Site-To-Site (S2S) Searching with Query Routing Using Distributed Registrars

Wan Yeung Wong
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
wywong@cse.cuhk.edu.hk

Irwin King
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
king@cse.cuhk.edu.hk

ABSTRACT
Site-To-Site (S2S) searching is a novel Web information retrieval method which uses peer-to-peer framework with CGI as protocol. It helps site owners to turn their websites into autonomous search engines without extra hardware and software costs. Thus, it improves some shortcomings of Conventional Search Engines (CSE) such as centralized and outdated indexing by distributing search engines over websites which maintain their updated local contents. However, it has query flooding problem. In this work, we extend S2S searching and propose our query routing algorithm to solve the query flooding problem by using distributed registrars for storing content summaries of adjacent sites. We also address some shortcomings of CSE and introduce S2S searching with some related work and comparisons. Moreover, we describe the system architecture and query routing algorithm. Finally, we summarize the experimental results and show that S2S searching works well in a large scaled S2S network.

Categories and Subject Descriptors
H.3.3 [**Information Search and Retrieval**]: Retrieval models, Search process. H.3.4 [**Systems and Software**]: Distributed systems.

General Terms
Algorithms, Performance

Keywords
Search Engine, Web Information Retrieval, Site-To-Site (S2S), Peer-To-Peer (P2P), Query Routing, Distributed System

1. INTRODUCTION
Information retrieval on the Web is significant. Conventional Search Engines (CSE) like Google and Yahoo have three shortcomings. They are (1) centralization of resources used, (2) outdated search results and (3) no control over information shared by content owners.

1. Centralization of Resources Used: CSE are centralized which require powerful servers to handle search requests. They also need large storage space to store crawled contents and indexes. Hardware cost is expensive for achieving high performance. Moreover, CSE have single point of failure.

2. Outdated Search Results: CSE preprocess the search by crawling Web contents and building corresponding indexes. Usually, the crawled contents and indexes are outdated as Web pages are being updated from time to time. So we may have some dead or outdated links in search results.

3. No Control over Information Shared: CSE crawl published contents on the Web and make them become searchable without their owners' permissions. The owners may only want their contents to be accessed by their authorized people. Although they could make their contents escape from crawlers by setting passwords or removing links in their Web pages, it is inflexible and requires technical knowledge. In addition, the owners are unable to alter their ranking strategy for their prioritized contents. Although they could use meta-tags and different headings in their contents, it is inflexible and does not guarantee how their contents are ranked eventually.

1.1 Site-To-Site Searching
We previously proposed a novel Web information retrieval method called Site-To-Site (S2S) searching [7] which uses Peer-To-Peer (P2P) framework with CGI as protocol. It helps those site owners to turn their sites into autonomous search engines without extra hardware and software costs. It improves the three aforementioned shortcomings of CSE by providing (1) decentralized searching, (2) updated search results and (3) full control over information shared by content owners. In this work, we extend S2S searching and propose our query routing algorithm to solve the existing query flooding problem.

1. Decentralized Searching: S2S search engines are decentralized so they need less powerful machines to handle search requests and less storage space to store the local index. We could use a search form in any one of the sites which joins S2S network to start searching Web contents. The query initiated site propagates the query request to its adjacent sites. Each site propagates the request, searches its own Web contents and gathers search results. Finally, all the search results are propagated back to the query initiated site which are ranked and displayed to the users. In addition, there is no single point of failure. If some sites are down, we could still use other search forms in other sites.

2. Updated Search Results: S2S search engines always provide most updated search results because each site maintains its own local index which is always up-to-date. When a local content in a site is updated, the corresponding index is recalculated. So we do not have any dead and outdated link in search results.

3. Full Control over Information Shared: S2S search engines allow site owners to fully control their information shared as they become administrators of their own search engines. They could selectively disable their published contents to be searchable for achieving the privacy. They could also prioritize their own contents and alter their ranking strategy of their own search engines for advertising and ranking their contents in a more customized way. However, adjusting the priorities of their contents in other sites' search results is not allowed in order to prevent cheating.

2. RELATED WORK

1. Gnutella [1]: It is a file sharing protocol which applies pure P2P paradigm without any centralized component. Its model is very similar to S2S searching but Gnutella is designed for searching files in personal computers instead of websites. We extend this model for information retrieval on the Web. Moreover, Gnutella has the query flooding problem which generates a lot of network traffic and wastes the resources of all irrelevant peers. On the other hand, S2S searching solves this problem by applying the proposed query routing algorithm.

2. YouSearch [4]: It is a Web search engine which is designed for searching contents in the network of personal Web servers, while S2S search engine targets on more general Web servers. YouSearch applies hybrid P2P paradigm which depends on a centralized registrar to summarize contents of each peer. Each peer creates its own content summary and pushes the summary to the registrar. When we search for contents by a peer, it queries the registrar to obtain a list of relevant peers. Although this model solves the query flooding problem, using centralized registrar is not scalable and it introduces registrar flooding problem. On the other hand, S2S searching solves these problems by applying distributed registrars.

3. CAN [5] and Chord [2]: Distributed infrastructures of both CAN and Chord use Distributed Hash Table (DHT) to map a filename to a key. Each peer is responsible for storing a certain range of key-value pairs. When a peer searches for a file, it hashes the filename to a key and asks the peers responsible for this key for the actual storage location of that file. For S2S searching, each site stores content summaries of its adjacent sites. When a site searches for contents, it matches the keywords with content summaries and calculates scores of adjacent sites. Query is only routed to the sites with high scores.

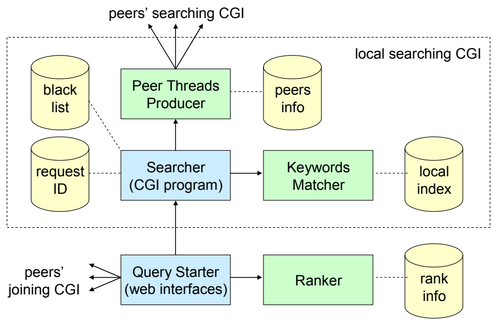

Figure 1. System Architecture

3. ARCHITECTURE AND ALGORITHMS

3.1 System Architecture

S2S searching includes a search engine core together with some CGIs for sites to communicate with each other. It also includes administration pages for site owners to administrate their search engines, and Web interfaces for search engine users to search for Web contents and then display results. We focus on the core module which has five components (rectangular boxes) as shown in Figure 1.

1. Query Starter: It provides Web interfaces for search engine users to search the target information. When it receives a query request from the search form, it first generates a unique request ID. Then the ID is passed to the local *searching CGI* together with the keywords and other parameters in the search form. The local *searching CGI* program searches the target information in the local site and also forwards the query request to adjacent sites. Two sites are adjacent if and only if they know the addresses (*starting URLs*) of each other. It returns a list of results to the *query starter* together with the *starting URLs* of the sites that contain any document which similarity is greater than the configurable quality threshold. The *query starter* joins those high quality sites by calling their *joining CGIs*. Finally, it forwards the results to the *ranker* and outputs the ranked results.

2. Searcher: It is the entry point of the local *searching CGI*. When it receives a query request, it first checks whether the requester is in the black list which is a list of banned IP addresses. After passing the black list test, the *searcher* checks whether the request ID exists in the file in order to ignore repetitive requests. If it passes the request ID test, the *searcher* adds the current request ID to the file and checks whether Time-To-Live (TTL) value from the CGI parameters is greater than zero. If it is, then the *searcher* asks the *peer threads producer* to route the query request to adjacent sites. At the same time, it asks the *keywords matcher* to search local contents by giving the keywords. The *peer threads producer* and *keywords matcher* work in parallel. After some time, both of them return results which include documents' and sites' information. The *searcher* then gathers these results and returns to the requester.

3. Peer Threads Producer: It is responsible for spawning threads to route a query request to adjacent sites. When it is called by the *searcher*, it spawns a requested number of threads. After spawning the threads, each one calls a unique site's *searching CGI* and waits for its return. The *starting URLs* of the sites are stored in the peers information file. Since the waiting time for other sites to return their results is dominant, the *peer threads producer* is always idle after sending the query request to all adjacent sites. So the *keywords matcher* gets full CPU resource to search local contents at that time. Finally, the *peer threads producer* finishes waiting all threads to join and returns the gathered results to the *searcher*.

4. Keywords Matcher: It is responsible for searching local contents by giving keywords. When it is called by the *searcher*, it extracts the keywords and tries to match with the local index stored in the file. Once it matches, the similarity is calculated. To index a document, we extract its words and calculate the corresponding term frequencies by

$$TF(w_i) = \frac{f(w_i)}{\max_{k=1}^{N} f(w_k)}, \tag{1}$$

where $f(w)$ is the frequency of the word w and N is the number of different words in the document. To match the keywords, we calculate the similarity of the document by

$$sim = \frac{1}{n}\sum_{i=1}^{n} s_i \quad \text{where} \quad s_i = \begin{cases} TF(w_j) & \text{if } \exists_{w_j} w_j = k_i \\ 0 & \text{otherwise} \end{cases}, \tag{2}$$

n is the number of different keywords and k is a given keyword. Finally, the *keywords matcher* returns the results to the *searcher*.

5. Ranker: It is responsible for ranking search results based on priorities and similarities of documents which are real numbers between zero and one. Priorities are stored locally. Therefore, only local documents take effect of their priority values because site owners should have right to advertise their documents in their own search engines. Other site owners are not allowed to rank their documents higher in other sites by setting higher priorities. This avoids cheating. If a document does not belong to a site, it is always set to the normal priority. The final ranking value is calculated by $rank = p \times priority + s \times sim$ where $p + s = 1$. The ranking parameters p and s are real numbers between zero and one which are configurable by site owners according to their preferences. Finally, the *ranker* sorts the search results in descending order by the ranking value. The ranked results are returned to the *query starter*.

3.2 Query Routing Algorithm

P2P search engine like YouSearch routes a query by looking up a centralized registrar which summarizes the contents of each peer. However, it has two problems. (1) It depends on the centralized registrar which is not scalable. The centralized registrar also needs to store many data of all peers. (2) It has registrar flooding problem because all peers query the centralized registrar from time to time. Moreover, when peers update their local contents, they also push their content summaries to the centralized registrar. In order to solve the query flooding problem in S2S searching, we improve the method of YouSearch and propose our own query routing algorithm which is fast and scalable. The idea is to distribute registrars over sites. Each site manages its own registrar which contains content summaries of all adjacent sites. This model solves the aforementioned scalability problem of YouSearch.

Content Summary Generation: The registrar is a file which contains *starting URLs* as IDs of adjacent sites and their corresponding content summaries. The content summary of a site is a fixed size hash table which stores the scores of different words of all documents in a site. We define the content summary as $S = \{s_i \mid s_i \in \Re \wedge 0 \le s_i \le 1 \wedge 1 \le i \le n\}$ where n is the number of blocks in the hash table. In order to obtain an even distribution, n should be a prime number. In our S2S system, n is 2,047. Given a lower-cased alpha-numerical word w, the hash function of S is defined as

$$H(w) = \left[\sum_{i=1}^{l} 27^{i-1}(c_i - 96) \right] (\bmod n), \tag{3}$$

where l is the length of w. In our S2S system, the maximum value of l is fixed to thirteen to prevent integer overflow. If the i^{th} character is a number, then c_i is fixed to 96. Otherwise, c_i is the ASCII code of the alphabet. To build a content summary, we traverse index files and get the information about the words and frequencies. Let N be the total number of different words in a site. We define the word set as $W = \{w_i \mid 1 \le i \le N\}$. For each word w_i in W, its total frequency $f(w_i)$ of all documents is calculated. We define the word importance of the i^{th} word relative to the whole site as

$$I(w_i) = \frac{f(w_i)}{\max_{k=1}^{N} f(w_k)}. \tag{4}$$

We also define the hash set of all words which has the same hash code i as $HS_i = \{w_j \mid w_j \in W \wedge H(w_j) = i\}$. Then the i^{th} element of the content summary is calculated by

$$s_i = \begin{cases} \dfrac{CL_i}{|HS_i|} \displaystyle\sum_{w_j \in HS_i} I(w_j) & \text{if } |HS_i| > 0 \\ 0 & \text{otherwise} \end{cases} \quad \text{where } CL_i = |HS_i|^{-1}. \tag{5}$$

Actually s_i not only stores the average word importance for all w_j in HS_i, but also stores the confidence level CL_i which is inversely proportional to the number of collisions in s_i. So the above equation already handles all collisions. When we compare s_i in different sites, the larger value of s_i has, the more important and confident the word w_j appears in that site.

Score Calculation: When a site needs to route a query, it first looks up its own registrar. For each content summary S in the registrar, it calculates the score of an adjacent site with the given lower-cased alpha-numerical keywords. Let m be the number of different keywords. We define the keyword set as $K = \{k_i \mid 1 \le i \le m\}$. The total score of a site, which is normalized between zero and one, is defined as

$$score = \frac{1}{m} \sum_{i=1}^{m} s_{H(k_i)}. \tag{6}$$

After calculating scores of d adjacent sites, a list of relevant sites with some false positives is obtained. It routes the query to x sites, which have the highest scores, such that $x = \lceil f \cdot d \rceil$ where the traffic reduction factor f is a real number between zero and one for reducing the network traffic. For example, if f is 0.2 and the current site has ten adjacent sites, then the query message is routed to the two sites which have the highest scores.

Infrequent Query Flooding: The proposed algorithm greatly solves the query flooding problem. However, it has two shortcomings. (1) The search is incomplete because not all sites in the same S2S network within a specific TTL receive the query request. So they are unable to search their local contents and return their results to the requester. (2) The results obtained are local optimal because the proposed algorithm performs a greedy search. Not all sites search their local contents and return their results. So the requester is unable to obtain global optimal results. We understand there is no free lunch in this world. So we make a balance between query routing and flooding. We enable infrequent query flooding in a site with a small probability p. So when it receives a query request, it has the probabilities p and 1-p to use the query flooding and query routing algorithms respectively. With infrequent query flooding, the proposed algorithm improves the search to be semi-complete and semi-global optimal.

Registrar Maintenance: In order to maintain most updated content summaries, we propose our registrar maintenance algorithm. Recall that every site stores its adjacent sites' content summaries in its own registrar. It is necessary for adjacent sites to send their updated content summaries if their contents are updated. So when a site updates its local contents, it recalculates its local index and also content summary. If the updated content summary is different from the old one, then it broadcasts its updated content summary to all adjacent sites by calling their *updating CGIs*. This model does not introduce the aforementioned registrar flooding problem of YouSearch because usually update is infrequent and it only disturbs adjacent sites in one level.

4. EXPERIMENTS AND DISCUSSIONS

There are three experiments. (1) We measure the indexing and searching time in a local site. (2) We also measure the S2S searching time in one thousand sites. (3) Finally, we measure the quality of the hash function (see Equation 3) in the query routing algorithm. We run the experiments on several computers of Sun Blade 1000 with Solaris 8, Java 1.4.2 and Jakarta Tomcat 3.3.1 as Web server. They have fast network speed (100Mbps) to simulate those Web servers which are placed in data centers. On the other hand, they have slow file I/O speed as they only use Network File System (NFS) instead of local raid-disks. Their overall performances are less than those dedicated Web servers.

1. Local Indexing and Searching Time: To conduct this experiment, we randomly select thirty-one HTML posters in the

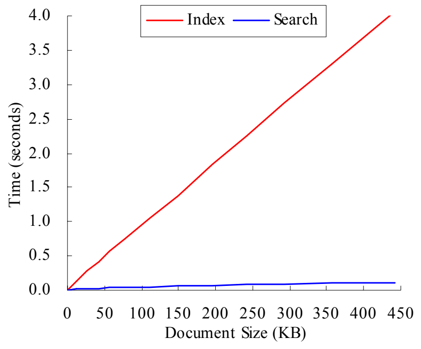

Figure 2. Local Indexing & Searching Time

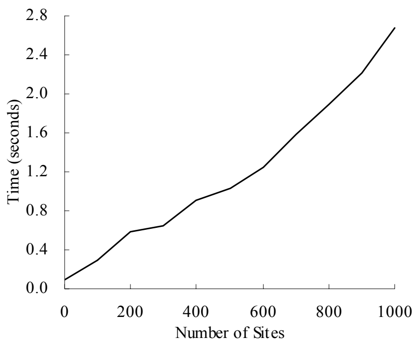

Figure 3. S2S Searching Time

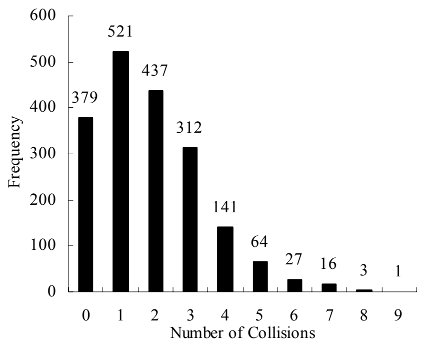

Figure 4. Quality of Hash Function

twelfth International World Wide Web Conference (WWW2003) [6]. The total document size of the text data to be indexed and searched is 441KB. We incrementally add the document size from 11KB to 441KB and measure the corresponding indexing and searching time with some random keywords (see Figure 2). Indexing and searching both take linear time. From the experimental results, we know that the indexing algorithm is acceptable in time and the searching algorithm is very fast. Actually the bottleneck is in the file I/O as we store the documents and index in NFS.

2. S2S Searching Time: We conduct this experiment by simulation. The local searching time of each site is fixed to 0.1 second. The total number of sites to be searched is one thousand which are evenly distributed in two computers. We incrementally add the number of sites from one hundred to one thousand and measure the total searching time (see Figure 3). The time measured is the worst case that every site has only one degree of fan-out so that the query request is oscillated between the two computers through the network. From the experimental results, we know that S2S searching is efficient in a large scaled S2S network. Each of the two computers needs to handle five hundreds requests and they still work well. In the real world, the sites are usually distributed in different Web servers and a dedicated Web server could handle a lot of requests within a short time. The efficiency is due to the fact that the searching process is highly distributed and is done in parallel. The bottleneck is in the local searching process which could still be improved by applying a better local searching algorithm.

3. Quality of Hash Function: To conduct this experiment, we use those thirty-one HTML posters in WWW2003 again. There are 2,047 blocks in the hash table. After removing the stop words, the total number of different words to be hashed is 5,423. For an even distribution, each block should have about 1.65 collisions. We measure the actual number of collisions in each block (see Figure 4). There are 1,901 blocks used. The usage is about 93%. The average number of collisions is about 1.85 with standard deviation 1.53. About 70% of blocks have two or less collisions. From the experimental results, we know that the hash function is acceptable in quality. The total usage is very high so that it is efficient. The average number of collisions is only a bit more than the ideal number of collisions with quite less standard deviation. So it is quite evenly distributed. Actually, the quality of the hash function depends on l and n (see Equation 3). If they are larger, then the quality is higher but the calculation time and memory requirement are more.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we focus on the extended S2S searching and proposed query routing algorithm. We also address the shortcomings existed in CSE. Then we introduce some related work with some comparisons. Moreover, we describe the system architecture with some algorithms. Finally, we summarize the experimental results which measure the performance of local indexing and searching, S2S searching and the quality of the hash function in the proposed query routing algorithm. From these results, S2S searching works well in a large scaled S2S network. Since S2S technology is a new topic, there are some research could be done in the future. We plan to integrate some security algorithms because the system does not check if the requests are from the trusted sites. We may also extend S2S searching to include multimedia information retrieval like Discovir [2].

6. ACKNOWLEDGMENTS

The work described in this paper was fully supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK 4351/02E and CUHK 2050259).

7. REFERENCES

[1] Gnutella website. http://www.gnutella.com

[2] I. King, C. H. Ng, and K. C. Sia. Distributed Content-Based Visual Information Retrieval System on Peer-to-Peer Networks. In ACM Transactions on Information Systems (TOIS), Volume 22, Issue 3, 2004.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In Proceedings of ACM SIGCOMM, 2001.

[4] M. Bawa, R. J. Bayardo, S. Rajagopalan, and E. J. Shekita. Make it Fresh, Make it Quick – Searching a Network of Personal Webservers. In Proceedings of 12th International World Wide Web Conference, 2003.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In Proceedings of ACM SIGCOMM, 2001.

[6] The 12th International World Wide Web Conference website. http://www.www2003.org

[7] W. Y. Wong. Site-To-Site (S2S) Searching Using the P2P Framework with CGI. In Proceedings of 13th International World Wide Web Conference, 2004.