Tuxedo: A Peer-to-Peer Caching System

Weisong Shi, Kandarp Shah, Yonggen Mao, and Vipin Chaudhary
Department of Computer Science
Wayne State University
{*weisong,kandarp,ygmao,vipin*}*@wayne.edu*

Abstract

We are witnessing two trends in Web content access: (a) increasing amounts of dynamic and personalized Web content, and (b) a significant growth in "on-the-move" access using various mobile resource-constrained devices by nomadic users. Web caching and the content distribution network (CDN) are popular solutions for improving Web access latency and have the effect of moving content closer to the client. However, these solutions typically do not work well with dynamically generated and personalized content. Transcoding *is a popular solution to resolve server-client mismatches (device heterogeneity), but is unable to benefit from caching in general.*

These trends necessitate revisiting the traditional Web caching and CDN approaches. In this paper, we proposed Tuxedo, a peer-to-peer caching system, that complements to existing hierarchical-based Web caching for efficient delivery of Web content and value-added edge services. Tuxedo allows multiple caches (peers) to efficiently share not only original Web documents, but also computing resources for transcoding (by sharing transcoded versions) and other value-added edge services. The novelty of Tuxedo includes an adaptive neighbor set algorithm *for different Web servers, and a* hierarchical cache digest *for sharing of transcoded versions and value-added services. Together, these two protocols contribute to the scalable and decentralized features of the Tuxedo system.*

1 Introduction

Peer-to-peer systems have increasingly become a hot research topic [15, 17, 21, 28]. However, in the context of peer-to-peer (P2P) computing, most of the research projects focused on routing algorithms and efficient query search mechanisms. Currently, major applications of peer-to-peer systems are music file sharing [7, 9, 14] and instant messengers [13, 27]. Although the technology is promising, we still face the barrier of lack of applications [11]. In this paper, we argue that peer-to-peer caching system is a very interesting application, especially with the following two trends of Web content access: (a) increasing amounts of dynamic and personalized Web content, and (b) a significant growth in "on-the-move" access using various mobile resource-constrained devices.

These trends point to a situation where a user would have ubiquitous access to content, but require that content be efficiently delivered to the user irrespective of location, and in a form most suited to the user's end device. Web caching and the content distribution network (CDN) are two popular solutions for improving Web access latency and have the effect of moving content closer to the client. However, in the last five years, we have witnessed the fast growth of dynamic and personalized Web content [29], observing that both Web caching and CDN typically do not work well with those content [5, 10, 24]. *Transcoding* is a popular solution to resolve server-client mismatches (device heterogeneous), but is unable to benefit from caching in general.

Object composition approach [19, 23] proposed recently is very promising in handling the first trend, which observes that despite multiple requests for the same site resulting in different content at document granularity, there exists substantial opportunity for reuse at the sub-document level (at the granularity of individual objects making up the overall document). Transcoding and applying other value-added edge services at the proxy cache to suit the client's end device based on user preferences and user access patterns are efficient techniques to address the second trend, as exploited by CONCA-like proxy caches [12, 19]. Although some pessimistic observations about cooperating caching were demonstrated in Wolman et al.'s work [24], in this paper we argue that these solutions for the two trends together hold the potential of both document sharing (e.g., document template and shareable objects) and computing resource sharing

(e.g., trascoded versions and other services) among multiple caches (peers).

In this paper, we proposed Tuxedo, a peer-to-peer cache system, as an alternative to existing hierarchical-based Web caching for efficient delivery of Web content and value-added edge services. The novelty of Tuxedo includes an *adaptive neighborhood set algorithm* for different Web servers, and a *hierarchical cache digest* for sharing of transcoded versions and value-added services. In comparison to Napster's centralized directory servers [14] and Gnutella's massive message flooding [7], the approach adopted in the Tuxedo is more scalable and completely decentralized.

The remainder of the paper is organized as follows. Section 2 describes briefly the CONCA proxy cache, which is the building block of the Tuxedo system. The design and scalability analysis of Tuxedo architecture is presented in Section 3. Section 4 discusses the related work in this area and finally the current status and several challenging issues are listed in Section 5.

2 Background

2.1 End-to-End is Not Enough

According to the end-to-end arguments [18], most of the intelligence has been deployed at the end systems on the Internet. However, the rapid proliferation of Internet users and increasing web traffic have led to a lot of load on the origin servers and thereby led the content-providers to adopt techniques that disseminate the load on origin servers. The deployment of caching proxies and surrogates makes the first step by moving content to the edge of the network. The rising demand for Internet services induces the idea of using existing caching proxies for more than simply accelerating the delivery of Web content. They seem to provide a viable location to deploy additional services. This implies a change in the current Internet model where the client and the server are the two end-points of communication and the introduction of "intelligent" networks where intermediaries could process certain requests and responses [2]. This suggests that the Internet will no longer be a mere data transfer network, more and more functionalities can be injected into the network along the data path, ranging from the network-layer, such as active networks [22], to application-layer, such as CANS infrastructure [6].

2.2 CONCA Proxy Cache

CONCA (**CO**nsistent **N**omadic **C**ontent **A**ccess) [19] is a proposed edge architecture for the efficient caching and delivery of dynamic and personalized content to users who access the content by using diverse devices and connection technologies. CONCA attempts to exploit reuse at the granularity of individual objects making up a document, improving user experience by combining caching, prefetching, and transcoding operations as appropriate.

To achieve its goals, CONCA relies on additional information from both servers and users. All content supplied by servers in CONCA architecture is assumed to be associated with a "document template" which can be expressed by formatting languages such as XSL-FO [26] or edge-side include (ESI) [23]. Given this information, CONCA node can efficiently cache dynamic and personalized content by storing quasi-static document templates and reusing sharable objects among multiple users. Moreover, based on the preference information provided by users, a CONCA cache node delivers the same content to different users in a variety of formats using transcoding and reformatting.

Figure 1 shows the logical organization of cache storage, which consists of two separate portions: *shared* and *personalized*. The shared portion contains static content (e.g., objects associated with the URL `www.nyu.edu`) and the sharable subset of dynamic content (e.g., the TV list channel associated with the URL `my.yahoo.com`). The personalized portion stores the per-user state, both for "home" users as well as other (nomadic) users who are temporarily using this cache. This consists of (*a*) the user's personal assistant, which contains information about the user's profile, devices, and transcoding preferences, (*b*) downloaded personalized objects, and (*c*) (intermediate) transcoded versions of these objects. Note that although the figure shows transcoded versions of all objects as being stored in per-user storage, in general we may be able to share intermediate transcoded versions of the shared objects. In response to a user request, the cache first looks up the personal assistant associated with the user to determine the objects of interest, then acquires the missing objects, transcodes them as required, and finally delivers to the user a document assembled from the various pieces.

The CONCA proxy cache is the building block of the proposed Tuxedo caching system, augmenting with functionalities such as neighbor discovery and resource sharing among adaptive number of neighbors.

3 Architecture

Different from the hierarchical-based proxy caching architecture, Tuxedo is an overlay network which charac-

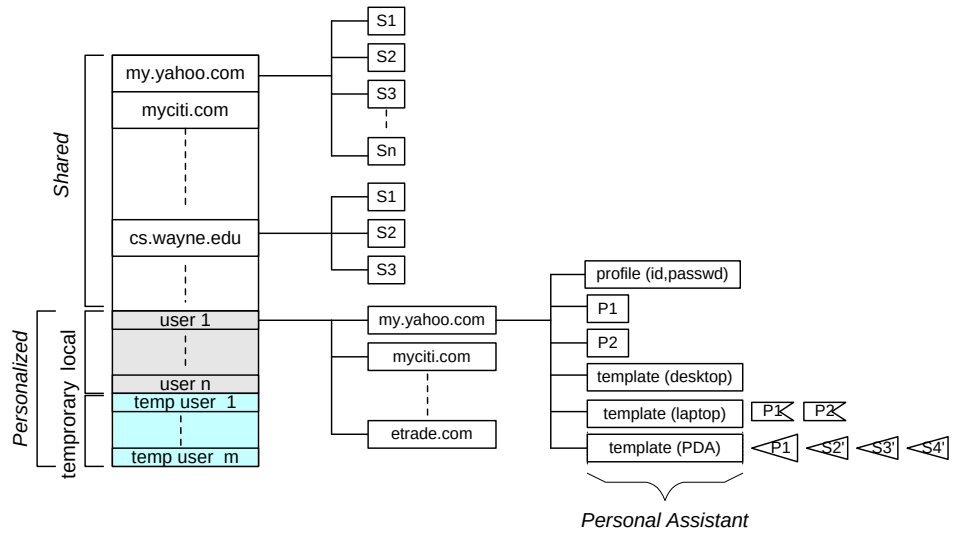

Figure 1: Detailed structure of a CONCA cache node, showing shared and personalized portions of the cache.

terizes itself by using efficient neighborhood propagation and adaptive neighborhood mechanism for different Web servers, and by using a hierarchical cache digest to store information related to transcoded content or results of other valuable edge services. Also, Tuxedo employs the notion of *reputation* to deal with security and trust concerns among peers.

3.1 Neighborhood Propagation

To ensure that neighborhood propagation is scalable, our approach includes the following four steps:

Step1: each peer (CONCA proxy cache) configures it's upper-level cache as it's neighbor.

Step2: each time when a peer makes request to its neighbor node and when the neighbor node sends response, a neighbor table (information related to peers address, latency, bandwidth, transcode services, etc.) is piggybacked along with HTTP request/response header. Note that some optimizations, e.g., diffing, are used to reduce the network traffic.

Step3: after receiving the neighbor table from other peers, the local Tuxedo cache filters out neighbor information from HTTP header and inserts it in local neighbor table according to some performance metrics, such as latency or bandwidth.

Step4: periodically each peer sends request to these neighbors to obtain accurate values for piggy-backed neighbor data and update neighbor table with fresh values. If the latency (or bandwidth) between the local cache and neighbor is too long (or small for bandwidth), local cache will remove this neighbor from it's neighbor table.

As shown in right part of Figure 2, each tuxedo node maintains a neighbor table which keeps the information regarding peer id (IP address) and a pointer to the peer information. Table 1 lists an example of all information that a peer cache maintained, including latency, bandwidth, availability of transcoded versions and results of other services (e.g. language translated file is available or not), etc.

Since each node (peer) maintains its own neighbor table and receives query only once, it saves the network from flooding effect. However, based on the information collected through piggyback technique only, it is hard to decide the value of latency/bandwidth between these two new peers, and this is one of the challenging issues we are focusing currently. In our initial approach, the latency between the local cache and a new added neighbor is calculated as the sum of two latencies (upper limit) as explained in the following example. Consider a scenario where A is neighbor of B with latency L_{AB}, and C is neighbor of B with latency L_{BC}. Now B tries to piggyback C's information to A. While inserting C's information in A's neighbor table, the latency between A and C can be calculated by: $L_{AC} = L_{AB} + L_{BC}$. For the bandwidth, we take the minimum as $B_{AC} = min(B_{AB}, B_{BC})$. However, when we update these peer information (executing step 4), we can re-sort out the elements of neighbor table with fresh values.

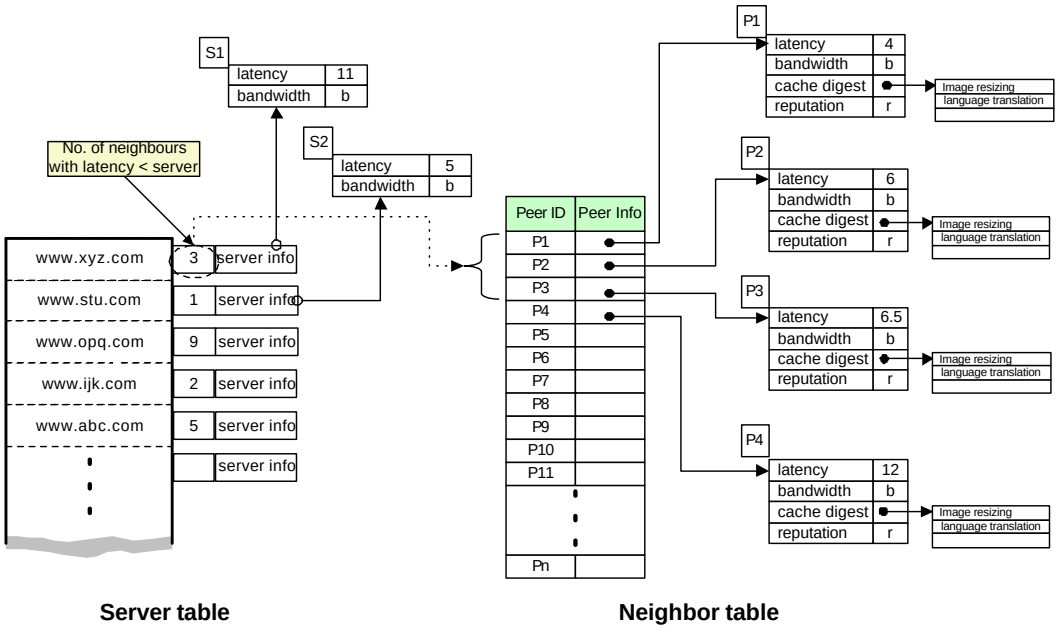

Figure 2: Two tables maintained at each Tuxedo cache node: server table and neighbor table.

PeerAddress	Latency (second)	Bandwidth (MBps)	ImageResize	LanguageTranslation	Reputation
141.217.16.181	4.00	100.00	true	false	0.98
141.217.16.182	6.00	110.25	true	true	0.72
141.217.16.183	6.50	124.35	false	true	0.83
141.217.16.184	11.00	97.62	false	true	0.98

Table 1: An example of 4 neighborhood information maintained at one Tuxedo cache node.

3.2 Adaptive Neighbor Set

One of the challenging problems in peer-to-peer infrastructure is to find neighbors (peers) and maintain their information. In addition to the neighbor table as described in last section, each Tuxedo node also maintains server table for different Web server, as shown in the left part of the Figure 2. Server table keeps two-piece of information: (1) a pointer (integer number) to the neighbor table and (2) original server information (latency/bandwidth). The pointer associated with each Web server depicts the specific neighbor set of this Web server. For example, as shown in Figure 2, the pointer of web server `www.xyz.com` is 3, which points out that it's neighbor set consists of the first three neighbors (P1,P2,P3). These neighbors are chosen because their latencies to this cache node are less than that of original server latency (11 sec in this case). Later, the local cache can make a request to any of these neighbors to get the requested content. Each Tuxedo node maintains the update of neighbor table information by periodically polling or piggyback updating from these neighbors. A similar approach can be used to update the server table information. In our current design, we maintain two ordered neighbor tables that indexed by latency and bandwidth respectively.

Consider a scenario where proxy cache `A` receives a request from a client to get Web content from `www.xyz.com`. `A` can then contact one of his neighbor P_i ($i \in [1, m]$), where P_i denotes the ith neighbor in the neighbor set, m is the number of his neighbor set. If the size of the required content is small (latency is dominant), cache `A` will select/prefer a neighbor which has minimum latency. From Figure 2, cache `A` will choose P_1. Consider the other scenario in which the requested content is a large media file (bandwidth is dominant), in this case, `A` might choose/prefer a neighbor that has maximum bandwidth available.

To generalize this, let us assume the file size is S, and X is a configurable threshold (intermediate size to decide whether entries in the neighbor table should sort in terms of latency or bandwidth). Using L_{p_i} and L_{s_j} to denote latencies between local cache and peer i and Web server j, B_{p_i} and B_{s_j} to denote bandwidth between local cache

and peer i and Web server j, respectively, the Web site j's neighbor vector V_{s_j} can be defined as,

$$V_{s_j} = \begin{cases} \{P_i | \text{where } L_{p_i} < L_{s_j}; i \in [1,n]\}, & S < X; \\ \{P_i | \text{where } B_{p_i} > B_{s_j}; i \in [1,n]\}, & S \geq X \end{cases}$$

By using the above technique we can choose a neighbor vector for each Web server dynamically/adaptively. One can consider latency for normal Web sites, and switch over to choosing high bandwidth neighbors for media intensive sites where large bandwidth is preferred.

Note that our adaptive neighbor set approach is different from the approach used in Squirrel caching [8], which was build on a distributed hash table and did not take the bandwidth or latency into consideration for different Web sites. Our initial results show that our approach is more scalable and efficient than their approach.

In peer-to-peer networks, peers are join and leave frequently. So it is important to keep track of this activity. To designed proficient caching system we believe that reputation factor should be considered. Tuxedo counts reputation for each peer in the following manner. When peer A contact any peer located in his/her local neighbor table, and that node is not live (or not respond) can be counted as false hit. Such node's (peer's) reputation should be decrease by some x percentage as a penalty and we can update the such information in local neighbor table. So next time peer A can take reputation factor into consideration while requesting the content from such malicious peer and able to find other trustworthy peers.

3.3 Transcoded Content and Edge Services

As an extended version of CONCA, Tuxedo supports computing resource sharing by using transcoding and other edge services at other peers. Currently Tuxedo adopts a simple approach to decide whether cached copy of transcoded content should be accessed from neighbors or it should perform transcoding locally. Let's consider $T_{notrans}$ as computation time to get a cached copy (not transcoded), T_{trans} as computation time to fetch a cached copy (transcoded version), then such values can be calculated as: $T_{notrans} = \sum_{i=1}^{h_1} L_i + T_{conv}$ and $T_{trans} = \sum_{j=1}^{h_2} L_j$, where h_1 equals the number of hops for original copy, h_2 equals the number of hops for transcoded copy, L_i is the latency for hop i, and T_{conv} is time required to convert content specific to user's end device locally. Based on the value of $T_{notrans}$ and T_{trans}, one can establish a connection with the preferred peer. We are also investigating efficient techniques to select a peer that deal with different scenarios by considering latency, bandwidth, valuable edge services, reputation, and computation time.

We propose to use a *hierarchical cache digest* approach to maintain consistency and share content among multiple peer caches. Our approach includes two parts. First, each peer maintains a digest that depicts what kind of services it supports in addition to original content. We plan to use a bit vector method [3] to store this information. Second, a cache digest is maintained for each service supported by this cache, using the Bloom filter algorithm [1]. The algorithm has been successfully used in cache digest algorithms proposed in [4, 16]. For each service supported by the cache, the corresponding digest stores the information about which transcoded content (by applying this specific service) is available in this cache. As such, this will support the reuse of transcoded versions resulting from different valuable edge services. Furthermore, based on these digests, a *lease-based* approach can be used to maintain consistency among remote replica and origins. The right portion of Figure 2 shows a example, where all peers support image resizing service and language translation service, and the digests for the availability of resized images and translated content are stored separately in the table. As shown in the Table 1, it sets the flag `true` or `false` depending upon the availability of such service. For example, from the table we see peer `141.217.16.181` has provided a image resize service, since flag is set to `true`.

3.4 Scalability Analysis

We argue that the proposed peer-to-peer caching architecture, Tuxedo, is highly scalable. It helps achieve scalability in many ways. First, by storing the peer's information locally it reduces the amount of search requests that the user needs to send out in order to locate Web content, which leads to reduction in response time (latency) as well as diminish the network traffic. Second, adaptive neighborhood is a scalable way to solve the increasing number of neighbors, and it is a completely decentralized approach. Moreover, Tuxedo supports asymmetric relationship between peers, for example, Peer A can be a neighbor of Peer B, while Peer B may not be a neighbor of Peer A. Third, the memory space requirement for our approach is small enough to keep all of data structure in memory, which augments the scalability of system. Using Table 1 as an example, it stores information related to peer id, latency, available bandwidth, availability of image resize service and language translation service, and

reputation percentage of peers. We observed that to store such information for 4 peers it requires around 1 KB of memory space. So, we can store 1000 neighbors information by utilizing 1 MB of memory space. Also, the memory space required by the server table is also very small (less than 100 bytes), so Tuxedo can support more than one million Web sites (100 MB) very easily. Finally, the piggy-back based information update protocol also enhances the scalability of Tuxedo.

4 Related Work

Our work on Tuxedo builds upon a large body of related work in the general area of Web caching and peer-to peer system. Here we will discuss only two recent efforts that share similar goals and approaches.

Backslash [20] is a content distribution system based on peer-to-peer overlay and used for those who do not expect consistently heavy traffic (flash crowds) to their sites. While Tuxedo focuses on the scalability and resource sharing among peer-to-peer proxy caches, leveraging the browser caches on the client machine to form a peer-to-peer cache , such as Squirrel [8] and Browser-Aware Proxy Server [25], is another approach to improve scalability and performance. However, in their implementation, the proxy server maintains the index file of data objects of all clients' browser caches. So, it's not a totally decentralized concept and may lead to a single entry point failure. We are maintaining such an index file on each peer's machine so that they can directly locate other peers who already have these data objects in their cache. Moreover, both Backslash and Browser-Aware Proxy Server do not provide mechanisms for transcoding or any other valuable edge services.

5 Current Status and Future Work

5.1 Current Status

At present, we have done a simulation-based proof-of-concept of Tuxedo system, including neighbor propagation and adaptive neighbor set algorithm. We plan to integrate it into CONCA proxy cache, which is under development at Wayne State University.

5.2 Future Work

During the simulation of Tuxedo protocols, we encountered several challenging issues that will be the focus of our future efforts.

- **Efficient mechanism to update server and neighbor information:** Currently we are updating information related to peers' latency, bandwidth, etc., by sending requests periodically (batch jobs) or by the piggy-back technique. But this is not efficient for dynamic updates of information. For example, information retrieved by piggybacked approach do not reflect the actual value (bandwidth, latency), e.g., triangle problem described in section 3.1. To investigate a proficient method to get such information is a challenging research issue.
- **Supporting multiple servers:** The adaptive neighbor set algorithm works fine for the Web site which has only one centralized server. However, with the prevalence of distributed server farms, application server providers, and content distribution network, the latency and bandwidth from one web site may experience different values at different time, which makes the server table in Tuxedo cache node unstable. Handling this issue is another aspect of our future work.
- **Optimization of computation and communication:** Transcoding operations are often time consuming, therefore the cache node may sometimes have to decide between (re)transcoding an object locally and fetching the transcoded object from a neighbor cache, even when the original version is available locally. Similar issues crop up in trying to decide between fetching content from a cache node connected with a slow link that already has the content in transcoded form, versus fetching it from a cache node reachable via a faster link but where one has to spend time transcoding the content. More generally, it is possible to treat the fetch and transcode operations as components along the path, each with their cost and effect on response time or throughput. Therefore, the tradeoff between computation and communication becomes an optimization problem, with the objective being minimization of response time or maximization of throughput.

References

[1] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13(7):422–426, July 1970.

[2] M. S. Blumenthal and D. D. Clark. Rethinking the design of the internet: The end-to-end arguments vs. the brave new world. *ACM Transactions on Internet Technology* 1(1), Aug. 2001.

[3] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, Inc., 1998, chapter 12.

[4] L. Fan, P. Cao, J. Almeida, and A. Border. Summary cache: A scalable wide-area web cache sharing protocol. *Proceedings of ACM SIGCOMM'98*, pp. 254-265, Mar. 1998.

[5] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. *Proc. of IEEE Conference on Computer Communications (INFOCOM'99)*, pp. 107-116, Mar. 1999, `http://www.douglis.org/fred/work/papers/hetproxcache.pdf`.

[6] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01)*, pp. 135-146, Mar. 2001.

[7] Gnutella, `http://gnutella.wego.com`.

[8] S. Iyer, A. Rowstron, and P. Druschel. SQUIRREL: A decentralized, peer-to-peer web cache. *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, July 2002.

[9] KaZaA, `http://www.kazaa.com`.

[10] B. Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. *Proceedings of SIGCOMM IMW 2001*, pp. 169-182, Nov. 2001, `http://www.research.att.com/~bala/papers/imw01-abcd.pdf`.

[11] J. Ledlie, J. Shneidman, M. Seltzer, and J. Huth. Scooped, again. *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Feb. 2003.

[12] V. Mastoli, V. Desai, and W. Shi. SEE: a service execution environment for edge services. *Proceedings of the 3rd IEEE Workshop on Internet Applications (WIAPP'03)*, June 2003.

[13] MSN Messenger, `http://messenger.msn.com`.

[14] Napster, `http://www.napster.com`.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content addressable network. *Proc. of ACM SIGCOMM'01*, 2001.

[16] A. Rousskov and D. Wessels. Cache digest. *Proc. of 3rd International WWW Caching Workshop*, June 1998.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. *IFIP/ACM Middleware 2001*, 2001.

[18] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems* 2(4), Nov. 1984.

[19] W. Shi and V. Karamcheti. CONCA: An architecture for consistent nomadic content access. *Workshop on Cache, Coherence, and Consistency(WC3'01)*, June 2001.

[20] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Feb. 2002.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM'2001*, 2001.

[22] D. Tennenhouse and D. Wetherall. Towards an Active Network Architecture. *Computer Communications Review* 26(2), Apr. 1996, `http://www.tns.lcs.mit.edu/publications/ccr96.html`.

[23] M. Tsimelzon, B. Weihl, and L. Jacobs. ESI language sepcification 1.0, 2000, `http://www.esi.org`.

[24] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. *Proc. of 17th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 16-31, Dec. 1999.

[25] L. Xiao, X. Zhang, and Z. Xu. On reliable and scalable peer-to-peer web document sharing. *Proceedings of 2002 International Parallel and Distributed Processing Symposium*, Apr. 2002.

[26] W3C XSL Working Group, `http://www.w3.org/Style/XSL/`.

[27] Yahoo! Messenger, `http://messenger.yahoo.com`.

[28] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry:an infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, UC Berkeley, Apr. 2001.

[29] Z. Zhu, Y. Mao, and W. Shi. Workload characterization of uncacheable web content — and its implications for caching. Tech. Rep. CS-MIST-TR-2003-003, Department of Computer Science, Wayne State University, May 2003.