Vineyard: A Collaborative Filtering Service Platform in Distributed Environment

Toshio Oka Hiroyuki Morikawa
The University of Tokyo
School of Frontier Science
3-7-1, Hongo, Bunkyo, Tokyo, Japan
{oka,mori}@mlab.t.u-tokyo.ac.jp

Tomonori Aoyama
The University of Tokyo
School of Information Science and Technology
3-7-1, Hongo, Bunkyo, Tokyo, Japan
aoyama@mlab.t.u-tokyo.ac.jp

Abstract

As the amount of data shared on the Internet drastically increases, it becomes more important to utilize human feedbacks in retrieving information. While collaborative filtering of information is a promising approach to retrieve useful information, unfortunately there is an obstacle in the current collaborative filtering. Since a single system that serves for every type of object with optimal performance is unlikely, or at least likely to be too complicated as to implementation, it is more practical to use multiple systems each of which is dealing with a specific target. In many collaborative filtering systems, a system for book only deals with books, and another system for movie only deals with movies. The data stored in the former system is not accessible from the latter, and vice versa. This situation brings about inflexibility of collaborative filtering. We claim that this issue can be addressed by realizing multiple collaborative filtering systems on top of a single platform. Since all the profiles are shared on a single platform, they are accessible from all the systems. The filtering accuracy of our approach is identical with a certain kind of collaborative filtering system under a certain condition. We describe a design of collaborative filtering service platform in this paper. Our design of platform is fairly generalized, and it can be realized both in a centralized and peer-to-peer fashion.

1. Introduction

A large amount of data is shared on the Internet as the digital media becomes widespread. We cannot, however, fully utilize the useful information on the Internet since shared information is huge in its quantity. So, many researchers are trying to alleviate this troublesome situation. One of the promising approaches is collaborative filtering[1, 2, 4, 6, 7, 8, 9], which efficiently eliminates useless items by taking advantage of others' ratings. While content-based filtering that rely solely on feature vectors are faced with difficulties as to certain kinds of applications, collaborative filtering supports various applications, including net news, music, movies, web content, books, and so on. One of the typical approaches of collaborative filtering is to employ correlation of user ratings towards target items. In this approach, personalized ranking of items is computed by using the ratings of others who have strong correlations in profile with the user. We are dealing with this type of collaborative filtering approach in this paper.

We observe that dedicated collaborative filtering for each target field is desirable in order to improve the utility of the system, because filtered results may still contain lots of unnecessary items if the system handles excessively broad areas. For instance, a collaborative filtering system that doesn't distinguish the subjects of pictures cannot necessarily eliminate portraits, even if a user is looking for pictures of beautiful scenery. It is advantageous to restrict the objects by some criteria also in other types of collaborative filtering(e.g., movie, news mail, html). Besides, there is another benefit of limiting the objects in collaborative filtering. The system can be optimized for its application if it handles specific objects. For example, a system for web pages with RSS(RDF Site Summary) can provide additional functionalities as compared with a system for ordinary web pages.

However, we think that there is an apparent disadvantage that users can retrieve only a limited kinds of objects with the system. Imagine, for example, that a user wants to retrieve all sorts of items of a TV personality including movies, web pages, and books. In this case, he may find only books of the personality with a dedicated system for book. Similarly, people cannot find books about a movie with a system only for movie. Notice that contacting all the collaborative filtering systems for movie, web page, and book is not satisfying solution because they cannot fully uti-

lize the client profile. Although the client has a profile that is composed of movie, web page, and book of the TV personality, the collaborative filtering system for movie only utilizes the profile entries as to movies when computing correlation, because the system doesn't store the data for books nor web pages. Most entries of the client profile are ignored during the computation. Accordingly, there is certain amount of degradation in filtering accuracy. So, collaborative filtering is also required to broaden its target in a certain situation. We have directly-opposed requirements for collaborative filtering systems. Instead of accomplishing the two requirements with one system, we try to address this issue by realizing various collaborative filtering systems on top of a single platform. Since all the profiles are managed by a single platform, systems can share their profiles. As a consequence, each system can utilize user input efficiently, and can gather a large number of profiles instantly. Additionally, we can save the implementation and processing cost by reducing the overlapping functions. Specific functions based on user requirement can be realized by each system.

Our platform provides primitive functions for collaborative filtering that utilizes correlation of user profiles. In this paper, we describe how basic and extension functions should be decoupled in collaborative filtering especially in the context of distributed environment. Furthermore, we discuss the way how we realize the extension functions. We also describe the load balancing mechanism of the platform. Load balancing is a crucial matter since the platform is assumed to deal with various kinds of items that can be objects of collaborative filtering, including web pages, books, movies and etc.

The reminder of this paper is organized as follows. Section 2 describes the overview and the design of the Vineyard platform. In section 3, we discuss the detail of each component, especially load balancing one, and we mention the implementation of this platform in section 4. We conclude this paper in section 5.

2. System Design

Collaborative filtering aims at saving users' labor in finding useful items by filtering out irrelevant ones based on user profiles. To the best of our knowledge, the term, *collaborative filtering*, firstly appeared in [2]. According to [2], it simply means "people collaborate to help one another perform filtering" by recording their reactions to items they have already seen. In many collaborative filtering, ratings of items are predicted in advance before users actually see them, so users don't have to spend time for items with low predicted ratings. One of the typical approaches to predict ratings utilizes correlation of user profile that contains list of (item, rating) pairs(Table 1). In Table 1, each row indicates an user's preference profile for items. The numbers(5-1) in the table indicate user evaluation, *excellent(5)* to *awful(1)*. For example, user P likes item e, but takes item g to be useless. Predicted value x of item i is computed based on others' ratings, but each rating is weighted by the correlation of profile between the target user U and other users. The rating for i by user Q is the most dominant entry in this case.

	a	b	c	d	e	f	g	h	i
user P		3			5		1		1
user Q	5		4			5	1		1
user R	1	1		5				4	
user S			1		4			3	3
user T							5	5	
U	4	3	5			5	2		x

Table 1. Rating of each Item

An interesting topic is how we realize various collaborative filtering systems on a single platform that manages the entire user profiles. The platform must provides simple and common functions that are required in general collaborative filtering systems. We describe the design of the platform that is composed of 4 components. Rough sketch of these components are depicted in Figure 1.

Server components

CServComp(Correlation Server Component)
CServComp stores the list of (item, rating) pairs (Table 1) and finds users whose profiles have strong correlation with the client user.

PServComp(Profile Server Component)
PservComp stores user profiles and provides the profiles for ClientComp. The format of stored data is the RDF(Resource Description Framework) style.

CServComp and PServComp run on a number of hosts in order to achieve load balancing.

We will illustrate how CServComp works by an example of user U. CServComp has a table shown in Table 1. When a request of user U arrives, CServComp put her profile in its table. Next, it sends a list of users who have strong correlation with user U. Thus, CServComp keeps a rating table and shows a list of users who have strong correlation with the client. Note that CServComp doesn't show an end result.

Function of PServComp is also simple. PServComp just shows a profile of the user designated in the request(e.g., user Q) to its client. It also gives metadata of items in the user Q's profile along with a response. In order to avoid

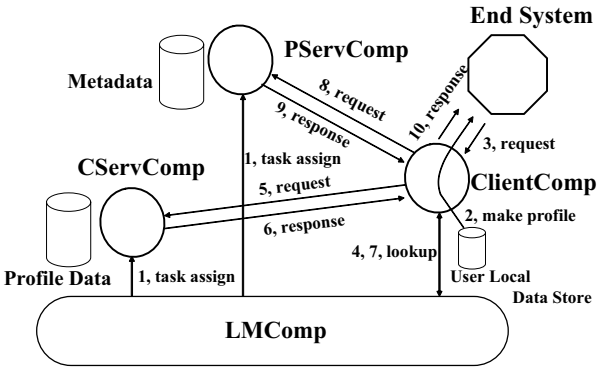

Figure 1. Overview

sending irrelevant items, items with low rating or items without metadata specified in the request are filtered. With these two server components, CServComp and PServComp, user U can obtain important information: users who have strong correlation with the client user U, their preference profiles, and metadata for items.

The both two components described above provide quite generic functionalities, and some may feel that the server components don't satisfy the original goal to realize a variety of collaborative filtering systems. One solution to get around this issue is to realize additional functions on the CServComp. Convenient functions(e.g., monthly access ranking) can be implemented as standard functions on CServComp. Although only a limited number of functions can be deployed as standards, it is still powerful approach. Another solution is to run programs downloaded from clients under the protection of Java security mechanism. Although this approach is costly in terms of network and processing resources, it can satisfy various kinds of user demands. Our main solution, however, is to realize the system specific functions on the client-side. This approach doesn't require modification to CServComp, nor downloading of programs from client. Since these three solutions are complimentary, users can use these solutions according to situations.

Client component

End system provides some functionality peculiar to each filtering system. As is depicted in Figure 1, end system is realized on the ClientComp, and it communicates with CServComp and PServComp via ClientComp. End system decides ranking of items based on user instructions and profiles provided by PServComp.

An important operation before the interaction with server components is to make a preference profile. Every client has a data store locally, and each system makes its own user profile that is extracted from the stored items, ratings and metadata. For example, the profile of book filtering system contains (book, rating) pairs selected from the store, and the profile of collaborative filtering system for show business may contain items related to TV personalities that are also selected from the same store. The profile is transmitted to the CServComp along with the request. It is noteworthy that the users specified in the response from the CServComp has similar profiles with the client profile. When the client sends a profile for books, a person who has similar taste with the client regarding movie is not necessarily in the response, because items as to movie are not counted in computation of correlation.

Load-balancing component

If our platform handles only a limited number of items, these three components stated above are suffice. The platform, however, is intended to manage a huge number of items that can be object of collaborative filtering, so load balancing among servers is a crucial matter. It is LMComp that provides a load balancing property in the platform. One of its important roles is to serve as a guide to direct the ClientComp to the appropriate servers to contact. The other role of this component is to assign each server its responsible task.

Although we discussed the roles of the components, it is still unclear how the components are involved in total procedures of collaborative filtering. So, the procedures are presented below:

a, LMComp, CServComp, and PServComp are already running

b, ClientComp is installed in every user host, and several filtering systems are installed as user needs arise

c, A user performs rating of her own items, adding attributes, and mapping between items and systems through system user interfaces

d, Others' ratings and item attributes have been registered in PServComp

e, The user specify target areas and make profile selected from the local data store

f, The end system hand the profile to the ClientComp along with a request

g, The ClientComp asks the LMComp the server to contact

h, The ClientComp send a request to CServComp and obtain a list of users who have strong correlation with the client

i, The ClientComp sends a list of the users(obtained in 2.) to PServComp along with a request, and get the information of items they have

	a	b	c	d	e	f	g	h	i
user Q	5		4			5	1		1
user R	1	1		5				4	
U	4	3	5			5	2		x
user P		3			5		1		1
user S			1		4			3	3
user T							5	5	

user Q	(a, 5)	(c, 4)	(f, 5)	(g, 1)	(i, 1)
user R	(a, 1)	(b, 1)	(d, 5)	(h, 4)	
user U	(a, 4)	(b, 3)	(c, 5)	(f, 5)	(g, 2)

Table 2. Profiles Stored in CServComp α

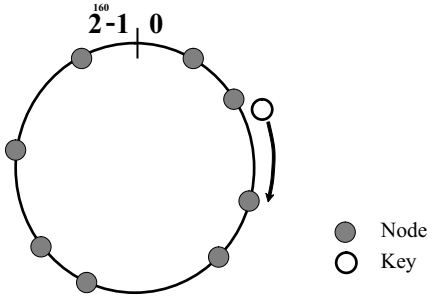

Figure 2. ID Space of Consistent Hashing

j, The ClientComp hand the information to the end system

k, The end system performs ranking of items

3. Operation

3.1. Load Balancing

Although we mentioned that LMComp performs load balancing of the platform, we didn't refer to the mechanism how it achieves the load balancing. First of all, we must consider how to distribute the requests among the servers. A naive solution is to use mirroring. Client can simply send queries to the least congested server because all the servers provide the same functions. No restriction, however, is imposed on the request in mirroring, so every server must have a complete set of data in order to cope with any kind of request. This approach is very costly when we deals with a huge amount of data. We must place some constraints on the request to the servers, so that the servers have only to store a part of the entire data. The key to address this problem is the required functions in the collaborative filtering servers(especially CServComp here). The CServComp returns the users who have strong correlation in terms of evaluation for the items specified in the request. Therefore, users who don't have these items at all won't be in the response, and their profiles are needless in resolving this query. Consider, for example, if all the queries to a certain server contain item a, then the server have only to manage the profiles with item a. Some may claim that people who have strong correlation with the user, but don't have item a won't be counted in this approach. In Table 2, the CServComp in a host α manages the profile of user Q and R, but don't manages the profile of user P, S, and T since they don't have item a in their profiles. So, the request by the user U cannot be fulfilled in the host α. This claim is rational if the user U can send request(s) only to a single host. The user U, however, can send requests to the host β, γ, δ that are respectively responsible for the item b, c, g to get around this problem. These hosts can compute the correlation with user P, S, and T.

Now, the load balancing mechanism performed in the LMComp is clear. The LMComp divides the entire item set into a number of small ones and assign the responsibility for a set to each server. For instance, the CServComp in the host α is responsible for the profiles that contain item a, and the CServComp in the host β is responsible for the profiles that contain item b. Note that you can use hash in order to keep the size of set uniform, and achieve the load balancing property. Consider itemID to be the key and (IP, Port) pair of the responsible server to be the value, then the hash table in the LMComp can tell which server is responsible for each itemID.

Although the interface of the hashing scheme is appropriate, there is serious disadvantage in the simple hash. The problem is that the table size of the simple hash is static. We have to reconstruct hash table, or rehash, if there is a big growth in number of users, thereby requiring installation of more servers. The cost of rehashing is prohibitive especially in the distributed environment because a large amount of data must be transferred in rearrangement. We adopt distributed hashing scheme in our Vineyard platform. We refer to consistent hashing here[3], which is one of the distributed hashing schemes. First of all, we explain how items with a key are mapped onto servers. In consistent hashing, we assume ring ID space(e.g., 160bit ID space) as illustrated in the Figure 2, and bucket(i.e., server here) is mapped to a randomly chosen point in the space. The item of the key is placed in the first node from the key in the clockwise direction. If the LMComp knows hostIDs and (IP, Port) pairs of all the hosts, it can solve which host is responsible for a key according to the key-bucket mapping rule. Under the condition where keys are uniformly distributed in the ID space, the number of keys a server is responsible for is roughly proportional to the length the server is responsible for. According to [3], the length each server is responsible for can be less than $(1 + \epsilon) \times \frac{L}{n}$ for arbitrary ϵ with high proba-

```
int GetRating(userIDu, itemIDi)
{
   if( (u, i) ϵ dataset ) return( r_{u,i} );
   else  return(DEFAULT);
}

double ComputeCorr(Profiletarget, Profilecprof)
{
   double corr=0;
   double cMean=cprof.mean;
   double tMean=target.GetMean();

   for(i=0, i<cprof.size, i++){
      itemID it=cprof.item[i].ID;
      int cRating=cprof.item[i].rating;
      int tRating=GetRating(target.userID, it);
      corr += (cRating-cMean)*(tRating-tMean);
   }
   return(corr);
}

void CResponse(Dataset dataset, Profile cprof, double thresh)
{
   Response response; double corr;
   for(i=0, i<dataset.size, i++){
      corr = ComputeCorr(dataset.profile[i], cprof);
      if(corr>thresh)
         response.insert(dataset.profile[i], corr);
   }
   SendResponse(response);
}

void OnRecvCRequest(Server serv, Requestrequest)
{
   Profile cprof = request.GetProfile();
   double threshold = request.threshold;

   Register(cprof);
   CResponse(serv.dataset, cprof, threshold);
}
```

Table 3. Pseudocode for CServComp

bility by using virtual servers(L:total length, n:the number of real servers). Therefore, we can achieve load balancing property with statistically assured performance. As it is clear from the Figure 2, the data transfer is little even if a bucket(i.e., server) is inserted/deleted. The transferred data during the node join/leave is the data transferred to/from the host that is going in and out, which is optimal in terms of quantity. Fault-tolerance and dynamic load balancing can be achieved in consistent hashing[5, 10].

3.2. Correlation Server Component

The outline of the CServComp is already discussed, so we are going to clarify the detail. CServComp returns the IDs of the users who have *strong* correlation with the client. Precisely, the correlation coefficient must be more than a certain threshold designated in the request. The correlation coefficients are sent along with the response. The pseudocode for the CServComp is shown in Table 3. Please note that the code is fairly abbreviated, and the language is quite different from existing one.

```
bool SatisfySpec(MetaDataTree mdTree, SpecTree spec)
{
   bool result; MetaDataTree childMD;
   SpecTree childSpec = spec.begin();//first child
   while(childSpec!=NULL){
      childMD = GetChild(mdTree, childSpec.avpair);
      //get child tree that satisfies attribute-value pair
      //Note that value can be "Don't care"

      if(childMD==NULL) return(FALSE);
      result = SatisfySpec(childMD, childSpec);
      if(!result) return(FALSE);
   }
   return(TRUE);
} // recursive version

void OnRecvPRequest(Server serv, Request request)
{
   ProfileSetprofSet = serv.profSet;
   SpecTreespec = request.GetSpec();
   Profileprof = profSet.GetProfile(request.userID);

   Response response;
   for(int i=0; i<prof.size(); i++){   //for each item
      MetaDataTree mdTree = prof.GetMDataTree(i);
      if( SatisfySpec(mdTree, spec) )
         response.insert( prof.GetItem(i) );
   }
   SendResponse(response);
}
```

Table 4. Pseudocode for PServComp

The client request is mainly composed of userID, and the list of (itemID, rating) pairs. When receiving a request, CServComp calls OnRecvCRequest function. After it registers the client's profile, it computes the correlation and sends a response to the client. In the pseudocode, itemID is a GUID(globally unique identifier) generated by hash function like SHA-1. Naming of userID depends on implementation, but must be unique on the platform. One alternative for userID is hashed value of the user host IP. This enables CServComp to avoid receiving massive registration from the same user. Another alternative is to use a number assigned by an entity(a person or an organization). The number must be signed by the entity, so that CServComp can verify that she is an authorized user.

Stored data on the server is maintained in soft state. After some time passes since it stores the data, the data is changed into cache. Client periodically renews the registration.

3.3. Profile Server Component

The pseudocode for the PServComp is shown in Table 4. When receiving a request, PServComp calls OnRecvPRequest function. The basic action of PServComp is to return user items and their information when it receives a query that specifies a user. Here, not all item data is sent to the client simply because all the data is not required in many cases. The items whose metadata satisfy the specification designated in the request are sent along the response. The processing of specification matching is shown in the Table 4 (SatisfySpec()).

Maintenance of data is performed by soft-state also in the PServComp. It is desirable that the interval until data is deleted is close to the interval in CServComp because it makes sense only when both data in CServComp and PServComp are available.

3.4. Client Component and End System

ClientComp provides common functions required for every end system. First, ClientComp functions as an interface to other components so that each end systems don't have to have their own implementation. Retrieved data from other components is cached in this component, and once the data is cached locally, client doesn't have to retrieve it from the distant servers. Second, it sends periodic registration message to the remote servers. If a certain end system is unused for a long time, it stops sending soft-state message. Third, it works as a data store for the end systems on it because storing user data in each end system respectively is inefficient. The format for the stored date is illustrated in the Table 5. When the end system wants to make the preference profile, it gives ClientComp a specification for the profile. For instance, the specification might contain "filetype=html" and "subject=wine" if the end system make a profile for wine.

3.5. End System

System-specific functions are implemented in the end system. Four main functions are enumerated below:

- To add metadata to items in the data store
- To extract items from the data store and make profiles
- To present the ranking of items
- To work with other applications

When items are registered with systems, the items are assigned attributes peculiar to the system. Further, additional attributes are added according to the system schemes. The attributes are composed of various entries including

```
<? xml=version="1.0" ?>
<rdf:RDF xmlsn:rdf="http://www.w3.org \
 /1999/02/22-rdf-syntax-ns#"
 xmlsn:vin="http://foo.bar.com/scheme/">
<rdf:Description about="urn:publicid:-:foo:sha-1:EN \
 :1234567890abcdef1234">
<vin:field>TV program</vin:field>
<vin:title>Full Home</vin:title>
<vin:genre>situation comedy</vin:genre>
<vin:director>Tom </vin:director>
<vin:star>
<vin:actor>Michael</vin:actor>
<vin:actoress>Lisa</vin:actoress>
</vin:star>
<vin:date>
<vin:year>2000</vin:year>
<vin:month>1</vin:month>
</vin:date>
<vin:filetype>wmv</vin:filetype>
</rdf:Description>
</rdf:RDF>
```

Table 5. Item Metadata

field, file type, subject, artist, author, actor, company, publisher, date, and what not. The example of the attributes is illustrated in Table 5. Since URN name space for SHA-1 has not yet been formally documented, we temporarily use "urn:publicid:-:foo:sha-1:EN" as the name space for the subject of the items.

Next, we refer to the topic as to making profiles. The number of queries sent to the servers is relevant to the way we make profiles. We will explain this by the notion of "view" in systems. Here, behavior of a system with a certain profile is referred to as view. If a system uses several profiles, then the system has several views. Each view periodically send queries to the servers and refresh the ranking of items. Consider, for example, a music collaborative filtering system that has two views, R&B view and classic view. Then the system has two profiles, and the host must typically send queries twice as much as the case only with one view. Let S be the number of systems in a user host, V_s be the number of views in the system s, and p_{ij} be a profile of the view v_j of the system s_i. The total number of queries the host must send to CServComp during a periodic interval(say, a month) is:

$$\sum_{i=0}^{S} \sum_{j=0}^{V_{s_i}} sizeof(p_{ij})$$

Although we can make a lot of views easily by selecting items from the local data store(ClientComp), we can have only a moderate number of views in a real sense because more network and computing resources are consumed with more system views.

The third role of the end system is to present the ranking of items. A typical approach for ranking is to calculate the average rating of others weighted by the correlation coefficient. But we don't define the methodology of this procedure not to lose the flexibility of the systems. The rating can be weighted by metadata like subject, date, artist and so forth.

Lastly, we mention the cooperation with other applications. An obvious example of the application is web browser. You can enter an item by drag-and-drop from the browser, or load html pages onto web browsers. Furthermore, book collaborative filtering system can check the recommended items in the online bookstore, and system for technical papers can check the author in digital libraries. Another example is to display thumbnail in a collaborative filtering system for picture.

3.6. Optional Components

Optional components are downloaded from the clients under the protection of Java security mechanism as already stated in section 2. It is similar to Java applet, but the downloaded code can access to the data of the Vineyard platform with read-only permission. The code cannot have an access to other data in the server host. Since it requires computational and network resource, the servers can refuse to serve this function when it is in a high workload. While this optional component seems to have much flexibility, it is uncertain if it really works in the real circumstances. This mechanism is left for the future work.

4. Implementation

Currently, we are implementing a prototype of Vineyard. We use DHT(Distributed Hash Table) in implementing LM-Comp instead of original consistent hashing. We adopt Chord as DHT, which belongs to consistent hashing family. Although the performance of Chord protocol greatly affects our platform, the details of Chord are beyond the scope of this paper. The platform of the prototype is written in Visual C++ on Windows OS. Current implementation is merely a prototype, so not all the functions described in this paper are implemented.

5. Conclusion

We designed a platform for collaborative filtering service in this paper. Unlike the normal monolithic collaborative filtering systems, we can easily realize fine-grained filtering services for the entire users on the platform. We studied required technologies to achieve the platform, especially about load balancing and the description of metadata. As we adopt a generalized algorithm(i.e., consistent hashing), we have two choices(centralized and peer-to-peer manner) to achieve the goal. If we deploy this platform on commodity computers, we may have to discard some data in order to avoid overload. Our current interest is the degree of accuracy degradation caused by the data discard. We must deploy this platform and clarify if the DHT approach appropriately works in a real environment.

References

[1] J. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the 14th Conference on Uncertainty in Articial Intelligence(UAI'98)*, pages 43–52, July 1998.

[2] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. In *Communications of the ACM*, December 1992.

[3] D. Karger, E. Lehman, T. Leighton, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC97)*, pages 654–663, May 1997.

[4] D. M. Nichols. Implicit rating and filtering. In *Proceedings of the 5th DELOS Workshop on Filtering and Collaborative Filtering*, pages 31–36, November 1997.

[5] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[6] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens : An open architecture for collaborative filtering of netnews. In *Proceedings of the Conference on Computer Supported Cooperative Work(CSCW'94)*, October 1994.

[7] J. Rucker and M. Polanco. Siteseer : Personalized navigation for the web. In *Communications of the ACM 40(3)*, pages 73–76, March 1997.

[8] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International World Wide Web Conference(WWW10)*, pages 285–295, May 2001.

[9] U. Shardanand and P. Maes. Social information filtering : Algorithms for automating 'word of mouth'. In *Proceedings of the Conference on Human Factors in Computing Systems(CHI'95)*, pages 210–217, May 1995.

[10] I. Stoica, R. Morris, M. Kaashoek, and H. Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication(SIGCOMM'01)*, pages 149–160, August 2001.