from nodes that are far (i.e., hops) from the originating node. Notice, however, that the total number of results also increases significantly. In the dynamic scheme, though, most of the results come from nearby nodes, and extensive searching is not necessary. Therefore, dynamic neighborhood reconfiguration can provide more results compared to the static approach and, more important, it can do that with a considerably lower delay.

In Figure 3(b), we evaluate the impact of the reconfiguration threshold r on the performance of the system. When r = 1, the total number of hits (i.e., for the whole 4-day period) achieved by the dynamic system is similar to the static one. This is due to the fact that any node that returns a result will potentially become a neighbor, even if the two users do not share the same interests. Therefore, the selected set of neighbors may not be beneficial in the long term. Neverthe-



(a) Average response time for first result.

(b) Effect of reconfiguration period.

Figure 3. Neighborhood reconfiguration evaluation.

less, if we allow time for enough statistics to be collected, the system can reach to a more advantageous state, increasing considerably the number of hits, compared to the static configuration. Observe that if the value of r is too large, the system does not have the chance to perform enough reconfigurations during the 3-hour period (on average) that a user is on-line. Since only one neighbor is exchanged during each reconfiguration, the entire set of beneficial neighbors may not be identified. Therefore, the performance drops again, converging asymptotically to the static case.

We should note, that the optimal value of r, indicated in Figure 3(b), is based on the specific settings that were used in our experiments. We expect that the optimal value of r will become larger if we allow each user to stay on-line for longer periods of time, or if we increase the frequency of the requests for each user.

5. Conclusion

In this paper we proposed a unified framework to characterize searching in distributed data repositories without centralized indexes. We identified three distinct modules: search, exploration and neighbor update, and provided general algorithms which capture the functionality of diverse systems. We applied our framework for many existing systems, including content-sharing networks, distributed caching and P2P DBMSs. Our study revealed that such systems can exhibit significant performance gain by incorporating dynamic network reconfiguration, in order to adapt varying query patterns. We illustrated this idea by a case study of an adaptive Gnutella-like content-sharing system.

Acknowledgements

The authors would like to thank Dimitris Papadias for his contribution in various parts of this paper. Spiridon Bakiras is supported in part by the Areas of Excellence Scheme established under the University Grants Committee of the Hong Kong Special Administrative Region, China (Project No. AoE/E-01/99). Wee Siong Ng is partially supported by the NSTB/MOE research grant RP960668.

References

[1] National Lab of Applied Network Research, IRCache project. Sanitized access logs, available at: http://www.ircache.net/.

[2] C. Aggarwal, F. Al-Garawi, and P. Yu. Intelligent crawling on the world wide web with arbitrary predicates. In *10th Int. WWW Conf*, 2001.

[3] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K. L. Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *ACM SIGMOD*, pages 25–36, Madison , Wisconsin, USA, 2002.

[4] D. Kossmann. The state of the art in distributed query processing. In *ACM Computing Surveys 32(4)*, pages 422–469, 2000.

[5] S. Ratnasamy, R. Francis, M. Handley, R. Krap, J. Padye, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, 2001.

[6] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *18th IFIP/ACM ICDSP*, 2001.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[8] D. Wessels. Squid internet object cache. Available at: http://www.squid-cache.org/.

[9] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *VLDB*, 2001.

[10] B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *ICDCS*, 2002.

[11] L. Zhang, S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, and V. Jacobson. Adaptive web caching: Towards a new global caching architecture. In *3rd Int. Web Caching Workshop*, 1998.