Managing Collaborative Feedback Information for Distributed Retrieval*

Pascal Felber‡ Toan Luu§ Martin Rajman§ Étienne Rivière‡†
‡ Université de Neuchâtel, CH-2009 Neuchâtel, Switzerland
§ Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland
{pascal.felber, etienne.riviere}@unine.ch, {vinhtoan.luu, martin.rajman}@epfl.ch

ABSTRACT

Despite the many research efforts invested recently in peer-to-peer search engines, none of the proposed system has reached the level of quality and efficiency of their centralized counterpart. One of the main reasons for this inferior performance is the difficulty to attract a critical mass of users that would make the peer-to-peer system truly competitive. We argue that decentralized search mechanisms should not aim at replacing existing engines, but instead complement them by adding novel functionalities that would be difficult to provide in a centralized manner. This paper introduces an example of such a complementary search mechanism and presents the design of a distributed collaborative system for leveraging user feedback and document/user profiling information.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Relevance feedback

General Terms

Algorithms, Design

1. INTRODUCTION

Efficient search mechanisms are unanimously recognized as crucial components of massively distributed information systems such as the Web or file-sharing networks. In this context, the last few years have seen the emergence of an intense research activity in the area of peer-to-peer (P2P) based search, with the objective of replacing centralized search services such as Google, Yahoo!, or Live Search by large-scale P2P search engines.

However, despite the many research efforts invested so far, none of the currently released P2P Web search engines (e.g., Faroo or YaCy[1]) has been able to reach a level of quality and efficiency that would allow them to truly compete with their centralized counterparts.

We believe that the lack of widespread adoption of P2P Web search engines is a direct consequence of the so-called "bootstrap problem" [7]: the added value of a distributed search engine becomes only perceivable when the system has attracted enough users to fully sustain its specific functionalities. As long as such a critical mass of users is not reached, the distributed system's performance is noticeably inferior to that of the centralized engine, which in turn prevents attracting the necessary number of users. Given the nature and objectives of search engines, the major potential advantages of distributed approaches over centralized ones (better scalability, better privacy enforcement, better resistance to censorship) cannot make up for the subpar quality of the search results.

As a consequence, we concur with the authors of [7] that the most promising approach for distributed search engines is not to launch them as autonomous systems aiming at replacing their centralized counterparts, but rather to couple a distributed engine with one (or several) centralized engine(s), as a companion system providing additional functionalities that are impossible—or difficult—to implement with a centralized approach. Such a companion system can later run autonomously when the users recognize its performance and functionalities as good enough.

In this position paper, we present an example of what such a coupled set up might be, in the specific case where the targeted added value functionality is the collaborative exploitation of the feedback information provided by topical communities of users. This information is used to construct a search mechanism that leverages (1) community-based information, reducing the risks of manipulation and fragility that are potentially faced by centralized search engines, and (2) user-specific information that helps targeting the results to some specific user's interests. This latter objective is further coupled with the goal of maximizing the so-called *info-diversity*, that is, to allow specifically tailored results for the largest possible set of users with distinct interest profiles.

Our system has similarities with meta-search engines [14], which send queries to several search engines and allow users to search more of the information system. Nonetheless, these systems compile results using merging and ranking functions that, without knowledge of the ranking methods internally used by the search engines, are known to produce dimly rel-

[*] The work presented in this paper is carried out in the framework of a collaboration between EPFL and UNINE funded by the EPFL Center for Global Computing.

[†] This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme.

[1] `http://www.faroo.com`, `http://YaCy.net`

LSDS-IR'08, October 30, 2008, Napa Valley, California, USA.

evant and fairly polluted results. Our approach is not to combine results from search mechanisms based on similar metrics (data popularity, structural information, etc.) but to present on one side the results obtained by such metrics and, on the other side, the results obtained from the collaborative construction of user-based relevance metrics and document/user interest profiling.

While we do not specifically focus in this paper on other aspects of distributed search systems, we expect them to provide perceivable benefits once the system has reached a critical mass of users. Examples of such features are scalability and privacy enforcement. Scalability comes from the aggregation of resources: the more users participate, the more resources are available for providing the service, i.e., the architecture scales gracefully with this number of resources. Privacy is another important concern for users, as a considerable portion of them care that search engines collect certain non-personally identifiable data about their queries.[2] All search engines (e.g., Yahoo, Google or Live Search) can use search data of their users to keep track of their interests and online activities. We propose a fully distributed architecture with an appropriate privacy-preserving design, which we consider as a sound approach for avoiding tracking and censorship.

The rest of the paper is organized as follows. Section 2 presents the targeted system and its architecture, and discusses its key components. Section 3 highlights the most important research challenges raised by the design and implementation of this architecture. Related work is discussed in section 4. Section 5 concludes our paper.

2. THE CFRS SYSTEM

This section presents our proposed Collaborative Feedback based Retrieval System (CFRS). First, we present a view of the service as it is presented to the user. Second, we highlight the components required to implement the service. We then describe the architecture of the system and discuss its main components.

2.1 The User Perspective

After being downloaded and installed by the user, the peer client software automatically connects to the CFRS. Whenever a user submits a query, this query is sent transparently to both a centralized search engine and the CFRS. The user obtains two results list: one from the search engine, one from the CFRS. In the result list from the CFRS, documents that match best the user's interest domains are ranked higher.

A typical example is when a user submits the query "Java". The first results returned by the *Google* search engine are for the programming language, the island in Indonesia, and a band named Java. A person, who has issued requests for tourism-related content in the past, and never about computer programming, would probably prefer to get the island-related content first. Such a user-centric ranking is not feasible with only structural information (links between content, content popularity, etc.). Second, the same request "Java" made by a programmer returns a set of resources that have high structural rank (e.g. *Pagerank*) but that are not necessarily the ones that would be useful first (the *Java API documentation* page in this case). Using information about pages ranked higher (i.e., visited more often) by users for this query would help the system propose more accurate results.

Perceivably, results from the CFRS rise in quality when: (1) the user submits more queries (as the system is able to determine his interest profile) and (2) all users submit more queries and relevance tracking information (as they indicate which data is deemed interesting by users as a whole). After the user selects and accesses an element in any of the two lists, information about this access is sent to the CFRS, which will in turn use it for improving the quality of replies to the same query made by other users.

2.2 Architecture

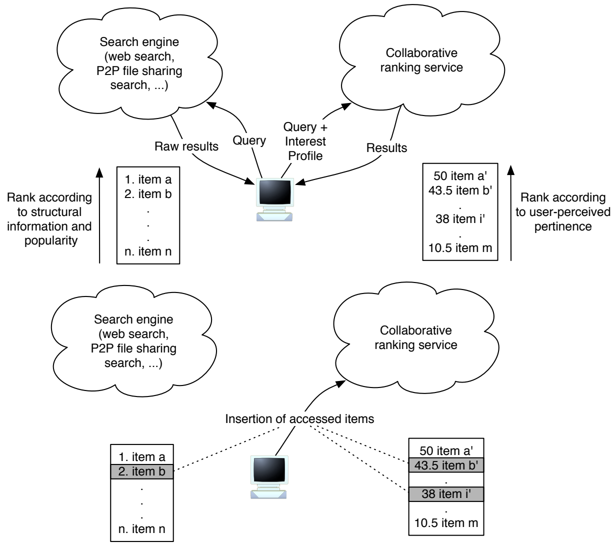

Search engine (web search, P2P file sharing search, ...)
Collaborative ranking service
Raw results
Query
Query + Interest Profile
Results
Rank according to structural information and popularity
1. item a
2. item b
.
.
.
n. item n
50 item a'
43.5 item b'
.
38 item i'
.
10.5 item m
Rank according to user-perceived pertinence
Search engine (web search, P2P file sharing search, ...)
Collaborative ranking service
Insertion of accessed items
1. item a
2. item b
.
.
.
n. item n
50 item a'
43.5 item b'
.
38 item i'
.
10.5 item m

Figure 1: Functional view of the system: query & relevance tracking information submission.

Figure 1 presents a *functional* view of the system. Queries to some existing search engine and to the CFRS are sent in parallel; results are then ranked by structural information or user-based relevance information. Queries to the CFRS are accompanied by the *interest profile* of the user, which is used to customize the result list according to his interests. Relevance information is gathered on the user's machine by observing accesses to elements returned by any of the search methods.

On the *user side* it is necessary (1) to build appropriate interest profiles for improving search results relevance (*user interest profiling*); (2) to present the results returned by the CFRS (*user interface*); and (3) to provide a mechanism for inserting new relevance information, based on the user's local accesses to elements from the results lists (*relevance feedback tracker*).

On the *collaborative retrieval service side*, mechanisms are needed (1) to direct requests and relevance tracking information to the relevant node(s) in the network (*routing*) and (2) to compute lists of relevant items for queries, based on relevance tracking information and user interest profiles (*storage and ranking*, denoted by *storage layer* in the paper).

The architecture of the system and the main interactions between its elements are depicted in Figure 2. The next sections describe its major components (shown in grey in the figure).

2.3 Main System Components

This section presents the design and analysis of the key components of the system: the user interest profiler, the

[2] `http://mashable.com/2007/12/11/poll-search-privacy`

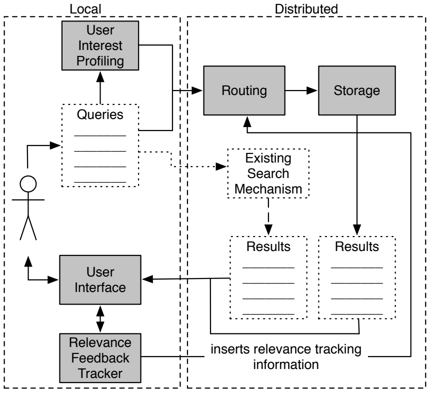

Local
Distributed
User Interest Profiling
Routing
Storage
Queries
Existing Search Mechanism
Results
Results
User Interface
Relevance Feedback Tracker
inserts relevance tracking information

Figure 2: System architecture.

relevance feedback tracker, routing and storage layers and the user interface.

2.3.1 User Interest Profiling

Users are reluctant to do extra work when searching distributed content and usually submit short queries. This can lead to ambiguous or inadequate results. Therefore, automatically exploiting user-specific information to personalize retrieval result ranking has received considerable attention [9]. Several data sources can be considered for personalizing retrieval, notably the history of queries and accessed documents, and search independent user-centric information (shared documents, bookmarked items, etc.). Some approaches [23, 24] obtained noticeable improvements by employing automatically created models to re-rank search results with user profiles. Using the overlap of shared documents as a measure of proximity for interest profiles has been successfully used in the context of search mechanisms for P2P file sharing systems [8].

In our approach, we rely on *user profiles*, which correspond to sets of representative keywords, extracted from visited Web pages and user's query history, and are used by the CFRS system to better estimate what the user is actually searching for.

In addition, as the profiles are transmitted with the queries, they need to be *small* and *encoded* (in order to preserve privacy). For this purpose, in the CFRS system, the profiles are represented in the form of space-efficient probabilistic data structures, the *Bloom filters* [4].

As described in [5], the very simple structure of the Bloom filters makes several useful operations straightforward to implement. A Bloom filter representing the union of two sets (S_1 and S_2) is obtained by the logical OR of their bit vectors. Bloom filters are also used to approximate the size of the intersection between two sets S_1 and S_2. It follows that Jaccard ($\frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|}$) or Dice ($\frac{2|S_1 \bigcap S_2|}{|S_1|+|S_2|}$) similarities between 2 sets are easily estimated with Bloom filters. These similarities allow us to *rank* the documents in the result list of a query, based on a profile sent with the query.

Profile maintenance. We assume that each user has several topical interests. Each topic is represented by a different profile, built and maintained locally at the user's side. The first time a user accesses the system, his profile set is empty. However, to use the system efficiently from the start, an initial user profile could be built from his bookmarked documents or some local specific document collection.

When the user selects a document in the result list obtained for a query Q, keywords are extracted, either from the document itself, or from the document summary (title, snippet) appearing in the result list. These keywords are weighted and the most representative ones are selected. The terms of the query are also added into the set if they are not already present in the set of top keywords. A Bloom filter DP is then generated from the selected keyword set and inserted in the local profile storage. The following insertion mechanism is considered: if the maximal size of the local profile set is not reached, the new document profile DP is simply inserted as a local user profile UP. Otherwise, the new profile is first temporarily inserted, and we compute the maximal similarity $s_{max} = max\{sim(UP_{i1}, UP_{i2})\}$, where the UP_i are the profiles in the local storage. The pairs of profiles with maximal similarity are merged into a single one of the form $UP = UP_{i1} \bigcup UP_{i2}$.

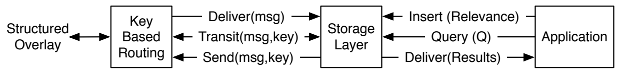
Profile usage. Profiles are used when submitting queries to the CFRS. At query submission, the system finds in the set of local profiles the one that has the highest similarity with the query, and sends it along with the query to the storage layer. Section 2.3.4 gives more detail on how queries and profiles are processed by the storage layer. The relevance feedback tracker, to be described next, also uses interest profiles. When the user accesses a document from the result lists (returned by centralized search engine or distributed collaborative service), a profile is build from this document and is sent together with the relevance feedback information.

2.3.2 Relevance Feedback Tracker

When a user browses the result list for a query, the title, document reference and snippet help him to determine relevant document w.r.t his query and interests. Accessed documents and user interest profiles are thus the important feedback information that is exploited by the CFRS. When the user selects one element in the results list for a query Q, the following information is tracked: (1) the query Q is used to route the feedback information to $P(Q)$, the peer responsible for processing it; (2) the document reference D — this is the identifier of the document (its nature depends on the centralized search mechanism being used, e.g., an URL for Web searching, a file hash for P2P file sharing, etc.) and the document descriptors (title, snippet); (3) the document profile DP which is extracted from the content or description of D, in a form of a Bloom filter. Meanwhile, the profile of accessed documents is inserted into the local interest profiles as described in the previous section.

2.3.3 Routing Layer

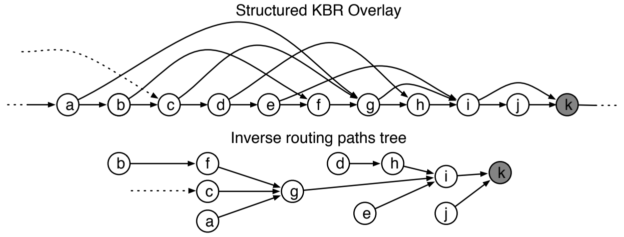
Each query Q is associated with some node $P(Q)$. This node stores all information pertaining to Q: a set of documents references, the associated relevance tracking and interest profiling information. Our design is based on the standard API for structured P2P system proposed by [6]. This design is a specialized form of a distributed hash table (DHT), which associates a key-based routing layer (KBR) and a storage layer. The role of the KBR layer is to locate the node responsible for some query's key. To that end, it relies on a structured overlay (e.g., an augmented ring), where each node is assigned a unique identifier and the responsibility of a range of data items identifiers. In our case, each query Q has an identifier determined by hashing its terms to a key $h(Q)$. The node $P(Q)$ whose range covers $h(Q)$ is responsible for maintaining information related to Q and provide the appropriate sorted set of document references when asked to by some distant node. The main APIs are:



The application issues a request or inserts relevance tracking information for some query Q to the local instance of the storage layer, which in turn uses the KBR `Send` call for reaching $P(Q)$. On each routing step towards the destination, the storage layer can be notified by the `Transit` call that a message is transiting *via* the local node, towards $P(Q)$. It can in turn modify the content of this message, or even answer the request on behalf of $P(Q)$. This latter mechanism is used in our design to implement load balancing and fault tolerance (as described in Section 3.2), by exploiting the *routing paths convergence* property of the underlying structured overlay.

Informally, the property of *path convergence* results from the greedy routing algorithm used by many KBRs. Our system will be based on the routing layer of Pastry [17], for its stability and its performance (number of hops, usage of network distance for choosing peers, etc.). In Pastry, nodes and items have identifiers of d digits (each digit is a number in base b, with $b = 4$ in the common case) and are organized on an augmented ring. Each node in the ring constructs a routing table that contains references to a set of other peers (chosen according to the prefixes of their identifiers). When routing a request to its destinations, each intermediary node will select as next hop a peer from its routing table with an identifier that has a longer common prefix with the target key than itself. As each routing step "resolves" at least one digit, at most $d = O(\log N)$ routing steps are required (N is the number of nodes in the network).

An interesting property of such a greedy routing strategy is that routing paths towards a destination converge to the same set of peers, and does so with a higher probability as digits are resolved: the more digits have been resolved, the less peers remain that have a longer common prefix with the target key. Routes from all nodes to some key in the network collide in the last hops, as illustrated below. The path convergence property is particularly useful for the design of load balancing and fault tolerance mechanisms [16, 20].



A "standard" DHT provides a *raw* put/get interface to the application [6]. Elements are stored as *blocks* on the node responsible for their key, and retrieved as blocks as well. Our design differs in the important following point: our storage layer does not store information *blindly*, but provides an interface and functionalities that are *specific to the storage and processing of ranking information*. This has a strong impact in particular on the design of load balancing and fault-tolerance mechanisms that would not be conceivable with a standard DHT.

2.3.4 Storage Layer

The storage layer at the peer $P(Q)$ responsible for the query Q is in charge of (1) the management of the relevance feedback information received by $P(Q)$ for Q, and (2) for the generation of the results to be sent back to a user submitting the query Q to the system.

Concepts. The main problem for the design of an efficient storage layer is the potential heavy skewness of the relevance feedback generation process (see also 3.2).

Under such conditions, it is therefore unrealistic to keep all the available relevance feedback information (as peers with highly used queries would quickly run out of main memory storage). Some sort of storage control mechanism must be implemented. Relying for this on some document-query relevance measure for evaluating the quality of relevance feedback items containing the considered document-query pair is potentially problematic. Indeed, (1) the most efficient document relevance scoring techniques (e.g. the link-based PageRank used by Google) are notably costly in a distributed set up; and (2) document relevance seems, by nature, not well adapted to the type of information processed by our system. Because they are all directly generated by users and all represent some form of user interest, the processed relevance feedback items should rather be considered of quite comparable intrinsic quality. Thus, discriminating between them needs to rely on other criteria, in our case, *popularity* and *semantic coverage.*

Popularity. A relevance feedback information associating a document D to a query Q is *popular*, if it is frequently produced by many users, and thus results in a high arrival rate at the peer responsible for Q. More precisely, the popularity of a relevance feedback item relating D to Q is the rate $p(D)$ of the stochastic process modeling the arrival at $P(Q)$ of relevance feedback information for D. An important part of the storage management mechanism described hereafter is therefore dedicated to make the selection of the stored items adaptively sensitive to the $p(D)$ rates.

Semantic coverage. Another important aspect to consider when selecting interesting relevance feedback information is the amount of semantic content the associated document brings w.r.t. the other documents already present in the system through other relevance feedback items. The goal is then to maintain a set of relevance feedback items providing a satisfactory coverage of the topics expressed in the documents present in the relevance feedback items associated with a given query. More precisely, in line with what was already mentioned in section 2.3.1 about User interest profiling, we measure the "semantic redundancy" of a relevance feedback item by the maximal similarity of the associated document profile with the document profiles of the documents already stored in the system. A second important part of our storage management mechanism design is therefore dedicated to make the relevance feedback item selection sensitive to document profile similarities. In this perspective, the goal is to focus on relevance feedback items with maximal relative similarities that are minimal, so as to maintain a maximal semantic coverage in the set of stored relevance feedback items.

Finally, in order to achieve a globally satisfactory storage management mechanism, it is also crucial to provide some ways to arbitrate between the two considered criteria (popularity and coverage). In our approach, we rely on the assumption that semantic coverage is less crucial for currently popular relevance feedback items, but very important for items that used to be popular in the past but progressively came out of trend. The underlying idea is that, for current "hot topics" (i.e., topics expressed in documents associated with currently popular relevance feedback items), the users are expecting a much higher level of detail than for topics that used to be popular in the past, for which only the main involved issues might be considered. In a nutshell, for selecting relevance feedback items corresponding to current hot topics, popularity is crucial, while for selecting the items that keep track of past hot topics, semantic coverage is more adequate. In consequence, our storage management mechanism aims at *selecting* the currently most popular relevance feedback items without taking semantic coverage into account, but in parallel, aims at *preserving* the relevance feedback items that used to be popular in the past without taking into account their current popularity.

Implementation. This section describes the targeted implementation of our relevance feedback information management mechanism. For readability purposes, the expression "relevance feedback" is often abbreviated to RF, for example, relevance feedback items are often simply called RFitems.

As already mentioned earlier in the section 2.3.2, the RF information concerning a query Q received from the network by the peer responsible for Q consists of a triple (Q, D, DP), where D is the information available about the considered document (mainly the document URL uniquely identifying the document, and a document summary containing the title and the snippet as it is quite standard in Web search engines), and DP is the Bloom filter representing the document profile. When acquired from the network, an RF information is first transformed into an internal data structure, an RFitem, which stores the (Q, D, DP) triple (plus other attributes required for the processing, such as its occurrence frequency or its maximal semantic similarity) in an easily computable way.

The general goal of the RF information management mechanism is then to process all the triples received by $P(Q)$ in order to select the ones that should be stored at $P(Q)$ for retrieval purposes. From a more computational perspective, the essential aspect is that the storage management mechanism requires a strictly bounded storage space that is controlled by a "maximal storage space" parameter (denoted by S in the rest of this section). There is no reason why this parameter should be the same on all the peers. On the contrary, it is probably advisable to adapt it when possible, for example to the observed average RF information arrival rate at the peer. Having a strictly bounded storage consumption is clearly a necessary feature to realistically consider a deployment of the CFRS system on a true set of peers, but it also strongly impacts how the peer performs its RF information management: the bigger the value for S, the higher the quality of the storage management. More precisely, with a larger storage space, the peer will be able to store more popular RFitems, and to keep a larger set of past popular RFitems, thus improving its overall semantic coverage. The adaptability of the algorithm makes it able to cope with large ranges of possible storage space: for what ever provided value for S, it will try to perform in a way that fits best the true distribution of the RF information receive by the peer.

The fixed size storage space is split into two distinct parts: *the main store*, and *the archive*. The purpose of the main store is to progressively identify and store the popular RFitems received by the peer, while the archive serves to identify and store the past popular RFitems providing an acceptable semantic coverage. Figure 3 shows the main components of the storage layer and how they process the RFitems.

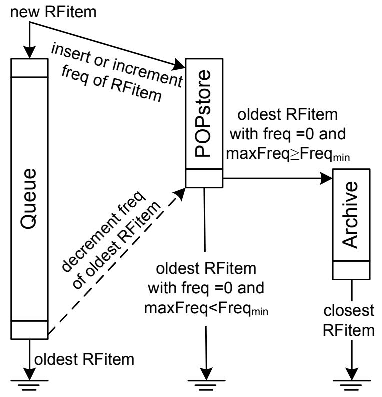

new RFitem
insert or increment freq of RFitem
Queue
POPstore
oldest RFitem with freq =0 and maxFreq≥Freq_min
Archive
decrement freq of oldest RFitem
oldest RFitem with freq =0 and maxFreq<Freq_min
closest RFitem
oldest RFitem

Figure 3: Main components of the storage layer.

The main store. The main store is decomposed into a variable size FIFO (First In, First Out) queue and a variable size associative memory (hereafter called the popular-unpopular store, and abbreviated to "the POPstore").

The queue is a temporal storage that serves as an "incubator" for the identification of popular RFitems. It contains the sequence of references to all the RFitems received by the peer during the time interval covered by the queue. RFitems themselves are stored in the *POPstore*, which is a map between URLs and RFItems. Both the queue and the POPstore are of variable size, which means that they grow or shrink independently (within the S bound imposed to the whole storage space) depending on the storage requirements imposed by the occurrence distribution of the received RFitems.

Each time a new *RFitem* r is acquired from the network, the POPstore is queried to check whether r is already present. In this case, its occurrence frequency is incremented and a reference to r is pushed into the queue. Otherwise, r is inserted into the POPstore with an occurrence frequency set to 1. If not enough storage space is available (either for the push or for the insertion), the oldest RFitem reference is poped from the queue, and the occurrence frequency of the corresponding $oldestRFitem$ in the POPstore is decremented.

If the updated occurrence frequency of the poped *oldestRFitem* r_{oldest} reaches zero (i.e., there is no more reference to it in the queue), r_{oldest} is removed from POPstore. In addition, if its maximal frequency (i.e., the maximal occurrence frequency observed for r_{oldest} during its lifetime in POPstore) is greater than a predefined popularity threshold $Freq_{min}$, r_{oldest} is inserted in the archive with the current value of its maximal frequency. Otherwise, r_{oldest} is simply abandoned.

If the processing of the poped *oldestRFitem* r_{oldest} did not free any storage space (i.e., its frequency it still non zero), the whole pop cycle is reiterated for the new currently oldest RFItem, until an RFitem with a frequency of zero is removed from the POPstore. The new acquired RFitem is eventually inserted in the POPstore and a its reference is pushed in the queue.

The archive. The archive is another variable size associative memory (indexed by the RFitem URLs) used to store past popular RFitems that have been rejected from the POPstore because their occurrence frequency in the queue has dropped to zero, but that were once popular (i.e., heir maximal occurrence frequency maxFreq is above the Freqmin popularity threshold) is above the $Freq_{min}$ popularity threshold). The purpose of this additional store is twofold. First, it is used to cope with the "burst" phenomena (i.e., a topic becoming suddenly extremely popular) that are frequently observed (e.g. on the Web), and can lead to the "flooding" of popularity based storage mechanisms with a limited number of very popular topics that expel all the other topics from the system. To some extent, the archive serves as a long term memory, while the *queue-POPstore* pair operates with a much shorter horizon (limited by the current time span covered by the queue). Second, the archive is the data structure that implements the semantic coverage based filtering mechanism.

As already mentioned, the semantic coverage based filtering mechanism relies on the notion of maximal document similarity that is derived from the document profiles associated with the stored RFitems. For this purpose, (1) each of the RFitems present in the archive is associated with a semantic redundancy score corresponding to the maximal similarity between its document profile and the documents profiles of all the popular RFitems in POPstore and all the RFitems in the archive; (2) the archive provides the possibility to efficiently access the most recent of the RFitems with the highest semantic redundancy it contains (hereafter called the closest RFitem and denoted by *closestRFitem*); and (3) each time a past popular RFitem needs to be inserted, if not enough storage is available for this operation, the current *closestRFitem* is first removed from the archive (and abandoned), and then the new past popular item is inserted.

As the computation of the required RTitem scores is a relatively costly procedure, it is not performed each time a change impacting the scores happens, but only when the ratio between the number of changes (insertions or removals) that happened in the POPstore and the archive since last score update w.r.t. the *POPstore+archive* size exceeds a predefined threshold γ, which thus allows to control the tradeoff between score accuracy and computational cost.

Finally, as the archive is an adaptive variable size data structure, it is important to provide some control on the storage space it consumes (w.r.t. the one used by the POPstore that competes for the same storage resource). For this purpose, our design provides a parameter α that imposes that at least a fraction α of the size of the POPstore is reserved for the archive.

Answering queries. When the peer $P(Q)$ receives the request (under the format (Q, UP)), the storage layer extracts from the popular RFitem in the POPstore and the past popular RFitem in the archive the list of of the k-most similar RFitem w.r.t the user profiles UP, and sends back the resulting list of documents descriptors (URL, title, snippet).

2.3.5 User Interface

Gathering information about the user interest and for relevance feedback tracking is based on some software pieces at the client side. There are several ways of implementing such a tool. These include, but are not limited to the following. First, one can use an extension to some browser (e.g., Firefox) that captures accesses to widely used search engines such as Google. This extension would be made available as a plugging. Second, open-source file-sharing software such as eMule can be used and their search mechanism instrumented easily. As systems such as eMule already propose a routing layer, this routing layer can directly be used by the CFRS system.

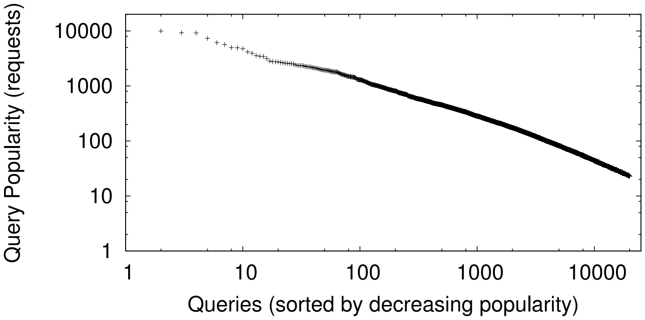


Figure 4: Wikipedia queries popularity distribution.

3. DESIGN CHALLENGES

This section discusses the most important research and system design challenges: on the distributed system side, performance, load balancing and fault tolerance; then challenges related to ranking and user profiling.

3.1 Latency and Throughput

The time required for querying the CFRS needs to be competitive with the time required to obtain results from the original search engine (e.g., within a second for Web search, a few seconds for file sharing search). This is a *low latency* requirement: query messages need to be treated with a high priority. On the other hand, insertions of relevance tracking information do not require this *soft real-time* quality of service. Such messages can be delayed for some time without implying a severe loss in the quality of the service as experienced by the user.

One should note that relevance-tracking messages from the peers sent through the KBR are of small size (the only space consuming element being the bloom filters used to encode the user interest profile). Sending each message separately is not cost-effective. Grouping low-priority messages can reduce the global overhead. This allows to use links more efficiently and to meet a *high throughput* objective. A similar priority-based mechanism has been successfully used in [10].

3.2 System Scalability and Load Balancing

The CFRS system is created for managing thousands of users simultaneously, and to store the information they create and access in a scalable manner. The underling KBR protocol based on Pastry [17] is particularly scalable: the state kept by each peer is $O(\log N)$ and routes are also $O(\log_{2^b} N)$. Lists maintained by the storage layer are of fixed size. The real problem one can face with such a storage layer is that of heavily sparse loads, which are common in distributed accesses patterns.

The skewness of the load. The most important challenge faced when designing a distributed search engine is the high unbalance of the load, which results from the skewness of users' interests. In most distributed systems, one can observe that the users interests follow a Zipf-like distribution. An example is given by Figure 4, which plots the distribution of query popularity on Wikipedia in September 2004. Only the 20,000 most frequent requests among 2,000,000 unique queries are plotted. One can notice that a few queries are extremely popular, while infrequent queries appear on the long tail of the distribution. This has an immediate impact on our design: the peers responsible for storing the most accessed queries will get an unbearable amount of load, which calls for some load balancing mechanisms.

Balancing by delegating. The basic idea of our load balancing mechanism is that of *delegation*. When some peer $P(Q)$ gets overloaded by requests to a popular query Q, it replicates its responsibility for managing information and answering requests related to Q. A wide range of techniques has been proposed for balancing load in structured overlays (e.g., [12, 16, 18, 20]). All these proposals however target scenarios where the number of accesses is much greater than the number of updates to the data. These systems support accesses to non-mutable data by placing replicas on nodes that lie on the path towards its key.

Our system requirements are different. First, the amount of writes (insertion of interest tracking information) and the amount of reads (queries) are of the same order. Caching only read accesses is thus not possible: routing every insertion for a query Q to the peer $P(Q)$ would involve notifying all copies, resulting in a load similar to the one avoided by caching access requests. It is thus necessary to also cache insertions, that is, to allow copies of information about a query to be modified *independently* to the "master" copy. We call such a copy a *delegate*: a replica onto which modifications are possible with only loose synchronization to its master copy.

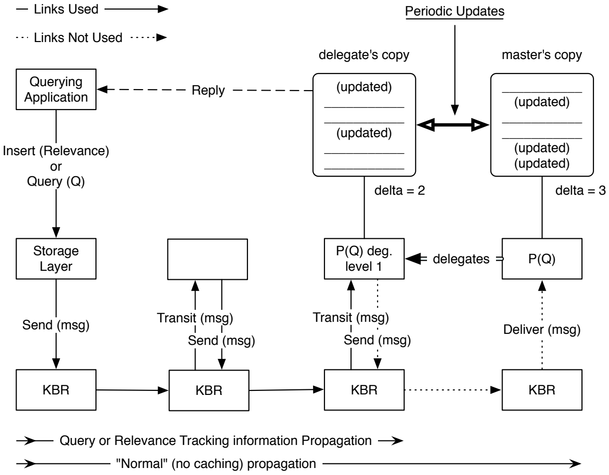


Figure 5: Delegation mechanism for load balancing.

Figure 5 presents the principle of delegations: a message from a peer (either a request or an insertion of relevance tracking information) is sent by the peer on the left side, and is routed towards the peer $P(Q)$, on the right side. As the next to last peer on the path is a delegate of $P(Q)$ for Q, it notices that a request for Q is going through its KBR layer and intercepts it. It replies on behalf of $P(Q)$ or inserts the information in its local copy. Delegates are chosen as follows: when $P(Q)$ is overloaded for a given query Q (the amount of received requests for Q reaches some threshold), it enters "pre-delegation" mode for Q and monitors incoming requests for Q. When enough such requests are known, the peer P_d that have transmitted the most requests for Q is chosen as a new delegate. This simple stateless model is preferred to a costly collection of statistics about all queries managed by the peer. It is based on the fact that routing paths towards $P(Q)$ form a tree, and incoming links of this tree forward in expectation an amount of queries that is roughly proportional to the number of incoming requesters on its sub-tree. Delegations are revoked by similar mechanisms: a delegate revokes if the amount of requests for the corresponding Q is below some threshold.

Delegates can in turn use the same mechanism for *re-delegating* Q as the master copy on $P(Q)$ and the delegates form a tree. Synchronization between the copies is performed periodically when the number of changes, denoted *delta* on Figure 5, reaches a configurable threshold. Peer-wise synchronization is used to aggregate the two copies (lists of documents and associated information) in a new list.[3] This list is then forwarded along the tree, resetting all *delta*s to 0.

3.3 Fault Tolerance

Constructing ranking information is a time-consuming process. As the system runs on a large-scale distributed system, it is likely to face a high level of churn. One needs to ensure that information about queries is not lost when a peer fails or leaves the network. We use a replication mechanism but count delegates as copies: if the replication factor is 3, and a query Q is already delegated twice, only one replica is needed. Replicas are loosely synchronized on a periodic basis. Nodes hosting replicas, as well as delegates, periodically *ping* the master peer. If the master copy of Q fails, the new peer responsible for the query receives the latest information pertaining to Q from the replicas. We use the *leaf set* for storing replicas as in Pastry [17] and PAST [18].

3.4 User Profiles

As previously mentioned, the user profile is built from the history of accessed documents. We plan to evaluate several recently proposed techniques to select representative keywords for a user profile [7, 23, 24]. When using profiles in the clustering algorithm, we have to deal with the problem of too general or too specific profile representations w.r.t to the user's interest topics. Important parameters include the time window that controls the evolution of user profiles, the number of profiles maintained, as well as the number of keywords in a profile. The settings of the Bloom filters (number of bits, number of hash functions, number of inserted elements) also need to be analyzed for finding good trade-offs between false positive ratio and profiles' accuracies.

Notice that in the feedback content, we do not take into account the relevance scores provided by the search engine because they are too heterogeneous and difficult to aggregate. We could, however, derive interesting information from the rank of document in the results lists, the order in which the users visits the documents, or the duration between successive accesses in the result list.

4. RELATED WORK

Many research efforts have been conducted for determining the feasibility of P2P Web search, mostly focusing on scalability and bandwidth consumption [3, 11, 13, 21, 22]. None of these, however, has addressed the bootstrapping problem or the added value of collaborative approaches. The authors of Chora [7] and Sixearch [1] use decentralized architectures for sharing and leveraging users' search experiences and addressing context limitations of centralized search engines. Note that using distributed techniques for enhancing the quality of service of existing centralized systems has been already considered sound in other contexts, such as for availability in [15]. In that paper, the authors define the concept of a "P2P-izer": a complementary P2P system working with some centralized system. We go one step further as the P2P

[3] Either by inserting "new" elements in the master list or by re-ranking the union of the two lists and keeping the k highest items.

system is actually *adding value* to the system and not only strengthening it.

The personalization of search results for some user based on his interest profile was proposed by [23, 24]. None of these systems use interest profiles for improving search results in a collaborative manner and not for just one single user but all of them. Recent work [2, 19] explores the use of social annotations to improve Web search, based on online bookmarking platforms such as *del.icio.us*. A drawback of this approach is that such services requires more effort from the user to bookmark and annotate the items he accesses. An automatic system is certainly more effective at attracting users.

A decentralized storage specifically designed for P2P Web search has been proposed in [10] for term frequency–inverse document frequency (TF-IDF). The authors present the priority-based and packing-based routing techniques that our system also uses. Nonetheless, they do not provide any mechanism for dealing with the skewness of terms popularity, and they do not deal either with the terms extraction, nor use user-centric information to answer the queries. Similarly, Lopes *et al.* have proposed in [12] a storage mechanism for large, non-mutable data on top of a DHT. This mechanism, while presented for TF-IDF, can be applied to any collection of large data. It uses B+-trees to balance the storage load over several peers in the DHT. Finally, a set of proposals uses the inverse routing paths convergence property: for load balancing [20], for replication and performance in BeeHive [16].

5. CONCLUSION

In this paper, we have shown how decentralized and collaborative mechanisms can efficiently complement existing large-scale search systems. The objective is twofold: avoid the typical *bootstrap problem* faced when designing distributed variants of centralized system; and demonstrate that collaborative approaches can provide added value and new services for the users.

We have presented the basic architecture and building blocks of a novel *collaborative ranking service*, which leverages user-centric information such as interest profiling and relevance tracking in order to return search results lists tailored to the user interests. These results are expected to be more relevant to the user that those returned by centralized search mechanisms based only on structural or popularity ranking. Result lists are based on the interest profile of the user, and special care is taken to ensure that the system keeps as much *information diversity* as possible to fulfill the requests from large sets of different users.

Our work builds upon a solid, fully distributed and specially designed P2P system. This system combines classical key-based routing with an application specific storage layer, as well as specifically designed load balancing mechanisms that go beyond typical “blind” approaches used in other P2P systems, by taking into account the specific needs and characteristics of the storage layer.

We are currently evaluating our system on a prototype implementation by observing its behavior along several metrics: (1) quality of the produced results, both analytically and based on sample user opinions, (2) overhead incurred by using the system at the client and on the distributed system, and (3) performance and flexibility of the load balancing, routing, and storage layers. Our tests are performed using both synthetic and real traces for both user-related behaviors (queries and accesses) and system-related characteristics (churn, failures, etc.). Deployment is being conducted both in emulated networks and on PlanetLab.

6. REFERENCES

[1] R. Akavipat, L.-S. Wu, F. Menczer, and A. Maguitman. Emerging semantic communities in peer web search. In *Proc. of P2PIR'06*, pages 1–8, 2006.

[2] S. Bao, G. Xue, X. Wu, Y. Yu, B. Fei, and Z. Su. Optimizing web search using social annotations. In *Proc. of the 16th WWW*, pages 501–510, Banff, Alberta, Canada, 2007.

[3] M. Bender, S. Michel, G. Weikum, and C. Zimmer. The Minerva project: Database selection in the context of p2p search. *Datenbanksysteme in Business, Technologie und Web*, 65:125–144, 2005.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[6] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proc. of IPTPS'03*, Berkeley, CA, Feb. 2003.

[7] H. Gylfason, O. Khan, and G. Schoenebeck. Chora: Expert-based p2p web search. In *Proc. of AAMAS'06*, Hakodate, Japan, May 2006.

[8] S. B. Handurukande, A.-M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the edonkey network, and implications for the design of server-less file sharing systems. In *Proc. of Eurosys'06*, Leuven, Belgium, Apr. 2006.

[9] M. Henzinger. Search Technologies for the Internet. *Science*, 317(5837):468–471, 2007.

[10] F. Klemm and K. Aberer. Aggregation of a term vocabulary for peer-to-peer information retrieval: a dht stress test. In *Proc. of DBISP2P'05*, Trondheim, Norway, 2005.

[11] J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. The feasibility of peer-to-peer web indexing and search. In *Proc. of IPTPS'03*, Berkeley, CA, 2003.

[12] N. Lopes and C. Baquero. Taming hot-spots in dht inverted indexes. In *Proc. of LSDS-IR'07*, Amsterdam, The Netherlands, 2007.

[13] T. Luu, F. Klemm, I. Podnar, M. Rajman, and K. Aberer. Alvis peers: A scalable full-text peer-to-peer retrieval engine. In *Proc of P2PIR'06*, 2006.

[14] W. Meng, C. T. Yu, and K.-L. Liu. Building efficient and effective metasearch engines. *ACM Computing Surveys*, 34:48–89, 2002.

[15] J. A. Patel and I. Gupta. Bridging the gap: augmenting centralized systems with p2p technologies. *SIGOPS Operating Systems Review*, 40(3):14–17, 2006.

[16] V. Ramasubramanian and E. G. Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc of NSDI'04*, Berkeley, CA, USA, 2004.

[17] A. Rowstron and P. Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware'01*, pages 329–350, Nov. 2001.

[18] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of SOSP'01*, pages 188–201, Banff, Canada, Oct. 2001.

[19] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *Proc. of SIGIR'08*, Singapore, 2008.

[20] S. Serbu, S. Bianchi, P. Kropf, and P. Felber. Dynamic load sharing in peer-to-peer systems: When some peers are more equal than others. *IEEE Internet Computing, Special Issue on Resource Allocation*, 11(4):53–61, 2007.

[21] G. Skobeltsyn, T. Luu, I. P. Zarko, M. Rajman, and K. Aberer. Query-driven indexing for scalable p2p text retrieval. *Future Generation Computer Systems*, In Press, Accepted Manuscript, 2008.

[22] T. Suel, C. Mathur, J.-W. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In *Proc. of WebDB'03*, Suzhoum, China, 2003.

[23] B. Tan, X. Shen, and C. Zhai. Mining long-term search history to improve search accuracy. In *Proc. of SIGKDD'06*, pages 718–723, Philadelphia, PA, USA, 2006.

[24] J. Teevan, S. T. Dumais, and E. Horvitz. Personalizing search via automated analysis of interests and activities. In *Proc. of SIGIR-IR'05*, pages 449–456, Salvador, Brazil, 2005.