Flexible Caching in Peer-to-Peer Information Systems*

Pınar Yolum Munindar P. Singh

Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7535, USA
{pyolum, mpsingh}@eos.ncsu.edu

Abstract. We view the Internet as supporting a peer-to-peer information system whose components provide services to one another. We model service providers and consumers as autonomous agents. Agents may provide services or give referrals to one another to help find trustworthy services. Once found, some services may be cached. We describe a flexible caching technique that allows peers to operate autonomously (based on their local policies), accommodates heterogeneity of peers, and enables peers to adapt by choosing policies and neighbors as best suits them. In this approach, cache entries are coupled with metadata, thereby allowing the use of heuristics and flexible queries for more informed searches. The entries that are of interest to more agents are replicated at more peers, providing on-demand performance improvement and fault tolerance.

1 Introduction

Caching over the Internet has drawn a lot of attention and is of great practical value. However, currently caching is restricted to specific objects that are served over the net. Traditional approaches beg two main questions. How can we move from object requests to serving information needs of the various principals? How can we factor in the trustworthiness of information sources and of the caches?

We define *flexible caching* as caching that allows parties to proactively cache information that are relevant to them, from information sources that they trust. Our approach is based on two recent trends over the Internet. One, distributed computing is advancing from components (responding to remote procedure calls) to services (capable of extended interactions). Two, there is an increasing interest in peer-to-peer (P2P) systems. Current applications emphasize low-level aspects such as file sharing, but the fact that P2P systems are becoming practical bodes well for decentralized information systems architectures where the peers could provide arbitrary services to each other.

Over these, we layer our approach, which models service providers and consumers as autonomous agents, capable of developing a relationship of trust with one another. Our approach gives a central position to the notion of referrals among the agents representing different principals. Our philosophical claim is that referrals are essential for

* This research was supported by the National Science Foundation under grant ITR-0081742. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the National Science Foundation. We thank the anonymous reviewers for helpful comments.

locating and caching services in an open architecture. Referrals have been used in specific applications (see Section 5). However, we propose that referrals form the key organizing principle for large-scale information systems. Thinking about referrals from this perspective enables a new approach to caching in P2P networks.

Because of the openness of large-scale information systems, instead of looking for correct or relevant results, we look for authoritative sources who can provide correct and relevant results. There is an increased emphasis on *locating* trustworthy (which we take to include authoritative) resources, who are willing and able to provide the services needed. The authoritative sources could be the originators of the desired information or merely caches of it. We cannot expect to find trustworthy resources through traditional mechanisms for three obvious reasons. One, interesting resources may be invisible to traditional search techniques, thereby yielding low recall. Two, because important information and service needs are personalized, traditional indexing techniques simply lack the understanding and the context to produce the right results, thereby yielding low precision. Three, we cannot rely on regulatory restrictions for ensuring that the services offered are of a suitable quality or that the peers found over the network are trustworthy.

Organization. Section 2 introduces key elements of our model for agent-based P2P information systems. Section 3 discusses the design criteria that should be satisfied by flexible caching techniques. Section 4 describes Marmara, a caching technique that fulfills the criteria of Section 3. Section 5 discusses the relevant literature with respect to our work and motivates directions for further work.

2 Peer-to-Peer Information Systems

We now introduce our basic model. The main parties that interact or *principals* could be people or businesses. They offer varying levels of trustworthiness and are potentially interested in knowing if other principals are trustworthy. Our notion of services is broad; they can involve serving static pages, processing queries, or even carrying out transactions (although caching makes most sense for retrieving rather than modifying information).

The principals can track each other's trustworthiness and can give and receive *referrals* to services. Referrals are common in distributed systems, e.g., in the domain name system (DNS), but are usually given and followed in a rigid manner. By contrast, the referrals here are flexible—reminiscent of referrals in human dealings. Importantly, by giving and taking referrals, principals can help one another find trustworthy parties with whom to interact.

The principals are autonomous. That is, we do not require that a principal respond to another principal by providing a service or a referral. When they do respond, there are no guarantees about the quality of the service or the suitability of a referral. However, constraints on autonomy, e.g., due to dependencies and obligations for reciprocity, are easily incorporated. Likewise, we do not assume that any principal should necessarily be trusted by others: a principal unilaterally decides how to rate another principal.

Principals must be represented computationally. Because of the above properties of principals, they are ideally represented via *agents*. Agents are persistent computations

that can perceive, reason, act, and communicate [7]. Agents can represent different principals and mediate in their interactions. That is, principals are seen in the computational environment only through their agents. The agents can be thought of carrying out the book-keeping necessary for a principal to track its ratings of other principals. Moreover, the agents can interact with one another to help their principals find trustworthy principals. The above describes a particular kind of P2P system in which the peers are agents. The peers are thought of as agents here because they perceive, reason, act and communicate. Further, the peers are proactive, carry out interesting interaction protocols, and are capable of entering into relationships such as trust.

In abstract terms, the principals and agents act in accordance with the following protocol. When a principal desires a service, or when its agent anticipates the need for a service, the agent begins to look for a trustworthy provider for the specified service. The agent queries some other agents from among its *neighbors*, which are a small subset of the agent's acquaintances. A queried agent may offer its principal to perform the specified service or, based on its *referral policy*, may give referrals to agents of other principals. The querying agent may accept a service offer, if any, and may pursue referrals, if any.

Each agent maintains models of its acquaintances, which describe their *expertise* (i.e., the quality of the services they provide) and *sociability* (i.e., the quality of the referrals they provide). Both of these elements are adapted based on service ratings from the agent's principal. Using these models, an agent applies its *neighbor selection policy* to decide which of its acquaintances to keep as neighbors. Key factors include the quality of the service received from a given provider, and the resulting value that can be placed on a series of referrals that led to that provider. In other words, the referring agents are rated as well. The interests and expertise of the agents are represented as term vectors from the vector space model (VSM) [18], each term corresponding to a different domain.

In some settings, services can easily be cached. For example, consider a knowledge management setting where the idea for "consuming" knowledge services might be to acquire expertise in the given domain. When an agent asks a question, it gets an answer that can be duplicated. For example, if agent A learns about car history trivia from B, A can then answer the same question by just giving the answer B gave. Thus, in addition to generating answers to queries on demand, the answers can also be cached at other peers. Caching aids the search since a peer who is looking for an item can find it in a nearby cache rather than having the provider generate the item afresh.

3 Design Criteria for Flexible Caching

For information systems to work effectively, the caching techniques should satisfy the following criteria.

- *Peer Autonomy*. Autonomy of a peer is important in two respects. First, a peer should be allowed to choose who it will interact with, and how it will carry out its interactions. Second, it should be allowed to cache items that are useful to itself rather than caching items to serve others.

– *Peer Heterogeneity*. A peer can offer services and follow policies distinct from all others. The difference in policies may result in two peers providing the same service with varying qualities. Put another way, node heterogeneity constitutes a natural way to model varying trustworthiness of peers. By accommodating heterogeneity, an information system also accommodates the fact that the trustworthiness of the peers can be different.
– *Neighbor Choice*. Some of the peers in the system can be more useful than others, because of the type and quality of the services they provide. A peer should be able to modify its choice of neighbors so that the useful peers are contacted before others.
– *Item Metadata*. It is common to identify a data item by a unique identifier. In addition to the identifier, useful information about the item such as the timestamp of the item or the keywords related to it should be made explicit as metadata. This captured metadata can be utilized to form expressive search queries.
– *Search Criteria*. The simplest way to find a data item is through a unique identifier. In many real settings, the users do not have access to these identifiers, but rather information about the item that is sought. Thus, the caching techniques should allow more expressive searches to be formulated. This has two consequences. First, the items should be searched through metadata, like a list of keywords. Second, trade-offs between different subsets of the metadata should easily be captured.
– *Heuristics*. Ideally, data items should be located by contacting as few peers as possible. In order to achieve this, heuristics should be used to exploit the metadata of the items or the identities of the peers.

4 Marmara: Flexible Caching Technique

Marmara is our flexible caching technique. In Marmara, a cache consists of a set of entries that contain the data item as well as some accessing information about the data item. Unlike traditional P2P approaches, which model the data item through a unique global identifier only, we represent it with metadata. Each data item has fields for its name, author, last modification date, version, and a list of domains that it belongs to. If an agent knows the name of the item it is looking for, then it can search for the item with the name field. Mostly, a user would have a list of keywords in mind but not know the exact name of the data item. Our approach allows queries to be formulated with keywords that are matched against the list of domains listed in the metadata. In other words, keywords make up a query vector and the data item is the answer for the query.

The access data of an entry contains at least the timestamp of the last access to that data item. Depending on an application, further information about number of accesses to the item, or peers that access this information can be kept. The date for the last access is used to decide which item to remove from the cache when we run into a space limit.

Each agent can decide on the size of its cache. That is, an entry can stay in the cache as long as it is not manually deleted or automatically replaced by another entry based on the replacement algorithm in use. An entry can specify an expiration date for an item. This expiration date is not meant to be a rigid limit on the life of the item, but a signal to update the data item. A peer is free to use an expired item or find an updated version.

4.1 Insert

The peers are free to insert items into their caches at their own interest. In other words, there are no rules governing what items each peer can have. The insertion into a cache results from two different operations. First, the user may manually insert an item into the cache, thereby making it also accessible to other peers. Initially, we would expect all items to be inserted this way. Second, a peer may get the item from some other peer's cache, and decide to keep it, say, because it might be useful in the future.

Many peer-to-peer systems—Chord [20], content-addressable networks [14], Pastry [16]—divide the set of data items among the peers, such that each peer in the network becomes responsible of some of the data items, independent of the cache owner's interests. Thus, a peer may end up with items that it has no use for, and worse, the items it really needs may be residing on some other peer's cache. Our approach, on the other hand, does not require any agent to cache any items that they are not using. Each agent caches items that are of interest to its principal.

4.2 Delete

An agent can delete any of the entries in its cache for different reasons such as lack of space, lack of interest in the item, and so on. Just as each peer can delete its cached copy, the owner can also delete the original item. Even if the owner of an entry deletes the original copy, the peers that have a cached copy of the item are not forced to remove that entry from their caches. Thus, an entry may reside on two caches at one timepoint and later be deleted from one but still remain on the second one. This contrasts with Plaxton *et al.*'s [13] system, where a delete operation is circulated in the system, with the aim that each peer that receives the message will delete its own copy.

In addition to the manual delete, entries in the cache can also be automatically deleted to restore space in the cache. If there is no space left in the cache, when a new entry comes in, then one of the existing entries is replaced with the new entry. We employ the least recently used (LRU) replacement policy. However, we only consider uses by the principal. That is, the cache replaces the entry that has not been accessed for the longest time period by the agent that owns the cache. This ensures that only the entries that the principal is using are kept in the cache.

4.3 Query

An agent can formulate queries in different ways, describing and specifying constraints on the information sought. Different applications can utilize a combination of these queries based on the requirements of the application.

– *Keyword*. The agent supplies a list of keywords that are used to form a query vector. This gives flexibility to the search, in that the source peer does not need to know the exact name or the unique identifier of the data item it is looking for. The requested item is returned without any concern of the cost for the search or the version of the returned item.

– *Cost-constrained*. In addition to supplying a list of keywords, the agent also specifies how much it is willing to spend to find this item (assuming that there is an associated cost for searching the data item). This allows an agent to trade quality for cost. An agent that is looking for a data item, but is willing to spend only a little on it, can get an old version of the data item.
– *Version-constrained*. With the cost-constrained query, an agent cannot specify which versions of the item are acceptable. Here, though, it can specify the acceptable timestamp of the item. This is especially useful when there is no constraint on the cost—the agent is willing to spend anything, but wants to get at least a certain version of the item.
– *Cost-version-constrained*. This is a combination of the previous two queries. In addition to the keywords, both the acceptable cost and the version are specified. The same example, with a timestamp of 1 hour, for example, means that the agent is looking for the version of the data item that is at most one hour old and it is only willing to spend a certain amount. Obviously, a successful search should satisfy the cost constraint and the returned data item should satisfy the version constraint.
– *Exact*. If the user knows the unique identifier of the item it is looking for, then it does not need to supply any keywords. The query vector can then be modeled as the interest vector of the agent. Note that this vector will only be used in searching for peers in the graph, not for matching the answers. Thus, only answers that have the unique identifier are returned.

4.4 Search

An agent will start the search by sending a query to its neighbors. When the neighbor gets the query, it searches its own cache. If it does not have the item, it can either refer some of its neighbors or, if it is also interested in having the data item, it can start a new search for the data item itself, cache a copy, and return it to the original requesting agent as well. If the agent decides to refer, it uses its referring policy to choose among its neighbors. The referral policies take into account the expertise of the agents. One way to choose the neighbors is through the capability metric, which measures how sufficient the expertise vector is for a given query vector [19, 21]. Essentially, it resembles cosine similarity but it also takes into account the magnitude of the expertise vector. This means that expertise vectors with greater magnitude turn out to be more capable for the query vector. In Formula 1, Q ($< q_1 \dots q_n >$) refers to a query vector, E ($< e_1 \dots e_n >$) refers to an expertise vector and n is the number of dimensions these vectors have.

$$Q \otimes E = \frac{\sum_{t=1}^{n} (q_t e_t)}{\sqrt{n \sum_{t=1}^{n} {q_t}^2}} \tag{1}$$

Figure 1 shows an example network, where the nodes denote the agents and a dotted line from an agent A to an agent B means that B is a neighbor of A. The vectors marked with I are the actual interest vectors of the agents. The vectors marked as E_X denotes that agent's expertise model of X. Let's walk through an example. Agent A is looking for an item that can be cast into a query vector of [0.2, 0.8, 0.3]. The peers, D and F have the item in their caches. A starts the search by generating the query vector. Next,

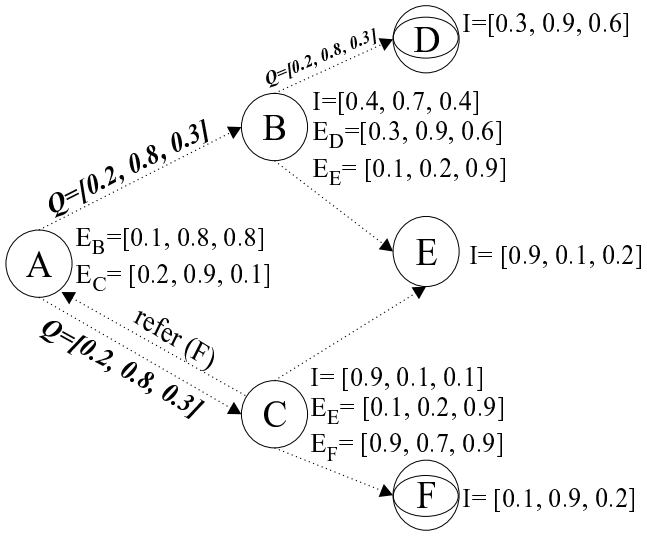

Fig. 1. An example search network

it decides which of its neighbors (B, C) to send the query to using (Equation 1). Even with a fairly high threshold, both B and C qualify. So, A sends its query to both B and C as indicated by the query vector on the links from A to B and A to C. When B gets the query, it checks its own cache to see if the item exists there. Not finding it in its own cache, it decides to send the query to its own neighbors since its interest vector is similar to the query. After it gets the item itself, it can forward it to A. Notice that among B's neighbors (D, E), D is the only one whose expertise vector matches to the query, so B sends the query only to D. After B receives the item, it puts the item into its cache and forwards it to A. Meanwhile, since C is not interested in the query, it just sends a referral back to A.

Notice that this search, which was instantiated with a keyword query, can yield multiple results. Some of the peers may have more than one item whose content matches the query vector. For each item, we can specify how good a match it is to the given query, which allows the returned items to be trivially ranked based on their apparent match. If different versions of the same item exist in different caches, how will the searching be different? Version-constrained and cost-version-constrained queries are used to search data items that can have different versions. Figure 2 shows another example network. Each edge is labeled with the cost variable that can represent the cost of contacting a peer.

At 1.20PM, A is looking for an item whose keywords map to the query vector of [0.2, 0.8, 0.3]. B, C, and F have versions of the item that are 60, 40, and 30 minutes old, respectively. The cost of sending a query to B, C, and F is c_1, c_4 and $c_4 + c_6$; respectively. Using a cost-version-constrained query, A can specify its preference of newer version versus less cost. In other words, A might be happy getting the oldest item from B and paying less. Or, if it is keen on getting a more recent version of the

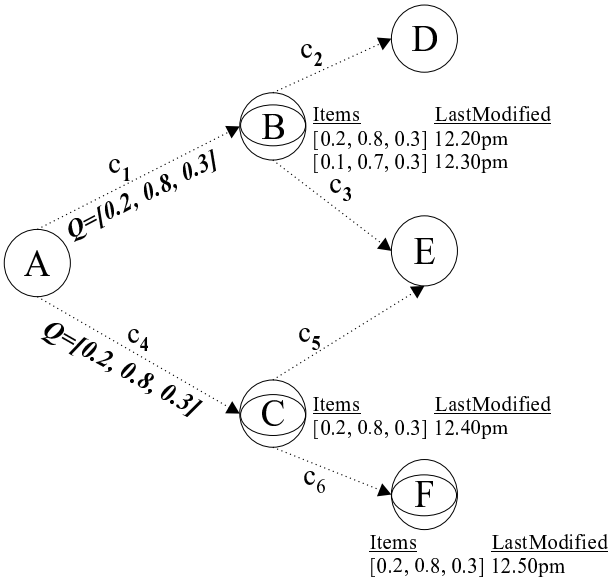

Fig. 2. A variable cost network

item, it can generate a version-constrained query to say that it is willing to pay (wait) whatever it takes to get an at most 30 minute old version of the item.

4.5 Update Models

Owners of data items can update them. Ideally, it is desirable to ensure that all caches that contain the particular entry are updated when one is updated. Heuristics based on circulating updates in the network turn out to be too costly, especially in distributed systems involving a large number of agents and a large number of items. For this reason, instead of maintaining strictly coherent caches, we let entries be timestamped by their modification date. This allows different versions of the same items to exist in different caches in the system. Obviously, when a cache gets more than one version of the same entry, it can get rid of the previous version and only save the most recent one. In addition, we propose several update models that will be useful depending on the importance of how recent the data item is.

Subscribe Under this model, when a peer requests a data item from a provider, it can subscribe for further updates from this provider. When the requester subscribes for the item, the provider is responsible for notifying the requester for any updates on that item. The requester can obtain the new version of the item whenever it receives an update notification. Alternatively, the requester can obtain the updated version of the item only when there is a request for the item (from its principal or from another peer).

This update model ensures that the requester will always have the up-to-date version of a data item with respect to a provider. In other words, if the provider is the actual

updater of the item, then the requester will always have the most recent version of the item. On the other hand, if the provider is not the actual updater of the item, it might not be getting the updated version of the item periodically. Thus, there may still be updates that the provider is missing, all of which will also be missed by the requester.

This leads to the identification of *authoritative* peers in the system. The peer that updates a data item is, by definition, an authority for the item. The peers that mirror the updates promptly can also serve as authorities for other peers. From a requester's point of view, there is no difference between the actual authority for a data item and these mirroring peers. Therefore, locating any one of these authoritative peers is enough to guarantee the retrieval of the most up-to-date data item.

Invalidate The subscription model is useful when a peer needs to have an up-to-date version at all times. On the other hand, a peer may need a data item, but not care for updates. In this model, the requester is sent a notification of only the first update and not any later updates. The requester would know that its copy was stale. It may look for an updated version when it needs the item, or (knowingly) use the invalidated copy. When an agent gets an answer to its query, it can request to get updates with one of the methods described. The service provider does not have to fulfill this request since not all service providers are required to support these update models.

5 Discussion

Referral networks are a natural way for people to go about seeking information [12]. One reason to believe that referral systems would be useful is that referrals capture the manner in which people normally help each other find trustworthy authorities. The importance of referrals to interpersonal relationships has long been known [5] as has their usefulness in marketing, essentially as a method for service location [3].

MINDS, based on the documents used by each user, was the earliest agent-based referral system [2]. Kautz *et al.* model social networks statically as graphs and study some properties of these graphs, e.g., how the accuracy of a referral to a specified individual relates to the distance of the referrer from that individual [10]. A more extensive literature survey about referral networks is available in [22].

Our referral-based approach takes an adaptive, agent-based stance on peer-to-peer computing. Below, we compare some related approaches based on the design criteria of Section 3. Piazza is a P2P system where each peer is a member of a sphere of cooperation and is assigned a role of data origin, storage provider, query evaluator, or query initiator [6]. Data items are placed to ensure that each peer finds the data it needs at a peer that is accessible with minimum cost. In our approach, each peer can autonomously decide on the data to store. Then, each peer becomes neighbors with other that are likely to carry the data items that it is interested in. Thus, each peer gets closer to the nodes where the items it is interested in resides.

In Piazza, when a peer receives a materialized view of a query, it pushes the view to its neighbors who then pushes it to their neighbors, and so on. Thus many peers —both interested and disinterested— receive the view but it also causes traffic. In our approach, we only push information to the peers that have shown interest in it by subscribing.

Similar to our approach, Piazza assumes that each data has an origin that is responsible for updates. The data in caches expire after a fixed period of time. This is useful when the expected expiration dates can be estimated. Our approach accommodates expiration dates like this, as well as explicit invalidation and subscription models. Piazza does not seem to keep metadata for data items and the type of queries allowed by the system are not discussed.

OceanStore is a global storage application that uses a peer-to-peer network [11, 15]. The network is highly controlled for the sake of performance optimizations. The number of times an item will be replicated, and the peers that will host these replicas are all controlled. This implies that nodes do not have autonomy since they cannot decide on which items to keep. OceanStore peers are not heterogeneous; they all operate in the same manner. The peers are allowed to change neighbors when a peer enters or leaves the system. OceanStore uses cluster recognition as a search heuristic. After each data access, a graph is constructed based on the semantic distance between the data items. When answering a query, the nodes locate and prefetch similar items with the idea that these items will probably be accessed as well. OceanStore can only be queried with the name or the globally unique identifier of the item.

PeerOLAP design an adaptive P2P system for caching online analytical processing queries [9]. Each peer decides on which items to cache autonomously based on its policies. Some of the policies consider the caching peer only while other policies take into account what the neighbors may be interested in and caches those items as well. Peers choose their neighbors based on how well they have answered previous questions. This is similar to our notion of neighbor selection policies and our model can easily accommodate a policy based on previous answers. For searching, a Gnutella-like protocol is used, where each peer sends queries to some of its neighbors who then either answer or forward them to their neighbors, for a given number of hops. There is an associated cost to search each node. The search tries to minimize the total cost of accessing an item. However, queries that specify a trade-off between versions and the cost of the item cannot be formulated.

Recently, several peer-to-peer network architectures have been proposed [20, 14, 16, 1]. These systems model the network as a distributed hash table where a deterministic protocol maps keys to peers. Thus, given the key of an item, there is one unique peer that is responsible for holding the key. Each peer in these systems has a table that aids the search when the item being sought does not reside at this peer. This is similar to our neighbors concept. Next, we discuss two storage systems that use one of these networks as an underlying substrate.

Cooperative File System (CFS) [4] is a file storage system that is built on top of Chord. Similar to OceanStore, replication of items are controlled by the system. Each file in the system is divided into blocks, which are automatically replicated on the k servers that follow the original server of the block. The nodes are not autonomous; the items that will be hosted by each node is controlled by the system. Each node is assumed to operate in the same manner, and there is no account for untrustworthy peers. Items do not have metadata associated with them. The search uses the Chord routing as an underlying layer. Hence, searches are actually a distributed hash table lookup, and no further heuristics are used in CFS. The items can only be searched by item keys.

Past [17] is a storage system that uses Pastry [16] for routing queries. Past is similar to CFS by design. Instead of replication blocks, files are replicated on k servers. Again, the peers do not decide on which items to store or what policies to follow. Again, there is no metadata of the items and the only way to search for a file is through its unique file identifier. The only search heuristic used exploits physical proximity of nodes. If a node can possibly forward a query to other nodes, it chooses the one that is physically closer.

	Peer Autonomy	Peer Heterogeneity	Neighbor Choice	Item Metadata	Search Heuristics	Expressive Search
Marmara	✓	✓	✓	✓	✓	✓
OceanStore	×	✓	×	×	✓	×
PeerOLAP	✓	✓	✓	×	✓	×
CFS	×	×	×	×	×	×
Past	×	×	×	×	×	×
Piazza	×	✓	×	×	×	×

Table 1. P2P approaches evaluated for flexible caching

Table 1 summarizes our evaluation of some P2P approaches with respect to the above design criteria. Please note that our purpose is not to show that existing approaches are inferior, just that they are unsuited to the specific criteria that we motivated. Our criteria are geared toward the higher-level aspects of caching in P2P information systems, whereas traditional caching approaches are geared toward caching at the object or file level.

Directions. The Marmara prototype is being implemented. In future work, we will evaluate the performance of Marmara, especially with respect to various policies and load distributions. We will also model richer properties underlying the connectivity among the peers, e.g., communication cost and available bandwidth. These properties affect how efficiently and effectively peers can be located. The overall idea is to understand the trade-offs between performance on the one hand and higher-level descriptions and local controls on the other.

References

1. Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *Proceedings of Cooperative Information Systems (CoopIS)*, pages 179–194, 2001.
2. Ronald Bonnell, Michael Huhns, Larry Stephens, and Uttam Mukhopadhyay. MINDS: Multiple intelligent node document servers. In *Proceedings of the 1st IEEE International Conference on Office Automation*, pages 125–136, 1984.
3. Jacqueline J. Brown and Peter H. Reingen. Social ties and word-of-mouth referral behavior. *Journal of Consumer Research*, 14:350–362, 1987.
4. Frank Dabrek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, pages 202–215, 2001.

5. Noah E. Friedkin. Information flow through strong and weak ties in intraorganizational social network. *Social Networks*, 3:273–285, 1982.
6. Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What can databases do for peer-to-peer? In *Proceedings of the Workshop on the Web and Databases (WebDB)*, 2001.
7. Michael N. Huhns and Munindar P. Singh. Agents and multiagent systems: Themes, approaches, and challenges. In *[8]*, chapter 1, pages 1–23. 1998.
8. Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1998.
9. Panos Kalnis, Wee Siong Ng, Beng Chin Ooi, Dimitris Papadias, and Kian Lee Tan. An adaptive peer-to-peer network for distributed caching of OLAP results. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 2002. To appear.
10. Henry Kautz, Bart Selman, and Mehul Shah. ReferralWeb: Combining social networks and collaborative filtering. *Communications of the ACM*, 40(3):63–65, March 1997.
11. John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, November 2000.
12. Bonnie A. Nardi, Steve Whittaker, and Heinrich Schwarz. It's not what you know, it's who you know: work in the information age. *First Monday*, 5(5), May 2000.
13. C. Greg Plaxton, Rajmohan Rajaraman, and Adrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 311–320, 1997.
14. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172, 2001.
15. Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, pages 40–49, September-October 2001.
16. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.
17. Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, pages 188–201, Banff, Canada, October 2001.
18. Gerard Salton and Michael J. McGill. *An Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.
19. Munindar P. Singh, Bin Yu, and Mahadevan Venkatraman. Community-based service location. *Communications of the ACM*, 44(4):49–54, April 2001.
20. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 149–160. ACM, 2001.
21. Pınar Yolum and Munindar P. Singh. Locating trustworthy services. In *Proceedings of the First International Workshop on Agents and Peer-to-Peer Computing (AP2PC)*, 2002. To appear.
22. Bin Yu. *Emergence and Evolution of Agent-based Referral Networks*. PhD thesis, Department of Computer Science, North Carolina State University, 2001.