CACHING NEWS CHANNELS ON THE USER'S DESKTOP

Christos Bouras
Computer Engineering and Informatics Department, University of Patras and
Research Academic Computer Technology Institute
N. Kazantzaki, Panepistimioupoli Patras, Greece
bouras@cti.gr

George Tsichritzis
Computer Engineering and Informatics Department, University of Patras and
Research Academic Computer Technology Institute
N. Kazantzaki, Panepistimioupoli Patras, Greece
tsixritzis@ceid.upatras.gr

Vassilis Tsogkas
Computer Engineering and Informatics Department, University of Patras
N. Kazantzaki, Panepistimioupoli Patras, Greece
tsogkas@ceid.upatras.gr

ABSTRACT

The exponential growth of the Web probes for new, better filtering systems and latency diminishing applications. We are presenting the caching features that are deployed in the client-side application that is developed within the scope of PeRSSonal, the generalized, multi-functional mechanism that delivers personalized, summarized and categorized news articles. We also evaluate the caching algorithm depicting the overall improvement that it offers to the client side desktop application.

KEYWORDS

News Channels, Client-Side caching, Data Preprocessing, News Personalization.

1. INTRODUCTION

The Web information age has brought a dramatic increase in the sheer amount of information that Internet users encounter every day, the access to this information, as well as the intricate complexities governing the relationships within this information. Nevertheless, web users today often experience long access latency due to network congestions, especially in peak hours or during big events. An effective technique to alleviate these problems is to cache frequently used data at proxies close to clients.

Caching popular queries and reusing results to speed up query processing is the most common query optimization technique for distributed environments like the Web. Additionally, XML is increasingly used in data intensive applications. Common applications that utilize caching of web content are web browsers. For example Mozilla browsers (Netscape, Firefox) use a cache system that temporarily stores all documents downloaded by the user. At first this may seem odd; however, this achieves making visited documents available for back/forward, saving, viewing-as-source, etc. without requiring an additional trip to the server. It likewise improves offline browsing of cached content as Mozilla <http://www.mozilla.org/projects/netlib/http/http-caching-faq.html> uses. Another interesting cache feature found in most of the web browsers is “Lazy Update”. This feature allows a Web site to cache an object as well as to be able to count the number of hit counts for the object; this is useful for advertising images, start pages, or search pages. For a lazy update page, the object is downloaded and cached the first time the page is visited. Subsequently, the user will see the cached copy. If the server has a new copy, the page will be

downloaded into the cache in the background. The next time the user visits the page, the updated content will be displayed < http://msdn.microsoft.com/en-us/library/bb250440(VS.85).aspx>.

As predicted by Terry & Ramasubramanian (2003), web services have emerged as the dominant application on the Internet. The growth in the number of Web services has been phenomenal over the last three years and most importantly, most of the services currently provided by the WWW are Extensible Markup Language (XML) – based services, thus XML Web services are the building blocks for constructing Web applications that leverage existing investments in Web technology. Within this scope, we are implementing a novel Web service that indexes news articles from several major or minor news portals and delivers XML formatted information directly to the user's desktop. PeRSSonal (Bouras et al., 2008), the automatic summarization, text categorization, personalized syndication system, applies several data mining techniques through a layered infrastructure that achieves filtering of information and adaptability to the user. We are focusing on news articles that are gathered from numerous news portals from around the world and we are targeting to the alleviation of the end-user from the cumbersome task of searching through this overwhelming plethora of information.

Client side caching has been widely employed to reduce the response time delay and improve the network utilization (Rizzo & Vicisano, 2000). However, as explained by Douglis et al. (1997), the benefit of caching diminishes as Web documents become more dynamic. A cached document may be stale at the time of its request, given that most Web caching systems in use today are passive (i.e., documents are fetched or validated only when requested).

It is possible to define at least two different caching structures in web services architecture (Fernandez et al. 2005). The first one is a two level caching architecture, which involves the existence of two entities: server and client. This architecture is not new, and caching is applied here using a typical implementation, like the one used by web browsers that access web servers directly, for instance. A three-level caching architecture (3LCA) is also possible. This kind of architecture is likely to occur in an environment where intermediate elements are an active part of the whole architecture. In a 3LCA, a third component appears, the intermediate element, who can also be an active actor in the caching system. A sample implementation of this caching architecture is the one used in web proxies, which are the intermediate elements between web clients (web browsers) and web servers. In our work we are utilizing a two level caching infrastructure since the caching system is embedded into the client application and requires no third party infrastructure.

There are numerous works on improving web services performance by using caching; some of them are based on several different programming mechanisms. For example, Goodman (2002), proposes a solution based on the use of a cache object. This object is a Java one that must be managed from the web service logic. Other kinds of solutions are based on the use of HTTP headers to manage the expiration of the HTTP responses. Microsoft support < http://support.microsoft.com/kb/318299> proposes the use of programming attributes (similar to compilation directives). The use of the WebMethod attribute, together with the CacheDuration property, allows a simple way to control the time-to-live (TTL) of the response. However, this solution is based on the use of HTTP, enforcing a dependency between a web service (and its business logic) and a transport layer (HTTP) which should generally be avoided.

There are also several works on news caching. Gschwind and Hauswirth (1999) designed a high-performance cache server known as NewsCache for Usenet News (a high traffic news source). The cache server is intended to save network bandwidth, computing power, disk storage, and be harmonious with existing standards. They describe the results of an experimental assessment which prove to work efficiently on several cache replacement strategies.

In this paper, we are dealing with the effective and adequate caching of personalized news summaries and information about articles that derive from the WWW to the user's desktop using a two level caching architectural approach. We present the caching algorithm that is used for speeding up the desktop application and the delivery of information within the scope of PeRSSonal in general. The rest of the paper is structured as follows: in section 2 we give in a nutshell the PeRSSonal's architecture. Section 3 presents some implementation issues of the caching system. Section 4 gives the algorithmic outline of the caching procedure. In section 5 we present the evaluation results for the client side application and section 6 concludes this paper with some additional thoughts for future additions to the system.

2. ARCHITECTURE

The system consists of four layers which work autonomously and collaborate through a centralized database. The web interface handles the incoming information flow of the mechanism which is then directed to the interior subsystems. The first level of analysis is responsible for the crawling of RSS feeds from major news' portals, and the discovery of the useful text portions within html pages. Content is thus crawled and fectched from a multitude of news web sites from around the Web. Text preprocessing techniques follow, and the results are led to the next level of analysis where core information retrieval techniques are located. These include: text categorization, text summarization and information personalization. Finally the results are presented to the end users through the information presentation subsystem which delivers information to the user's browser or client side desktop application. An important feature that the server side of the PeRSSonal mechanism offers is a query cache of its own in order to improve the response times of the requested pages. However, this cache level is database-based thus storing the results from non-expired queries to the SQL database for future reuse. More information about server-side caching of PeRSSonal is beyond the scope of this work.

In the current paper, we extend the delivery capabilities of PeRSSonal by enriching the client-side desktop application with caching features. As already explained, the delivery subsystem is utilizing XML channels for the delivery of information to the client. The aforementioned implemented system architecture is presented in Figure 1 with the modified component in the dashed box.

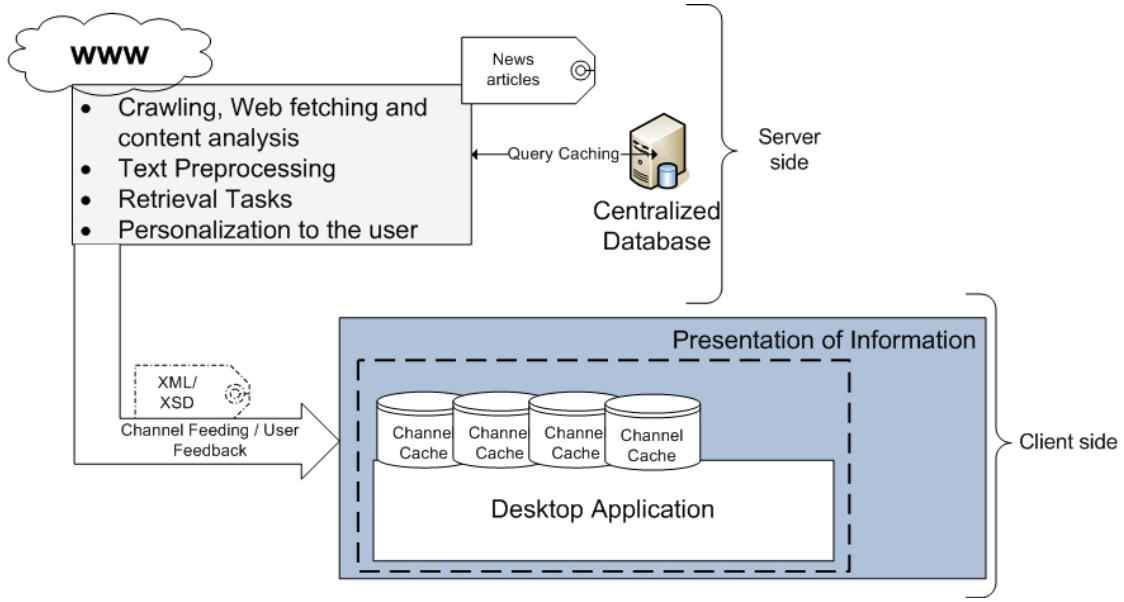

Figure 1 System's architecture – A Client/Server approach

The caching infrastructure of the client application is based on raw text files for information storage. However, any other storing subsystem can be used, e.g. SQLite database, etc. An important aspect to note is that the system's performance depends heavily on user feedback based on implicit rather than explicit choices. A user is not expected to provide direct answers to whether an article is of her appetite or not. On the contrary, the system keeps track of the articles/summaries the user chooses to read as well as the viewed times and implicitly maintains a user profile. Thus there is a continuous feedback channel that makes sure that the user's profile is updated on the server. This channel utilizes also the cache in a way to properly support offline usage of the application; in case of disconnection, feedback is kept in the local cache and transmitted when the connection is re-established.

3. IMPLEMENTATION ISSUES

PeRSSonal features a “channelled” approach for delivering interesting information to the end user. There are two different types of channels that the system uses. The first channel type contains news articles content and metadata about the articles themselves; their representation is made through XML/XSD schemas. Among the types of news channels that PeRSSonal features are: recent, personalized, readers’ choices, and category specific (e.g. business/sports/etc. articles). Each XML channel of this type contains the article’s title, summary, origin URL, date of publication, origin RSS and categorization as completed by the particular subsystem.

The second channel type that the system delivers contains related information about a specific article that has been selected by the user. More specifically, related, similar and identical articles are conveyed by this type of channel, empowering the user with easier tracking of articles that are of his/her interest. The distinction between those two channel types is because of their different XML structure (XSD schema). Related, similar and identical articles are computed on the server side using the cosine similarity metric.

The client side caching scheme that is implemented is based on flat XML files. The client is keeping track of the grabbed documents in a local directory using a naming convention that allows fast indexing of the fetched content. More specifically, when the recent or personalized articles channel is fetched, the stored XML file is named accordingly based on the date of the last inserted article and the cache file is updated in order to include the new fetched articles as well. In this manner, subsequent XML responds from the server can be easily compared for invalidation to the local cache. Should a channel be found updated on the server, meaning that new articles have been inserted in the channel that don’t exist in cache, the cache file is updated using the new naming and content. Notice that since the personalization procedure has already taken place at the server-side, all of the delivered articles are of relevance to the user’s profile. Moreover, the update procedure guarantees that the user receives fresh articles, even though a user can dictate the amount of articles that he prefers to be fetches on each cache-fill (and so keep track of older articles). A user will most likely be more interested in newer articles (even though less related to her profile) rather than old ones.

4. ALGORITHM ANALYSIS

In this section we are presenting a novel caching algorithmic approach that was implemented for the presentation of information by our client side application. As explained earlier, the presentation subsystem utilizes two different channel types with different XML structure and usage and thus we describe the procedures that are followed in order to keep the cache in an up to date state for both of the cases.

The algorithm describing the caching case for the first type of channels (from now on referenced as recent channel type) is depicted in Algorithm 1.When the client application launches, it starts the GrabberRecent thread which runs asynchronously and fetches the requested articles of the specific recent type channel. Following the fetching of an article set from the grabber, the XML parsing of the server’s response takes place based on the predefined server’s XSD schema that is utilized. Next, the GrabberRecent thread sleeps for a predefined period of time between subsequent fetches. Afterwards, the retrieved articles are stored in the suitable memory structure (grb_recent) and simultaneously, the main process thread is signaled to deliver the newest articles to the User Interface.

In this sense, when the main application confirms that the grb_articles structure isn't empty, the readcache procedure is called, as depicted in the access_latest_cache function of Algorithm 1. The readcache function checks if the cache is in a valid state, which means that it is up-to-date and the user preferences for the requested articles are met. In order for the readcache function to check the state of the cache, we utilize a special naming scheme which allows us to get this information without the need to explicitly parse the cache file. Keeping in the cache file name the publication timestamp of the latest article that was inserted in the cache, readcache merely compares this timestamp with the publication dates of the articles stored in grb_recent. Additionally, if the publication dates of the articles in cache are older than the threshold parameter, readcache also returns that the cache is in invalid state. At any other occasion, readcache successfully returns the requested articles that fulfill the parameters, how_old and num_articles which express the maximum acceptable aging date and the total requested number of articles respectively. When the readcache procedure notices an invalid state for the cache, the update_cache function is called which takes

care of the explicit population of the caching file with the latest articles (since they are available by the grabbing thread).

Even though the GrabberRecent procedure, which fetches the most recent documents, seemingly runs constantly, is it scheduled with a low CPU priority so that the actual overhead to the client's system is kept as low as possible. Furthermore its complexity is quadratic to the amount of the actual fetched articles.

```
GrabberRecent
  While (true) {
           thread_grabber_starts:
           fetch_articles() //grabber runs
           asynchronously
           grb_recent =
  XML_Parse_fetched_articles()
           sleep(fetching_period)
  }
access_latest_cache(how_many)
  While(grb_recent isEmpty){wait}
           R = readcache( recent, threshold,
num_articles, how_old, grb_articles,
user_articles)
           if (R==CACHE_INVALID){
           update_cache(grb_articles,cache_arti
cles,threshold)
           readcache(recent,threshold,num_artic
les,how_old,grb_articles,user_articles)
          }
           else if (R ==CACHE_OK){
           return grb_recent;
  }
```

Algorithm 1 Cache Handling for the Case of Recent Channels

```
 Grabber_Related
      fetch_related_articles(index)
      //grabber runs asynchronously
      grb_related = XML_ParseRelated_Articles();
      update_cache_related(index,
      grb_related_articles,threshold)

access_related_cache(index)
      R = Readcache_Related(index,    threshold,
      num_articles,  how_old.user_articles)
       While  (R == CACHE_INVALID)
          (wait for new articles to be available)
      Return (identical, similar, related
articles)
```

Algorithm 2 Cache Handling for the Case of Related Channels

The benefit from the aforementioned set of procedures is that, provided that the application has already run sometime in the not so far past, the user that first launches the application faces cached articles instantly. Following, when the grabbing procedure completes each loop, the UI is updated with the newest articles. From this point on of course, subsequent calls for articles fetching are visually transparent to the user meaning that articles are added on top or removed from the bottom of the lists as appropriate when the grabbing procedure finishes. The cache is kept to an updated state while the article's grabbing is done on the background. For the rare case that the application has not run for a period of time that exceeds the how_old parameter, and thus the cache contents are invalid, the user experiences a first time fetching delay.

The algorithm describing the caching case for the second type of channels (from now on referenced as related channel) is depicted in Algorithm 2. The main difference between the latest and the related channel types is that the related channel type contains information about each selected article. Thus, when the user selects an article, the application launches a grabbing thread for the related articles (related,similar,identical). Following the fetching of the XML file by the grabber, XML parsing takes place. Afterwards the update_cache_related procedure keeps the cache for the related articles of the particular item up-to-date. Simultaneously, the main process checks for a cache related file for the article that the user currently reads. If there isn't such file (cache is empty) then the application waits for the grabber_thread to end. Otherwise the requested numbers of related articles are read from the pre-existing cache file and are sent to the UI.

The benefit from the previously described procedure is that the user experiences a short delay only when his cache is empty. However, subsequent calls to fetch the related articles of a specific item are expected to return their results much faster. As it is obvious, the grabbing thread of Algorithm 2 lacks any loop for a continuous fetching of the related articles, which is quite reasonable since related articles change relatively rarely once they have been fetched. Nevertheless, related articles do change as new articles are added to PeRSSonal and hence checking for them when the user selects the article is crucial for the coherency of the cache.

5. EXPERIMENTS

In this section we are evaluating the client side cache implementation by comparing it with the content delivery of the Web interface of PeRSSonal as measured in our previous work.

In order to execute this experimental procedure, we created a basic user profile, selected randomly 100 articles that are indexed by our system and requested them both from the client side application and from the PeRSSonal web interface. Because of the fact that the client side application access preceded the web interface, the server side query cache was populated and was used in the case of the web interface, whereas the client application only utilized its local cache. The times are counted as the maximum between the time needed to fetch the actual article (type 1 channels) and its related content (type 2 channels). This is because of the fact that fetching the article's information and the related with its information is done in parallel. We used a time window of one minute between each subsequent fetch request from the client. The results for prefetched channels are presented in Figure 2.

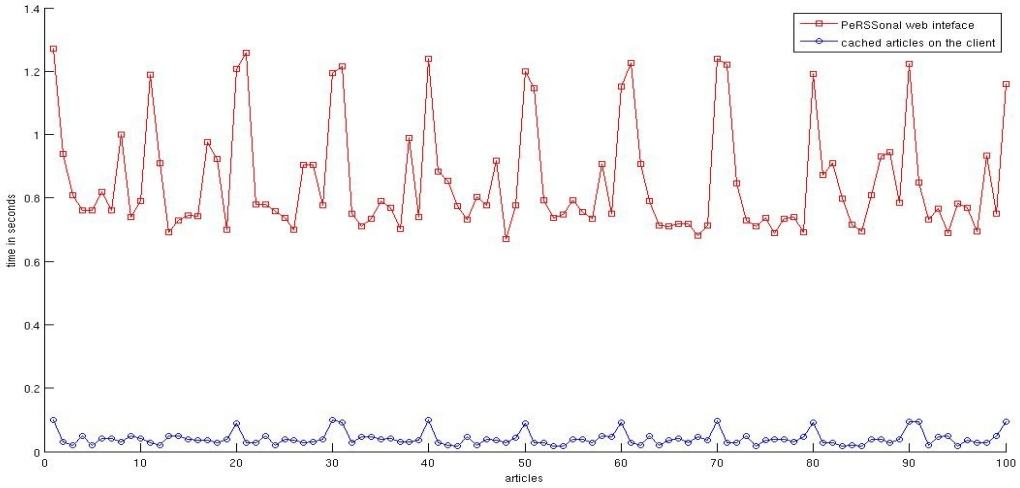

Figure 2 Fetching of articles on the client side application and the PeRSSonal web interface

From the graph in Figure 2 it is clearly depicted that the client side application features an overall improvement of four to six times when it comes to the total fetching period. However, it is essential to note the following fact: about 85% of the needed time for fetching the articles is spent in fetching the related channel due to the complex queries needed to evaluate the similarities between articles. This is so because of the fact that the discovery of related articles requires multiple queries to the database and cosine similarities calculations. The aforementioned explains to an extent the results that are depicted in Figure 2, since the related channel is cached on the client side and its transmission time from the cache to the application, once cached, is trivial.

The variations of the fetching times for both of the cases observed in Figure 2 are explained by the fact that different articles have different sizes and different related information to be fetched from the server. For example small, recently added articles about health might have none related article's information stored in the database. The peaks observed in the exact multiples of 10 articles are due to the sizes of the specific articles. We deliberately selected articles two or three times larger than the average article size for those experiment points. This is clearly illustrated in Figure 3 where the sizes of the articles plus the related information for each of them are presented. The aforementioned sum covers the total amount of information needed for the UI depicted articles to be coherent with respect to the information kept in the database. This information is fetched by the algorithms described in section 4.

Even though the previous results are somehow expected, one immediate implication of the diminishing of the articles fetching times is low latency between the moment that the user requests the information and when it appears on the UI. This in turn leads to the impression that the client application is almost running locally making in turn the communication times with the server transparent.

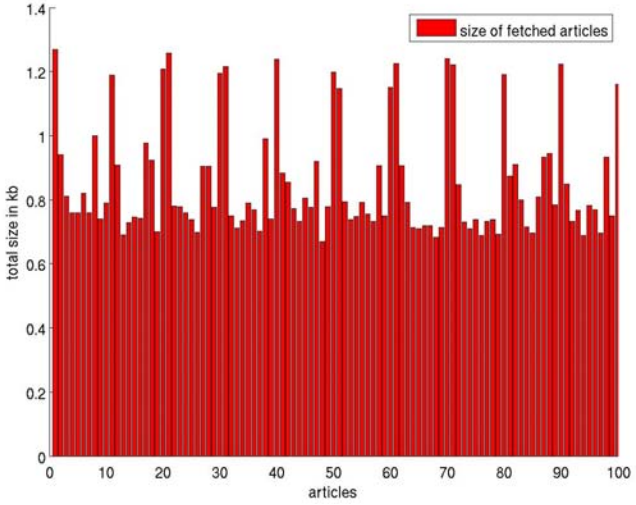

Figure 3 Total size of information needed for the coherency of each article

Moving to our second set of evaluation, we tried to determine the effectiveness of the developed caching system as far as its hit ratio is concerned. For this set, we asked 10 human system users to register to PeRSSonal and use the service through the desktop application for a month's period of time, which is considered enough for their profile to reach a steady state. We then collected from the application's log files, which expressed the users browsing behavior, the hit ratio percentage of the cache system according to their personalized choices. The caching system identifies a "hit" when a request is served using the local file structure; if on the other hand a round-trip to the server is required the request is marked as a "miss". The results are presented in Fig. 4 from which it is deducted that the caching system offers an overall hit ratio of around 47% which is expected bearing in mind the following facts:

- The users' choices are random as far as the system is concerned, following their established user profiles.
- Users tend to revisit previously read or similar articles that have already been fetched by the caching system.

The aforementioned hit ratio means that almost half of the requests made from each user, for either of the transmission channels, are causing a cache miss and are thus served by a new request to the PeRSSonal server.

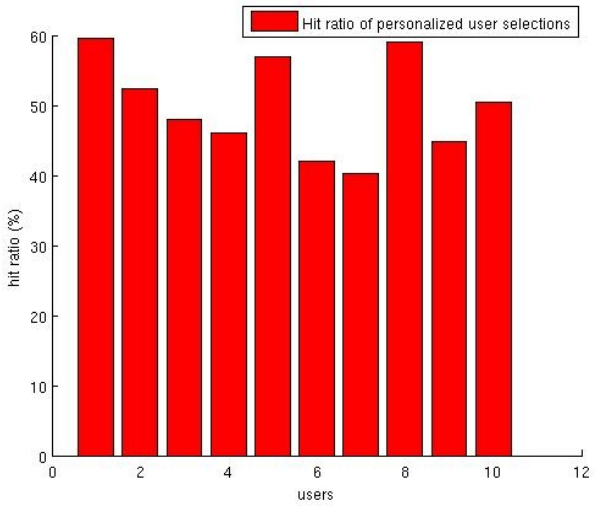

Figure 4 Hit ratio of the cache system for random user selection

6. CONCLUSION AND FUTURE WORK

We have outlined the caching technique that is utilized by the developing desktop application of PeRSSonal. The client side caching system utilizes two different XML channels types for transmission: one for the recent or personalized articles content and one for the related articles to the currently viewed one. The local storage of information is done using flat XML files with a naming schema that provides easy access by the caching algorithm. The proposed algorithms provide a simple yet effective framework to improve the application's responsiveness to the user's choices.

The conducted experimental procedure that revealed the superiority of the client side application in terms of responsiveness over the web based interface of PeRSSonal. For previously cached articles we found that the client application offers four to six times lower fetch times for the serving of channels. Furthermore, we discovered that the proposed caching system suffers from low hit ratios when it comes to serving personalized content to the registered user. The previous probes for our further research in the field in order to predict the future user choices based on the established profile. Prefetching is currently under research and will enrich our caching system with the ability to predict the future user choices amending thus the hit ratio.

REFERENCES

Bouras, C. et al, 2008. PeRSSonal's Core Functionality Evaluation: Enhancing text labeling through Personalized Summaries, *Data and Knowledge Engineering Journal, Elsevier Science*, Vol. 64, Issue 1, pp. 330-345.

Bouras, C. and Tsogkas, V., 2009. Personalization Mechanism for Delivering News Articles on the User's Desktop. *The Fourth International Conference on Internet and Web Applications and Services (ICIW 2009)*, Venice/Mestre, Italy.

Douglis, F. et al. 1997. Rate of change and other metrics: a live study of the World Wide Web, *In Proceedings of USENIX Symposium on Internet Technologies and System.*

Fernandez, J. et al, 2005. Optimizing web services performance using caching, *Proceedings of the International Conference on Next Generation Web Services Practices.*

Goodman, B. D., 2002. Accelerate your Web services with caching, *DeveloperWorks Journal, IBM*, viewed: 14 October 2008, <http://www-306.ibm.com/cics>.

Gschwind, T., and Hauswirth, M., 1999. NewsCache: A high performance cache implementation for Usenet news, *Proceedings of the 1999 USENIX Annual Technical Conference Monterey, California* pp.16-16.

Microsoft Internet Explorer Lazy Update, viewed 28 October 2008 <http://msdn.microsoft.com/en-us/library/bb250440(VS.85).aspx>.

Microsoft Support. Perform Output Caching with Web Services in Visual C# NET, viewed 14 October 2008, <http://support.microsoft.com/kb/318299>.

Mozilla HTTP Caching, viewed 28 October 2008, <http://www.mozilla.org/projects/netlib/http/http-caching-faq.html>.

Rizzo, L. and Vicisano, L., 2000. Replacement policies for a proxy cache, *IEEE/ACM Transaction on Networking*, 8 (4) pp. 158–170.

Terry, D.B. and Ramasubramanian, V., 2003. Caching XML Web Services for Mobility, *ACM Queue*, V.1 No.3 pp. 70-78.