Sinergia: Improving Browsing By Exploiting Community Knowledge

Javier Nicolás Sánchez
Rohit Singh
Stanford University

March 22, 2001

Abstract

Finding useful information in the web using today's search engine technology is still a time consuming, and often futile, task. This is not just because of the overwhelming number of results returned but also because of the uncustomized, one-size-fits-all results which do not care about the context and ability of the client. We propose to improve the situation by aggregating knowledge of what users with common preferences find useful while doing similar search queries. This knowledge can then be leveraged to provide refined results using that information. Furthermore, we extend the idea to present users with a customized list of pages related to their current page. We also use the concept of letting users rank different pages and clustering users based on these rankings. In this paper, we describe the motivation, design and implementation of an application to demonstrate proof-of-concept. We also discuss the general issues of scalability, portability and availabity etc. for this application

1 Introduction

Searches done using today's search engines usually require a time consuming iterative process. The user goes to, say Altavista, plods through the results page in search for links that seem to be appropriate for what she is looking for. If she finds it there, great! Otherwise, she will go on to some other engine, say Google. And of course, at every search engine she might have to go through many useless links before she finds something of interest. This happens because of many reasons, some of which are:

- the amount of content in the web today is overwhelming and a few words are not enough to narrow down the list of plausible matches.
- the search engine ranks pages on the basis of an algorithm which totally ignores the user preference, background/ability and the context in which the query was performed.
- no single search engine covers the whole web. So one has to go through many to find that elusive link.

Frequently, she might need to iterate the whole cumbersome process over various search engines. An obvious solution is to use a meta-search engine. It tries to collect information from different search engines automatically. By doing this, it saves the user from going through several search engines. Still, the number of links returned is big and the user has to face the hassle of digging through the mass of irrelevant information till she finds something useful. We noticed that we can improve the accuracy of links returned if we add to the search information that is relevant to the user's background. In particular, if a user belongs to a community of people who share similar interests (say a work-group in a company), then most probably they will be interested in the same kind of web pages (the common tools and references they use).

Our approach does not reinvent the wheel by building a new search engine. Instead we try to improve the accuracy of searches by enabling an implicit cooperation between users in a community to filter the links that are interesting to them. So, the users still can use the existing search engine technology and go through the (time consuming) process. But if someone in her community already did a search of the same material then she can save her time by just looking at the links those people found interesting for the same search.

We also attempt to leverage our information about commonality in user preferences to provide customized lists of related pages. In this context, we also support user feedback on web-pages. Unlike the approach followed in some commercial products recently, we do not attempt to annotate pages. Instead we just have the simple concept of voting for a page and ranking it.

Our approach borrows and combines ideas from a lot of domains. That transformational proxies can be used to provide tailored information in an unobtrusive way to the community they serve has been showed by [FGCB98]. Aggregation and analysis of usage patterns of different users (in particular, bookmark lists) to enhance search and browsing has also been studied [PGRC00]. The use of user surfing behavior to provide dynamically tailored information has also been investigated [WJGD96]. To generate clusters of users with similar preferences, we also took ideas from collaborative filtering [BC92]

In section 2, we describe our goals and a business perspective of the project. Section 3 deals with the architectural and design issues of the project. Section 4 talks about the implementation details of the project. We then evaluate our implementation on the issues of usability, stability, availability and scalability in Section5. Section 6 covers related work and how our compares with them. Finally Section 7 contains our conclusions and possible future directions of the project.

2 Goals and Business Model

When we started off, we had the following assumptions in mind:

- Members of a community served by a proxy often have similar backgrounds, preferences and interests. This assumption is usually justified because a proxy is mostly used to serve a special segment of users (say mobile users, a company work-group, students and faculty of a department). However this assumption is not totally necessary. After a while, we can use the clustering information generated by the proxy to develop different communities each of which can be, then, handled independently of other communities.
- The items in the result of a search query that are of interest to one user in the community will probably be of interest to others as well.
- If there is a high correlation between the rankings assigned to a set of pages by two users A and B, it is highly likely that if A likes a new page, then B will like it too.

Apart from these, we assumed that the browser supported forms and Javascript. Considering the current state of browsers, this is a very fair assumption. Apart from this, we did not make any other *technology penetration assumptions.*

2.1 General Goals

Our objective was to design a service that will enable users to leverage the experience and knowledge of their peers to fasten and facilitate their surfing and searching. To demonstrate the idea, consider the following scenario about the service we would like to develop:

Suppose that the service serves the general CS department community and has been collecting information about the pages visited by the members of the community. Drew, a graduate student in the CS dept, is taking CS241 and is interested in ubiquitous computing related articles on the web, but she does not have the time to do an extensive search and hence can not get the most relevant results using a traditional approach. As like most of the people in the department, Drew is also subscribed to our service. Many of Drew's friends in the department are also taking CS241 and some of them have also searched for articles on that topic. Thus the service's data-repository might already contain information about some related pages which others have found relevant in the context of a search for "ubiquitous computing". Thus when Drew searches for these words, the service can point her to this list of (known to be relevant) pages, saving her a lot of time.

The general goals for a service like this would be:

- Improve the search and surfing experience of the user through the use of information gathered by the proxy. The information gathering should be done with minimal disruption of the users' browsing experience.
- At the same time, the training effort required from the user should be minimal and the client-side interface should be highly portable. Any artificial limits on the number of users that can be served should be avoided
- Complement and extend current search technology to provide better and more relevant search results.

2.2 Business Perspective

2.2.1 Target Population

Our target population is a group of people who are interested in similar things. This might be a company work-group where everybody needs to look up similar information on intranet and the internet. An ideal example would be a group of a users in an educational institution. The general expertise of the users there would mean that general results (meant for the layman) of a search query are often inadequate for them. At the same time, most users would be looking for similar information. Another target group would be a group of users, in a small geographical location, served by a proxy of an ISP. In this case, all the users will have similar preferences for local news, activities and issues.

2.2.2 Revenue Model

This application can either be used as a service and people could subscribe to it for a, say, monthly fee. Otherwise, it can be licensed to to institutions (like universities and ISPs) for a capacity-based fee. Of course, in the spirit of free software, it can also be given out free as the authors believe that costly software is the root of all computing evil.

3 Architecture and Design

As mentioned in the previous section, we would like to collect and disseminate information as transparently as possible. Also, to make it as usable as possible we also wanted minimal client side modifications. The insight here, borrowed from [FGCB98], [WBI], is that a proxy can not only observe all the net traffic in the community but it can also perform analysis and transformations almost transparently. Thus our basic approach was to build the system on top of a proxy. However, it is desirable (eg. for fault isolation) that the hard state and the soft state of the system be in different compartments [FM01]. The hard state should be kept as available as possible and the soft state should be such that it can be regenerated with very little degradation in performance as viewed by the user. In that spirit, each instance of the proxy carries its own soft state and the hard state (across all instances) is stored in a centralized database. The database can be made highly available by resorting to the various advanced techniques available for this matter. However to facilitate user access to the information stored by the service, we also need a web-server that can communicate with both the proxy and the back-end database. Given these main components, here is the basic architecture of our system:

3.1 Proxy

An important issue to resolve was of attaching user identifications to the HTTP requests made from an IP address. We resolved this issue by resorting to the concept of a session.

Session A session is a period of network use by a client at a single IP address (which need not be the same across sessions). The session is tracked by the proxy and spans all the sites the user visits. A session is terminated after a fixed time-period of inactivity or if another session by the same user is started from a different IP address.

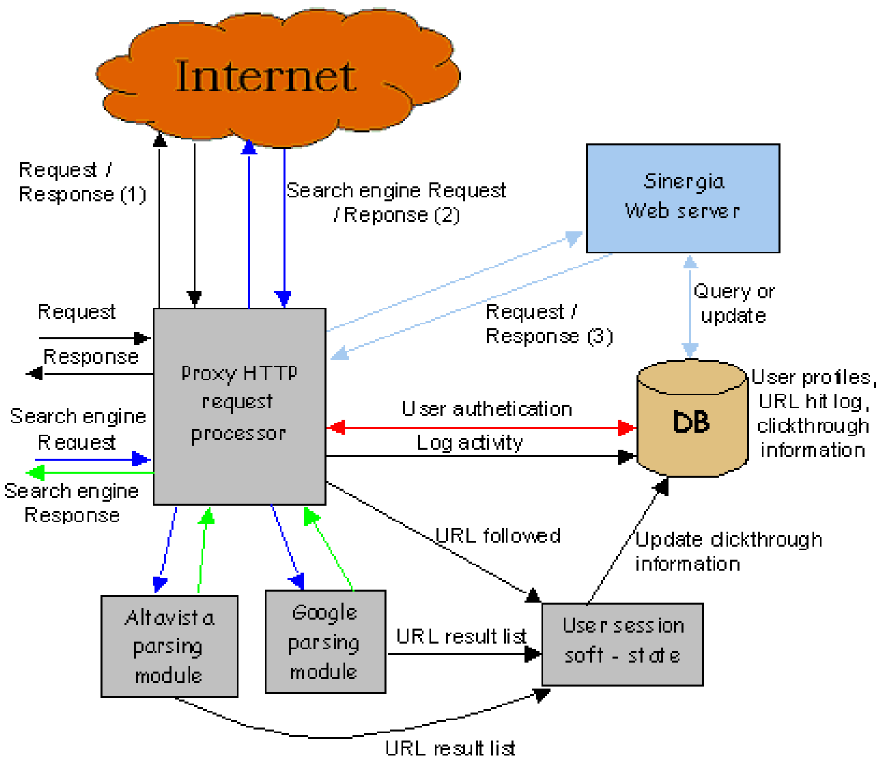

Fig: 1 This describes the basic architecture and the data/control flow among the basic components. Most HTTP messages are passed on unmodified by the proxy (1). However, the ones pertaining to search queries are observed and modified (2). HTTP requests to the Sinergia webserver are modified by appending userid information to the URL(3). This is the only point of contact between the proxy and server.

Unlike a normal web-site, we could not use a cookie since the proxy must be functioning transparently. It is also not advisable to hard-code mappings from IP address to user-ids because a user may access the net from different locations. The only solution that satisfied our criteria of minimal client-side modifications was of letting the user log on to a proxy. When the user starts a session, before the requested page is passed on, the user gets a form (in his browser) where she needs to authenticate herself. The whole process is very simple and exactly like the 'Sign me up' process on a normal website. Once authenticated, the proxy keeps track of HTTP requests and logs the urls visited by the user. Unless a HTTP request points to a search engine that the proxy supports, the request and the response are passed on unmodified. Currently, the list of supported search engines includes Google, Altavista and Hotbot.

If the HTTP request is a search request to a supported search engine, the proxy intercepts the result page. Then it does two things:

- The set of urls U returned by the search engine are stored in a hash table of *active searches*. Each active search is indexed by the userid i and the sequence of keywords (κ) used. A search becomes inactive (and is discarded) as soon as the session times out or if the user starts a new search. In subsequent requests from the client, the proxy checks if the requested url $u \in U$. If it does, then the proxy flags u as being followed by the user i in the context of the query κ. This hashtable is stored at the proxy since its loss will not be of catastrophic impact. At the end of a session or when the search becomes inactive, this soft state information (i.e flagged urls) is transferred to the backend and becomes persistent.

- The proxy also modifies the result page to contain a link that points to our web server along with the associated query.

We faced a problem in communicating the userid of a request to our web-server. A bad approach would have been to pass this userid as a global variable of some sort. However, we wanted to retain the capability of running the proxy and the webserver on different machines (to ensure availability and scalability). As a solution, all HTTP requests pointing to our webserver are modified into a fat URL that contains the userid information too. This modification is done by the proxy and is completely transparent to the user.

3.2 Web Server

The webserver is completely independent of the proxy except that they both communicate with the same backend database. It has three main functions:

- To provide a front end for querying our database of keywords and matching urls. The queries can either be made directly using the form at the main page or by passing a fat URL that encodes the query. This is the mechanism used when the proxy embeds a link in the results page.
- To provide an interface for users to vote pages and see page rankings. On the client side, the interface to get and set ranks is provided by personal toolbar buttons on the browser (implemented in Javascript). Our ranking mechanism allows users to rank pages on a 3-point scale.
- A user may wish to lookup pages that are related to the page she is currently visiting. There are a lot of sources on the net that offer this service. However, the webserver has the advantage that it knows the user and her profile (in terms of her browsing preferences). Thus, it can provide filtered results that matches her profile.

3.3 Database

The database is an off-the-shelf relational database. The schema used is described in Appendix A. The database contains the hard-state information and is the common backend for all instances of the proxy and the webserver. Also, a lot of processing is done solely at the database level. For example, our current clustering (of users) algorithm is executed solely at the database level. Hence it is trivial to insert a more complicate information retrieval technique to do processing of the database. The availability of the database is critical to our system. However, we can use numerous available techniques like replication and distribution to ensure that the database is fault tolerant, highly available and scalable [GDW00]

4 Implementation: The Concept at Work

We had problems finding a proxy that would allow us to perform authentication and session management across different sites. The popular proxy/cache application `Squid` was very hard to understand and modify. A Java-based proxy, Pluxy [PLXY], did allow plugins to watch and modify the HTTP stream but it was very hard to modify it to implement our session concept. To fit our needs, we designed our own proxy. However, we did use a lot of the ADS [HLBF00] class libraries for parsing HTTP requests and responses. We also copied Pluxy's mechanism of observing and modifying HTTP requests and responses.

Our backend database is MySQL, a freely available relational database. Again, it would have been a good idea to use Apache as our webserver. But we did not have the time to modify, configure and install it. We wanted to build a prototype fast and our webserver only needed to handle GET requests. So we just implemented the webserver ourselves.

4.1 Details of Sub Processes

Authentication We currently implement authentication via a form-based interface. A new user can sign up using a simple single-webpage interface. Users choose usernames and passwords and the proxy automatically assigns a unique userid to each user. After authentication, the proxy associates the client's IP address to the userid. For the length of that session, all HTTP traffic to/from that IP address is assumed to be done by this user. At any time there has to be a one-to-one mapping between the active users and active IP addresses.

Clustering of users We attempt to cluster users into groups where they share common preferences. Consider two users P_1 and P_2 and the set of urls U visited by the members of the community. Let the set of urls $U_1 \subseteq U$ and $U_2 \subseteq U$ ranked by P_1 and P_2 respectively. Also, let the set of urls that both of them have ranked: $U_{12} = U_1 \bigcap U_2$. Define a set of pairs Θ_{12} such that

$\Theta_{12} = \{\langle rank(u, P_1), rank(u, P_2)\rangle | u \in U_{12}\}$ where $rank(l, P)$ is the rank assigned to link l by the user P

we define our correlation function σ_{12} between P_1 and P_2 as follows:

$\sigma_{12} = \frac{\sum_{\langle r_1, r_2\rangle \in \Theta_{12}} similarity(\langle r_1, r_2\rangle)}{\|U\|}$

where $similarity(\langle i_1, i_2\rangle)$ is a function that returns a number $n \in [0 \ldots 1]$ which reflects the belief that the two rankings i_1 and i_2 assigned by P_1 and P_2 for the same url u are similar. If $i_1 = i_2$ then the function must return 1. In our current implementation the function returns 1 if the ranks are same and zero otherwise. Other more complex correlational techniques can be used just as easily as long as they fulfill our criteria of a similarity function. One efficient way to do similarity calculations is described in [FSGM]. If $\sigma > \tau$ where τ is a predefined threshold (currently 0.5), then the two users P_1 and P_2 are classified in the same cluster.

Finding related pages The processing of the database to generate clustering information can be done independently of the proxy and webserver's functioning. We chose to do it periodically every few hours. Once this is done, when a user makes a request to see pages related to the one she is currently viewing, the request is forwarded to Google. The results are filtered to allow only those pages which have been given a 'good' ranking by other users who belong to the same cluster as her.

Keyword to URL matching Whenever a user P follows a link u in response to a query Θ consisting of a set of keywords $\kappa_1, \kappa_2 \ldots$ the tuple $\langle P, u, \Theta = \{\kappa_1, \kappa_2 \ldots\}\rangle$ is logged in the database. Currently we store pairs of url and each keyword separately. For example, for a query $\Theta_1 = \{\kappa_1, \kappa_2\}$ that resulted in the url u being followed, we store two entries, one corresponding to $\langle u, \kappa_1\rangle$ and other corresponding to $\langle u, \kappa_2\rangle$. This has the advantage that we can easily answer complicated boolean queries rather than only simple queries. Boolean queries can be handled by introducing a flag variable in each row which tells how that keyword relates to that url. For example, if the row $\langle u, \kappa\rangle$ is flagged with 'NO', this implies that κ doesn't occur in u. Finally, when a user searches our database with a set of keywords (this could be a boolean query) we find the set of urls such that each url is indexed against one or more keywords. We filter this set according to the same binary operations as specified on the keywords.

More precisely, suppose that the user specifies a set of query words $\kappa_1, \kappa_2 \ldots \kappa_n$ and a boolean filter function b_κ. First of all, we calculate $\forall i$ the set of urls U_i which have been indexed against κ_i and then take their union $U = \bigcup_i U_i$. We apply the filtering function b_κ on U to filter out those urls that do not satisfy the query. After this we rank pages on the number of hits received and the number of keywords matched. While calculating the number of hits received by a page, we consider the hits received in 'normal' surfing as well as when the url was visited as a result of a previous search. The latter case is given more weight.

This procedure is simple to implement, maintain and we can perform complicated queries on it. However, its size grows as $O((\#urls)(\#keywords))$ which can become cumbersome. A way out is to use techniques from information retrieval. One such technique involves keyword vectoring where keywords are stored as bitvectors. Another approach would be to use hash tables. We believe that it is not a simple problem to choose between these approaches. It probably boils down to a space-time trade off. A bitvector approach would have much better space usage but its speed would be far less than our current implementation. Our implementation might be large but it facilitates fast querying. With current databases, large database sizes are not really a big problem.

4.2 Client Side Implementation

As mentioned before, one of our objectives was to have minimal client side implementation. This would help in letting users get started easily. By moving most of the work to the server side we also avoided facing portability problems across browsers. Currently, our only requirements for the browsers are that they support proxies and Javascript. This is already implemented by most major browsers.

The authentication part is exactly like any other form based interface. So it did not require any client side modification. To vote on pages and to find related pages, the user has to query our associated webserver. We implement this querying capability in the form of personal toolbar buttons written in Javascript. These are, essentially, bookmarks written in Javascript. Basic Javascript is fairly standard across almost all browsers and is supported pretty well. Installing these bookmarklets is as simple as click-and-drag.

Voting is implemented by two buttons, 'Vote' and 'Show Rank' which allow a user to vote on the current page's quality and view its ranking. To do a search on our repository of keywords and urls, the user has to go to the server and use it like a normal search engine. We found this much more straightforward than installing some client side plugin to do this. To find related pages to one being currently visited, the user only needs to click on the 'Find Related' button on the toolbar. Again, this button is a bookmark, written in Javascript. So we believe that the client side implementation is pretty stable while, at the same time, fulfilling our goals.

4.3 Web Server Implementation

The current implementation of the webserver is pretty basic. It could have also been implemented as few CGI scripts on a more complete server like Apache. However, we found it faster to just write the code ourselves. The web servers handles only four kind of GET requests. The first are requests to search the backend database given a list of query words (possibly combined by a boolean function). It also provides an interface to register a particular user's vote on a specified page. A parallel mechanism to display the average ranking of the specified page is also provided. Finally, the webserver also services requests to find pages related to the specified page.

Handling all database related queries via HTTP requests to the server implies that the server can handle requests that come not only from our applications but also other applications.

5 Evaluation

Our implementation of Sinergia was completed by the first week of march and has been runnning continously since then. We wanted to stress-test the system and to find out the bugs and problems in its usability. To accomplish that we asked some friends to use Sinergia as their proxy server. Almost all of these people are CS graduate students and they pushed the application to its limits and they provided great feedback on various matters. For example, they told us about keeping our authentication page as simple as possible. After their testing we realized that the hardest part of introducing our system is to bootstrap the process of generating relevant information in the repository. Unless there are sufficient users initially, it is hard to generate a sizable repository. This, in turn, implies that there is hardly any perceivable difference to the users while they are surfing or searching. But again, we can not get more users unless we have a good corpus of information already.

The way out of this chicken-and-egg situation is to initially start off the proxy in a stealth mode where there is no authentication and no modifcation of results, and the proxy only limits itself to collecting data and build its repository. Once the repository has reached a significant volume, then the system can start functioning with its full capacity. While in this gathering-of-data stage, voting and other related web server functions would be disabled.

5.1 Lessons learned

Appart from the above mentioned experiences, we observed the following:

- It was better to aim for a smaller target and then expand upon it, rather than being overambitious at the start.
- Even in the cases where source code of some building blocks was abailable it was not easy to adapt them to our needs.

- As observed in other web services composition related work [MPF00] , code that relies on parsing the HTML output of a service to gather information is very fragile and susceptible to breaking down whenever the service changes its HTML format or layout. We experienced a similar problem when Altavista changed their search server URL from www.altavista.com to search.altavista.com.

5.2 Reusability

Our implementation reuses a lot of ADS [HLBF00] code to do the parsing and construction of HTTP messages. Our code can, in turn, be used by people who want to develop transformational proxies by just pluggin in modules into the proxy. In particular, our work is suitable to be reused in projects where the need of a user session spanning many sites arises.

5.3 Customization

Because our current implementation of the client side is very simple, there is not much room for customization. An extension of this project would be to allow users to customize their preferences to certain categories and to use that information to do a better matching of profiles.

5.4 Failure behavior

From the user's point of view a transient failure in the proxy would imply that she was logged out, which will require her to repeat the authentication. That will not affect the correctness of the results returned, but will be perceived as a minor malfunction of the system. A failure of the webserver will be even less disruptive because, well, people are used to loaded servers and timeouts. A transient failure in the database would imply that no new users can log on to the system and the web server will not be able to return results to queries. However a failure in the database is highly unlikely because we assume that the database has been made higly available through replication and distribution of data. Our decision of separating the hard-state information from the soft-state was made to isolate faults and to reduce database transactions.

5.5 Scalability

Our basic assumption about the community (served by the proxy) having specific interests is rendered invalid on the internet scale. Therefore we are interested in scaling it to only to the level where it can serve a large intranet. For that scale of operations our architecture is sufficient:

- The database is independent of the other components and can be made as large, available and fault tolerant as needed.
- The web server can be run independently of the proxy and multiple instances of each can be run depending upon the workload.
- With our architecture replicating the proxy is slightly tricky because a proxy must maintain a state for the user across multiple HTTP messages. With multiple proxies, this might mean that each proxy has to be 'hardwired' to its clients. That is an undesirable solution because a hard failure on the proxy node will hang the browsers of the clients assigned to it. We think that this problem can be overcome by the use of a distributed data structure. In particular, we can use a distributed hash table as mentioned in [GBHC00]. Such a data structure would allow the same state of the client to be available to all the machines of the cluster working as proxies. Thus, the scalability problem can be solved too.

5.6 Implementation

Since we are a two person group and the work was a lot we ended up both doing parts of everything. As such it is hard to assign individual responsabilities.

6 Related Work

We have already explored some of the background behind our work. The idea of using a proxy as a transformational proxy has been demonstrated in [FGCB98]. A lot of research effort has gone into tackling the problem of finding relevant data in the always-growing Web. However, most of the work done is specially centered in the area of using server logs to obtain usage patterns and based on that, discover information that might be relevant to the user. In absence of an aggregating agent like the proxy, server logs are one of the few good ways of collecting traffic data. Other research projects in recent years have focused on finding novel ways of providing the user with useful results for their specific information needs. An interesting work in the area was done by [BB00] where they use the concept of relating a set of keywords given to the search engine with the URLs followed from the results page. Their work differs from ours in two points. Firstly, they are only concerned with improving the effectiveness of the alternate keyword set suggestions in Lycos Search (say, 'wild cat' instead of 'cheetah'). They do not take into account information other than the relation established by clickthrough data obtained from search result pages. Our approach goes further by incorporating specific information about the user to obtain more meaningful results. The second difference is that their approach is centered on the use of log data from a particular search engine, while our approach has the goal of collecting information from as many search engines as possible plus the regular browsing behavior of users. Nevertheless, the algorithm described in their work is of a high relevance to possible future extensions of our idea.

Another related work has been done by [WJGD96]. It was centered on the examination of log records of a web server in order to obtain categories of users that share common interests (determined by the pages they visit in the site). We share, with this work, the idea of clustering users according to matches in their browsing behavior. However, we also go further here by allowing the voting of pages which tells us which of the links followed were really relevant to the user.

Dynamically updating pages to reflect some information gathered by an agent monitoring the traffic has been investigated by [WBI] and [WJGD96]. They focussed on a client-side agent but the idea of annotating a page dynamically to pass on sone information is an idea that we both have used.

In the area of commercial products, we found two projects that are closely related to our idea. One is Alexa [ALEXA], which bases its page rankings on URL hits and annotations of web pages. Unlike in our project, they are targeting their service to the broad internet community. Therefore, they do not have the advantage of access to a community with specific interests. Also, they limit themselves to obtain information from users with Internet Explorer. A third difference is that their annotation process for web pages is not a simple task because it requires writing a comment about the page which in our case is a very simple and straightforward task. Finally, their funcionality of related URL is targeted toward sites, instead of pages, which rules out a lot of possibilities when trying to find similar content. Another commercial system that is along the lines of our project is Pandango [PNDGO], a system that is still under development. Their idea is to combine the peer-to-peer model to provide a distributed search engine based on bookmarks from users. The closest point of contact with our idea is that they try to match users with similar preferences and suggest result(s) for the query being made on the basis of information relevant to those users. These preferences have been evaluated by watching the clickthrough information. The main difference is precisely that they base their results in bookmarks. Data provided by bookmark lists is simply insufficient to evaluate the browsing preferences of a user because not all searches result in new bookmarks. Another big difference is the model of computation: we use a central infrastructure while they use a peer-to-peer based architecture where the search request for a node is processed by its neighbours.

7 Future Directions and Conclusion

We have explored the idea of using a proxy to enhance the browsing experience of a community. By combining ideas from various domains we have been able to build a prototype which we think is a powerful demonstration of the concept. In future, we would like to experiment with more sophisticated techniques of clustering users, incorporating user's explicitly stated profiles in these clusters. For example, clusters can be fine-grained and we can develop hierarchical clusters. As investigated in [BB00], we can cluster queries that are related to a topic and use that knowledge to retrieve more suggestions when the user queries on keywords from that (synonym) class. Other extensions would involve improving the backend implementation. If possible, we would like to find a more efficient way of storing url, keyword pairs. Another extension would be to integrate the proxy's data structure with the distributed data structures mentioned in [GBHC00] so that multiple instances of the proxy can view the same data.

Finally, the authors would like to place on record that even though this project ruined many nights, we derived a lot of satisfaction from it. While writing this writeup, we realized that our work seems to be actually useful, judging by the amount of similar work going on. We really believe that we are addressing a real open problem and our solution has some merit.

8 Acknowledgements

We would like to thank Prof. Armando Fox and Andy Huang for their valuable input and constructive critcism. More than a dozen friends tried our system and we are indebted to them. Javier talked to Prof Garcia-Molina about IR tools. Rohit talked to Mayank Bawa and some other people in the same context. We would also like to thank them for their valuable insight and suggestions.

References

[FGCB98] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, *Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives* IEEE Personal Communications, August 1998.

[PGRC00] Andreas Paepcke, Hector Garcia-Molina, Gerard Rodriguez-Mula, Junghoo Cho *Beyond Document Similarity: Understanding Value-Based Search and Browsing Technologies*, SIGMOD Records, 29(1):March 2000

[WJGD96] Tak Woon Yan, Matthew Jacobsen, Hector Garcia-Molina, Umeshwar Dayal *From User access Patterns to Dynamic Hypertext Linking* http://www-db.stanford.edu/pub/yan/1996/analog.ps

[BC92] N. J. Belkin and W. Bruce Croft *Information filtering and Information retrieval: two sides of same coin ?* Communications of the ACM, 35(12):29-38 1992

[FM01] Armando Fox and Murray S. Mazer, *On the Structure of Interactive Internet Applications* CS241 class readings 2001

[GDW00] Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom *Database System Implementation* Prentice Hall, New Jersey 2000

[WBI] *Web Intermediaries* http://www.almaden.ibm.com/cs/wbi/

[PLXY] *Pluxy: A Pluggable Java-Based Proxy* http://webtools.dyade.fr/pluxy/

[HLBF00] A.C. Huang, B. Ling, J. Barton, and A. Fox, *Appliance Data Services: Running the Web Backwards* Proc. WWW-9 . *code available at http://ads.stanford.edu*

[FSGM] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani et al. *Computing Iceberg computes efficiently* http://www-db.stanford.edu/ sergey/ddm.ps

[GBHC00] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David E. Culler *Scalable, Distributed Data Structures for Internet Service Construction* In Proc. OSDI 2000.

[BB00] Doug Beeferman and Adam Berger, *Agglomerative clustering of a search engine query log*, ACM KDD 2000 Proceedings

[ALEXA] *Alexa Inc.* http://www.alexa.com

[PNDGO] *The Pandango Search Engine* http://news.cnet.com/news/01-1005-200-4950537.html@stanford.edu

[MPF00] Laurence Melloul, Trevor Pering and Armando Fox *WeSCoS: A First Step Towards Web Programming* OPENSIG 2000 Workshop

Appendix A: Description of Database Schema

Table Describing User-ID infornation

Field	**Type**	**Key**
UID	int(11)	PRIMARY
Username	varchar(15)	UNIQUE
Password	varchar(8)	
FullName	varchar(40)	
Email	varchar(30)	

Table where URLs are logged

Field	**Type**	**Key**
UID	int(11)	PRIMARY
URL	varchar(255)	PRIMARY
HitCount	int(11)	
UserRank	int(11)	

Table Where Search Results are Logged

Field	**Type**	**Key**
userid	int(11)	PRIMARY
keyword	varchar(63)	PRIMARY
url	varchar(255)	PRIMARY
flags	varchar(7)	
hits	int(11)	