The following paper was originally published in the
Proceedings of the USENIX Symposium on Internet Technologies and Systems
Monterey, California, December 1997

Alleviating the Latency and Bandwidth Problems in WWW Browsing

Tong Sau Loon and Vaduvur Bharghavan
University of Illinois at Urbana-Champaign

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Alleviating the Latency and Bandwidth Problems in WWW Browsing

Tong Sau Loon   Vaduvur Bharghavan
Department of Electrical and Computer Engineering
& Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{s-tong,bharghav}@crhc.uiuc.edu, http://timely.crhc.uiuc.edu/

Abstract

This work addresses three problems that are associated with Web browsing: (a) low bandwidth available to the end user who is connected via slow modems or outdoor wireless networks, (b) long and variable latencies in document access, and (c) temporary disconnections of mobile users. Three techniques are used with a variety of heuristics in order to overcome these problems: (a) profiling user and group access patterns and using these profiles in order to pre-fetch documents, (b) filtering HTTP requests and responses in order to reduce data transmission over bottleneck links, and (c) hoarding documents based on user profiles in order to support limited web browsing even during disconnection. In this paper, we describe the design and implementation of a WWW proxy-based system that incorporates the above techniques. We describe our experiences with the proxy system, and present performance results that show an improvement in the experience of Web browsing using this system.

1 Overview

In the last few years, the World Wide Web has had a remarkable effect on computing and communications in general, and Internet traffic in particular. Due to the explosion of the offered network load and the inherently best-effort paradigm of Internet service, WWW users typically notice long latencies and large variations in latency for web accesses. The scarcity of network bandwidth, particularly of the last link for users connected via low bandwidth modems and outdoor wireless networks exacerbates the latency problem, as does the transmission of increasingly graphics-oriented documents over slow networks. Mobile users face additional challenges in terms of frequent disconnections. In order to solve the above problems, we have built a WWW proxy-based distributed system which is compatible with existing browsers and protocol standards. This paper presents the design and implementation of our system and shows the performance improvements we obtained using this system in conjunction with our standard browsing environment.

Three problems motivate this work: (a) low bandwidth available to the end user who is connected via slow modems or wireless networks, (b) long and variable latency due to congestion in the network, best-effort service in the Internet, and transmission of large amounts of data over slow links, and (c) temporary disconnections of mobile users - either involuntary due to fades, or voluntary to save cost and battery power. In order to find effective solutions to the above problems, our work is based on three key observations: (a) User accesses to WWW documents have been shown to follow certain patterns, albeit changing over time [8] - these changing patterns can be learned by monitoring user accesses and used for both pre-fetching and hoarding. (b) Most large documents are graphics-intensive and graphics data can tolerate loss - thus, filtering images can significantly reduce data transmission without compromising severely on quality. (c) Groups of users with similar interests tend to access similar documents - the commonality of their access patterns can be learned by monitoring their access patterns. It should be noted that none of the above techniques are unique to our work. Caching of documents based on recent history of accesses is provided with most browsers (e.g. Netscape, Explorer). Intelligent prefetching based on document hyper-links and user access patterns have been proposed in several studies [3, 8, 9]. Filtering in order to adapt to dynamic network quality of service has been proposed in related work, both in the context of WWW accesses and in the context of application adaptation in general

[7, 16, 21]. Collaborative filtering has been proposed in the context of newsgroups [10] as well as WWW [1]. Hoarding has been proposed in the context of file systems support during disconnected operation [14]. The contribution of this paper is the combination of several mechanisms and the use of multiple heuristics in order to intelligently pre-fetch documents (based on user profiles, group profiles, and associated heuristics), filter documents (adaptively to varying QoS), and hoard documents anticipating disconnection (based on a hoard database that is learned over time as well as a user-defined hoard file). Performance results show that our system can learn and adapt to changing user behavior quickly, and significantly improve the experience of WWW browsing once the user profile is learned.

The rest of the paper is structured as follows. Section 2 describes the architecture of the system. Section 3 discusses the various heuristics employed in order to improve efficiency. Section 4 provides implementation details, while Section 5 presents performance results. Section 6 compares related work to our approach, and Section 7 concludes the paper.

2 Architecture of the WWW Proxy System

The three key aspects of our system design are pre-fetching documents based on user and group profiles, filtering retrieved documents based on the available network quality of service, and hoarding documents in anticipation of network disconnections (for mobile users). For pre-fetching and hoarding to be effective, the cached copy of the documents must be as close to the browser as possible. For filtering to be effective, it must be done as close to the server as possible; in particular, filtering needs to be performed *before* the bottleneck link on the retrieval path of the client, while pre-fetching and hoarding need to be done *after* the bottleneck link.

In the ideal case, a server would have a set of filters associated with a document type. A client request would be accompanied with the measured network quality of service. The server would then retrieve the document and pass it through the filter (with the QoS level as a parameter) before sending it back to the client. The advantage of this design is that only the required data is sent over the network, thereby decreasing the latency of access. Besides, if the user has to pay for receiving data over the network (proportional to the amount of data received), this mechanism can reduce the cost of Web access. The disadvantage of this design is that it requires QoS aware servers, and also places the burden of filtering on the server.

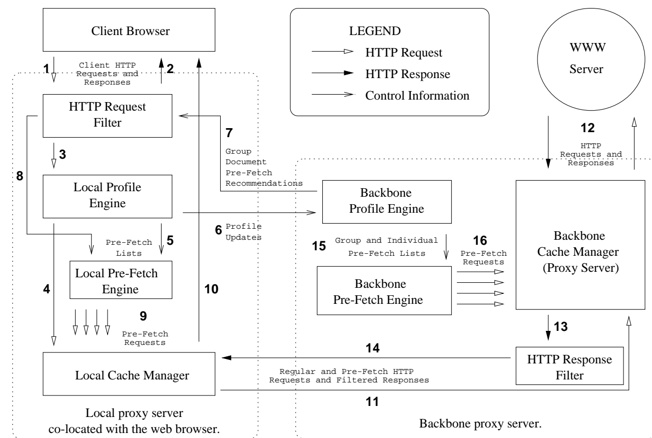

Figure 1: System Model

In cases where the user is connected to the network via a slow modem link or an outdoor wireless link, the last link typically happens to be the bottleneck link in the retrieval path. In this case, filtering the document before the last link may work just as well in reducing latency and maybe cost. Since this does not require any change in the server, we use this model for our system.

The architecture of our WWW proxy system is shown in Figure 1. A WWW browser points to a local proxy server, through which all requests are routed. The local proxy server contains an HTTP request filter, a profile management engine, a pre-fetching engine, and a cache manager. The local proxy server points to a backbone proxy server. Thus, the local proxy server acts as a server to the browser but as a client to the backbone proxy server.

The backbone proxy server essentially contains the same components as its local counterpart, but may service multiple users. The backbone proxy server thus handles both group profiles and individual profiles while the local proxy server handles only individual user profiles.

In order to effectively manage the usage profile, all user accesses must traverse through the local proxy server; thus, the browser's cache is disabled. This is because some HTTP requests would be intercepted by a browser cache if it were not disabled, and the

local proxy server would not be able to learn the access pattern properly.

Profile-based pre-fetch is performed at both the local proxy server and the backbone proxy server, although the backbone proxy server does a more aggressive pre-fetch (more documents). Hoarding is done by the local proxy server. Filtering is done on both HTTP requests (e.g. to reduce HTTP headers) and HTTP responses (e.g. to clip images). The WWW server is not required to have any special functionality that is specific to our system.

2.1 HTTP Request/Response Paths

In this section, we provide an overview of each of the components in the architecture by means of an example of an HTTP request/response path. We then describe the functionality of each component in the system.

The numeric steps below refer the the steps shown in Figure 1:

1. When the user requests a document, the browser issues the request to the local proxy server.

2. This request first goes through an HTTP request filter. The request may be immediately satisfied (e.g. if the request is for a site that is blocked out, such as advertisements), may be modified (e.g. header compression [13]), or may be passed through without modification. Determination of filterable requests is based on substring matching of URLs to key strings and running corresponding scripts defined in a configuration file.

3. If a response was not generated immediately, the request is logged by the local profile manager and the user profile is updated.

4. The request is then passed on to the cache manager.

5. The profile manager will create a pre-fetch list based on the usage profile and send it to the pre-fetcher.

6. Requests which change the profile, specifically URLs which point to HTML pages, are sent to the backbone profile engine to enable backbone aggressive pre-fetches and to update the backbone profiles. Note that the use of an explicit connection to send the profile updates is mainly for ease of implementation. A more efficient mechanism would be to piggy-back such data on HTTP requests that gets transmitted from the local proxy server to the backbone proxy server.

7. Periodically, the backbone profile engine returns a list of recommended pages to pre-fetch based on group profiles. This can occur when many users of a particular group visit a particular page. Similar to above, such information can be piggy-backed onto HTTP responses in a more efficient implementation.

8. The recommended URLs are operated on by a function in the HTTP request filter to eliminate URLs that would be filtered (i.e., we do not want to pre-fetch items that we will filter). This new list is submitted to the pre-fetcher.

9. The pre-fetcher collates the pre-fetch list and group document pre-fetch recommendations that are found to be not filterable. It then amortizes the pre-fetch requests to the cache manager.

10. If the cache has a fresh copy of the document originally requested, the request is satisfied immediately.

11. Otherwise, the request is forwarded to the backbone proxy server.

12. The normal HTTP transaction occurs between the backbone cache manager and the WWW server.

13. After retrieval, the document is passed through the backbone HTTP response filter.

14. The response is sent back to the local cache manager, who will cache the document if it is a cacheable item. It is then sent back to the browser (10).

15. The backbone profile manager maintains individual as well as group profiles. Periodically, it creates a list of recommended group documents and sends it to the local proxy server (7) of each member of the group. As profile updates arrive, it creates a list of documents to pre-fetch based on individual and group usage profiles. The only difference from the local pre-fetch list is that the backbone list is longer (i.e., we do more aggressive pre-fetches on the backbone). This list is then submitted to the backbone pre-fetch engine.

16. The backbone pre-fetcher will issue the necessary pre-fetch requests.

2.2 HTTP Request/Response Filters

The filter engines are responsible for modification of HTTP requests and responses in order to reduce data traffic over the bottleneck link. Typical examples of filtering include reducing the color depth of images, clipping images, transmitting parts of audio files only, transmitting the first few words of each paragraph in a document, reducing the HTTP request header, suppressing requests to documents from a particular server, etc. Every user has his/her own HTTP request filter and HTTP response filter.

Invocation of the appropriate filter is accomplished by means of a rules database. A rule in the database specifies the invocation of a filter based on its document type, available network quality of service, size of the document, etc. The goal of the rules database is to allow adaptive filtering based on dynamic network conditions and data types. While the set of rules in the rules database is currently small, the architecture allows for more sophisticated filtering.

The HTTP request filter also notifies the user of the documents that are pre-fetched via a well-known URL. That is, it intercepts a well-known URL and formulates an HTML page which represents the pre-fetched items in the cache. This allows the user to see which URLs have been pre-fetched and manually set or modify the list.

2.3 Usage Pattern Profile

A usage profile is a representation of a user's or group's usage pattern of the Web. The profile is learned over time by monitoring the stream of HTTP requests of users. Using the profile to determine which documents to pre-fetch, rather than simply using the document layout, can significantly improve the efficiency of pre-fetching. The usage profile is a directed weighted graph, where the nodes represent URLs and edges represent the access path. The weight of a node u indicates the frequency of access of the corresponding URL, while the weight of an edge *(u,v)* indicates the frequency of access of the URL v immediately following the URL u. In order to reflect the changing access patterns of users and the temporal locality of accesses, recent history is given precedence in the weighting heuristic.

Let $n_t(u)$ represent the number of accesses of node u (i.e. the URL corresponding to node u) during the time interval t, $n_t(u,v)$ represent the number of accesses of node v immediately following node u, $w_t(u)$ represent the weight of node u after the time interval t, and $w_t(u,v)$ represent the weight of edge (u,v) after the time interval t. The weights of nodes and edges are computed as follows:

- $w_{t+1}(u) = w_t(u).\alpha_1 + n_t(u).\beta_1$
- $w_{t+1}(u,v) = w_t(u,v).\alpha_2 + n_t(u,v).\beta_2$

where α_1, α_2, β_1, and β_2 are constants which indicate the relative weights of recent history versus past history. These constants, along with the time window t, play a key role in determining how effectively the profile adapts to the changing user access patterns. Modifying these parameters will determine whether the profile does long term adaptation or short term adaptation. Based on our own experience, we have currently set $\alpha_1 = 0.9, \beta_1 = 0.1, \alpha_2 = 1.0$, and $\beta_2 = 1.0$. This means that we have set a very high weightage to previous history. This is because we wanted our system to be insensitive to spurous bursts of visits to sites that will not be visited again i.e. long term adaptation. We have set t to be the time between two successive *sessions*. This means that the weights are recalculated at the beginning of every Web session. While the current values of the constants are based on what worked for our usage profiles, we plan to do more experiments in order to determine the weights in the graph. While our weights are determined solely by frequency of access, we plan to use the following parameters for determining weights in the future: percentage of membership that uses an edge or node for group profiles, expected latency, liveliness of documents, and size.

Group usage profiles are a natural extension of the idea of exploiting individual usage profiles to predictively pre-fetch documents. They also fit naturally into collaborative filtering of documents discussed in [10]. Group usage profiles are inherited by users that join the group. Thus, a new user would first enroll in several groups. This will ensure that there is some form of informed speculative pre-fetch service for the user while his/her own usage pattern is being learned by the local profile engine. In addition, as the interests of the group change over time, the user will be able to inherit the changes automatically.

In the case of group profiles, we keep more state. In particular, we are interested in the number of members in a particular group who have visited a particular URL and who have used a particular edge.

This allows us to make recommendations on which URLs to pre-fetch. For example, if more than 50% of the group uses a particular edge (u, v), then we recommend v when a member of the group visits u.

While individual profiles are used for predictive pre-fetches, group profiles serve two purposes. First, a group profile is inherited by a new member to a group, hence, while his/her individual profile is being learnt, it is still possible to predictively pre-fetch. Second, group profiles are useful for notifying a member of pages that most of the other members of the group are visiting.

2.4 Profile Engines

The local profile engine is mainly responsible for maintaining a single user's usage profile. It receives filtered HTTP requests from the HTTP request filter and then updates the profile. It then creates a pre-fetch list based on the profile and submits the list to the local pre-fetcher. At the same time, the local profile engine updates the backbone profile engine.

The backbone profile engine is responsible for maintaining individuals' profiles as well as group profiles. After it receives an update from the local profile engine, it creates a (longer) list of items to pre-fetch based on the individual's profile and submits it to the backbone pre-fetcher. Note that the list based on individual profile is longer because we want to ensure that items are readily available at the backbone proxy server, should it not be found in the local cache. It also creates another pre-fetch list based on the groups that the user has subscribed to. This list is then submitted to the backbone pre-fetcher. Periodically, a list of recommendations are sent back to the local proxy server. Note that instead of pushing the document directly to the user, we employ the use of a notification system with the local proxy server making the final decision of whether to pre-fetch a document or not. This pull with notification mechanism is more flexible than a pure document push from the backbone proxy server because the user might not need all the documents.

2.5 Pre-fetch Engines

The local pre-fetch engine's main responsibility is to issue pre-fetch requests to the local cache manager based on the lists obtained from the local and backbone profile engines. The recommendation list that arrives from the backbone profile manager goes through the local filter first, in case there are some documents that the individual wishes to block out. It then amortizes the pre-fetch requests over time so that the local cache manager does not get overloaded. At the same time, the local pre-fetch engine makes sure that duplicate items are not pre-fetched.

A special case of pre-fetch is *hoarding*, which is done only by the local pre-fetch engine. In the case of hoarding, only the node weights are used in order to pre-fetch the documents that the user is most likely to require in the future. As in the case of disconnected file systems, this gives rise to the problem of caching versus hoarding. An approach similar to hoard-walking [14] is used in order to refresh commonly used items.

It should be noted that on the local machine, user requests are given priority over pre-fetch requests. That is, no pre-fetch requests are issued until there are no pending user requests.

The backbone pre-fetch engine's main responsibility is to issue pre-fetch requests based on the list obtained from the backbone profile engine. Note that multiple lists can be submitted to the backbone pre-fetcher if there are multiple users logged on. The backbone pre-fetcher will ensure that duplicate items are not pre-fetched and that pre-fetch requests are amortized over time.

2.6 Cache Manager

Each proxy server has a cache manager which maintains a cache of documents available as a result of user requests, pre-fetches, and hoard-walks. When the proxy server receives a request, if the document is not available at the cache, the request is forwarded to the next level of proxy or directly to the WWW server. Note that the browser cache is disabled, since all user requests must go through to the local proxy server in order to build an accurate user profile.

Should disconnections arise, the local cache manager is also responsible for providing the documents. In our system, if any control connection with the backbone machine is broken prematurely, the local proxy server will go into a *disconnected* mode. In this mode, it will only present to the browser the items on the local cache. If a request is made for an item that was not hoarded, the local cache manager returns an empty file. The browser will then present a "Document contains no data" message to

the user.

The backbone cache manager is any standard off-the-shelf proxy server. In the current implementation, we use *Squid* [4].

3 Performance Enhancing Heuristics

We used several heuristics in order to improve the efficiency of pre-fetching. While many of the heuristics listed below are simple and intuitively obvious, we found that a combination of these heuristics provided a remarkable performance improvement in our daily operation. This section lists the heuristics we used.

1. *Web Sessions*: The notion of a Web session had one of the greatest performance impacts. Without the notion of a session, the pre-fetch engine re-issued multiple requests for the same documents if a user accessed the same pages again later in the session. With the notion of a session, documents are only fetched once during the session. Subsequent requests are satisfied from the cache, unless the user explicitly requests a fresh access using the RELOAD button (which issues a HTTP "Pragma: no-cache" header). At the start of a session, documents with the highest node weights are hoarded. The idea of Web sessions is not new. Current browsers all have a notion of a Web session.

2. *Hoard walking*: Hoard-walking [14] periodically refreshes pages with the highest node weight. Since hoard-walking involves pre-fetching the pages in the user defined hoard file as well as the most heavy nodes in the learnt database, consequently, a large fraction of typical user accesses can be satisfied locally during disconnection.

3. *Issue Of Pre-Fetch Requests*: Pre-fetches are performed only when the network is "idle". In our system, only four ongoing pre-fetches are allowed at any time at the local proxy server and only eight ongoing pre-fetches are allowed at any one time in the backbone proxy server. Furthermore, on the local proxy server, pre-fetches are not started until there are no pending user requests. We observed performance improvements when we amortize pre-fetch requests. Of course, not issuing pre-fetch requests while there is a pending request might actually *lower* performance, especially in the case of requests with high delays. However, giving priority to user requests eliminates the harmful effects of pre-fetch tying up bandwidth when it is needed.

4. *Weighting edges*: Using weighted edges as opposed to only using node weights for pre-fetches ensures that the proper usage pattern is captured. When a user visits a URL, we should choose the edge with the heaviest weight rather than the adjacent node with the heaviest node weight. For example, consider the following scenario: C was visited 2 times from A and 100 times from B. B was visited 10 times from A. When a user is at A, B should be pre-fetched even though it has a smaller node weight than C.

5. *Dynamically determining a document's dependents*: We distinguish a document from the images that are linked to it (which constitute its dependents). Dependents do not appear in the user profile, because they are accessed automatically upon access of the document, and also because they change frequently over time. The original implementation stored the URLs of the dependents. However, due to the frequent changes in HTML documents, requests were being made for non-existent documents. This heuristic removed all those redundant pre-fetch requests, at the same time keeping the user profile graph small. Furthermore, pre-fetch requests for dependents are issued (at both the local proxy server and the backbone proxy server) before the browser issues them.

6. *Continued download of document*: Even after the user specifies "stop", we continue downloading a document in the local proxy server. This allows a user to click on another page while the previous page is being downloaded in the background. It also serves as a crude form of short-term user-specified hoarding or "user-driven pre-fetch". Even though this is not captured in the performance tests, we found it extremely useful when we were reading a page with links we knew we wanted to visit. We would click on the link and quickly press stop. This would issue the pre-fetch request but keep the browser on the current page. Later, when we eventually clicked on the page, it came up instantly.

7. *CGI scripts*: CGI and other dynamic pages are pre-fetched and retained in the cache for a short period of time. Currently, CGI pages are not cached either in browsers or proxies; thus

CGI latency is not hidden. However, with this heuristic, we can pre-fetch CGI pages and cache them for short periods of time in anticipation of an access. A CGI page is deleted upon the timeout, or after it is read once.

8. *HTTP Redirections*: HTTP response codes 301 and 302 indicate that a particular URL has moved. If the local proxy system detects this, it will store the redirection and provide the correct response to the browser. This prevents the browser from reconnecting to the server.

9. *Thresholds*: Since many documents are only accessed once from a parent document (e.g. a news item from a topic), we impose a minimum threshold edge frequency in order to pre-fetch the node. Another threshold we impose is the size of documents. Large documents (more than 1 megabyte) are not pre-fetched in our current implementation. However, we anticipate that with the inclusion of size as a weight parameter, this heuristic will be subsumed in the weighted edge heuristic.

4 Implementation

The WWW proxy caching system is written in C++ in the Linux environment. On the local machine, the user needs to run only one process, the local proxy server called *localCache*, in addition to the browser itself. On the backbone server, several processes need to be run. These are the backbone cache manager *squid* [4], the HTTP response filter *filterd*, the backbone pre-fetch engine *pfetcher*, the backbone group profile manager *gpm*, and the backbone communication surrogate *bcs* which talks to *localCache*. We will describe these processes in more detail next. It must be noted that other than *squid* [4], all the other processes can be combined into a single one that spawns into different modules. The use of different processes is purely to ease debugging. In essence, *filterd* and *bcs* keep states about the current session. The rest of the processes keep state about the backbone proxy server.

4.1 Local Proxy Server

The local proxy server, *localCache*, implements all the different parts of the local proxy server shown in Figure 1. Specifically, the HTTP request filter is implemented as a method call that does substring matching on HTTP requests and invokes scripts to create HTTP responses based on the rules database. The local profile engine is implemented as a class (a directed graph with weighted edges) which trims itself when the number of nodes grows too large. Currently, it trims itself when the number of nodes reaches 1024. It does so by removing half of the nodes (512) that are lightest according to the weighting method we described earlier. The local pre-fetch engine is implemented using a queue. The local profile engine feeds the queue with URLs. When a pre-fetch request is issued, the URL is removed from the queue and placed into a session set. The same URL will then no longer be pre-fetched, even if it was placed in the queue again. The local cache manager is implemented as a class that interfaces a URL to a disk file via a hashing function. Currently, HTTP response headers are also stored. In addition to disk store, we also keep a copy of pre-fetched items in virtual memory. This is more for ease of implementation than for performance. An optimized version of the program can use just the disk store and thus leave a small memory map.

Of particular interest with the local cache manager is the determination of an item's expiry time. Let `t_expire`, `t_date`, and `t_last_modified` be times (in seconds) extracted from the respective HTTP response headers. If a header is not available, the variable defaults to 0. Further, let `HTML_expire` and `IMAGE_expire` be the times (in seconds) the user provides. These times specify how long the user wants an HTML file or an image file to stay fresh in the absence of some/all of the above HTTP response headers. Let `t_now` be the current time in seconds at the local machine and `expire` be the time when the document expires. The algorithm for determining the freshness of an item in the local cache manager is as follows:

```
if (t_expire>0 && t_date>0)
  expire = t_expire - t_date + t_now;
else if (t_last_modified>0 && t_date>0)
  expire = (t_date - t_last-modified)/2 + t_now;
else if (t_last_modified>0)
  expire = (t_now - t_last_modified)/2 + t_now;
else if (HTMLDocument(url))
  expire = HTML_expire + t_now;
else
  expire = IMAGE_expire + t_now;
```

4.2 Backbone Proxy Server

For the backbone cache manager, we use *squid* [4]. However, any proxy server that understands standard HTTP/1.0 should work, since all communication with the backbone cache manager occurs via HTTP/1.0 requests.

The HTTP response filter is implemented in the process *filterd*. *filterd* reads a configuration file upon being started by a running copy of *bcs*. The configuration file indicates what scripts to run on what kinds of responses. For example, JPEG files might require the script *jpegfilter*. *filterd* listens to a well known port for incoming HTTP requests. It then forwards all requests to *squid* [4]. After it receives the response from *squid* [4], it checks to see if the response is filterable. In the above example, if the response file is called *picture.jpg*, *filterd* will issue the command:

```
jpegfilter picture.jpg tempfile.nam QoS
```

The script is expected to produce the filtered response in *tempfile.nam* based on the QoS parameter. *filterd* then makes necessary changes to the HTTP response headers and sends the new response back. Basically, *filterd* maintains the state of pending HTTP requests, responses, and the connection to the local cache manager and *squid* [4].

The QoS parameter measures the quality of service of the network connections between the local proxy server and the backbone proxy server along two somewhat orthogonal dimensions: rate and delay. Periodically, the local proxy server measures the HTTP request-response round trip time (RTT) for a request and signals the backbone proxy server to do the same for the same request. (This is done via piggy-backed HTTP headers.) The difference in the measured RTTs at the local and backbone proxy servers gives an RTT estimate for the HTTP request. By fitting a curve over a sequence of such measurements, the local proxy server can estimate the rate and delay QoS parameter. These values are then transmitted to *filterd* and fed back into the local HTTP request filter.

The backbone pre-fetch engine *pfetcher* keeps state of all current Web sessions. It is able to do this because every copy of *bcs* connects to it. It also maintains a queue of URLs that it feeds (as HTTP requests) to *squid* [4]. Each running copy of *bcs* will send their own pre-fetch list. *pfetcher* will amortize these pre-fetch requests to *squid* [4]. As pre-fetch responses arrive, *pfetcher* will determine the dependents and issue more pre-fetch requests if necessary.

The backbone group profile manager *gpm* keeps track of all the groups which have their group profiles stored on that particular backbone server. It also maintains the list of members of each group. It acts as a query-response server to the group profiles database. When a running copy of *bcs* informs *gpm* where its user is currently located, *gpm* will update the group profiles of all the groups that the user is subscribed to. It then returns a list of recommendations based on the updated group profiles to *bcs*.

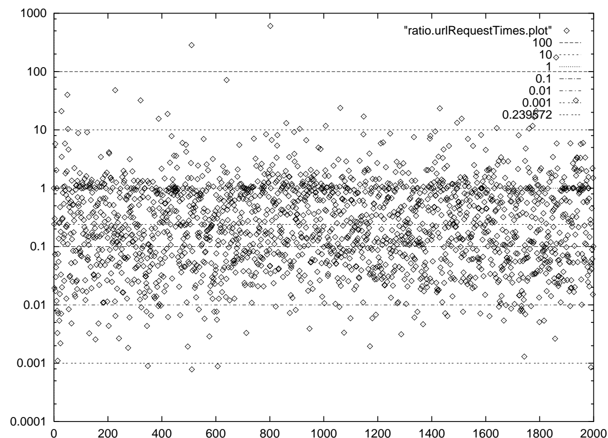

Figure 2: Ratio of individual URL request times. X-axis represents each of the 2000 URLs. Y-axis represents the ratio of delay of each URL with the proxy system to the delay with only *squid* [4].

The backbone communication surrogate *bcs* is a forking process that forks itself for every new Web session by a different user. It maintains the per-user states. That is, it maintains the user profile, the groups a user belongs to, and the user's current QoS. It updates *gpm* about the movements of the user and obtains a pre-fetch list from it when it wants to recommend pages to the user. It connects to *pfetcher* and sends it a list of items to pre-fetch based on the user's individual profile. This is for the aggressive backbone pre-fetch. It also spawns the user's HTTP response filter, i.e. *filterd*.

5 Performance Measurements

Performance of the system was evaluated with a browser simulator called *surf*. *surf* reads a text file containing a list of URLs to access. It then issues HTTP requests for each of the URLs and fetches the dependents if necessary. *surf* simulates user reading time by sleeping an amount of time proportional to the HTML file size without the HTML tags. This is called the *readspeed*. If a URL is not retrieved within a certain amount of time, the whole URL is abandoned. This is specified by the *impatience* parameter and is used to detect Web sites that are down. There is also the caching version of *surf* called *csurf*. The only difference is that *csurf* fetches every unique URL only once per session. This simulates the internal browser cache.

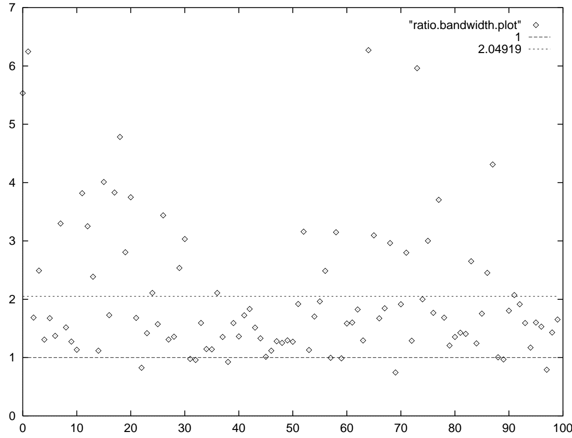

Figure 3: Ratio of session HTTP bandwidth. X-axis represents each of the 100 sessions. Y-axis represents the ratio of the HTTP bandwidth of the session with the proxy system to the HTTP bandwidth of the session without the proxy system.

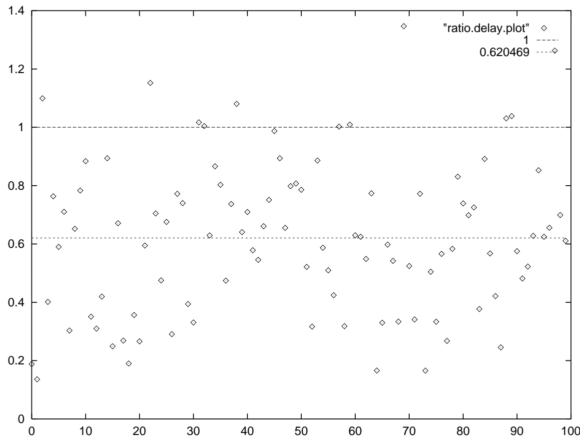

Figure 4: Ratio of average HTTP request delay. X-axis represents each of the 100 sessions. Y-axis represents the ratio of the average request delay of the session with the proxy system to the average request delay of the session without the proxy system.

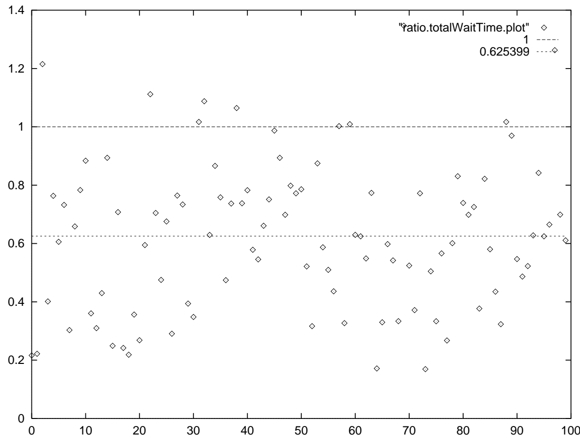

Figure 5: Ratio of total network wait time. X-axis represents the 100 sessions. Y-axis represents the ratio of the total network waiting time of the session with the proxy system to the total network waiting time of the session without the proxy system.

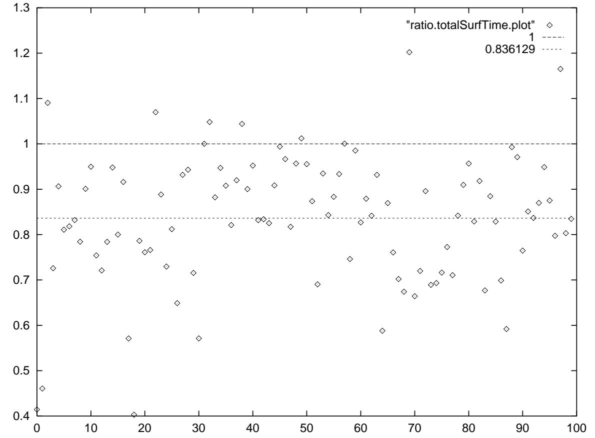

Figure 6: Ratio of total surfing time. X-axis represents the 100 sessions. Y-axis represents the ratio of the total time of the session with the proxy system to the total time of the session without the proxy system.

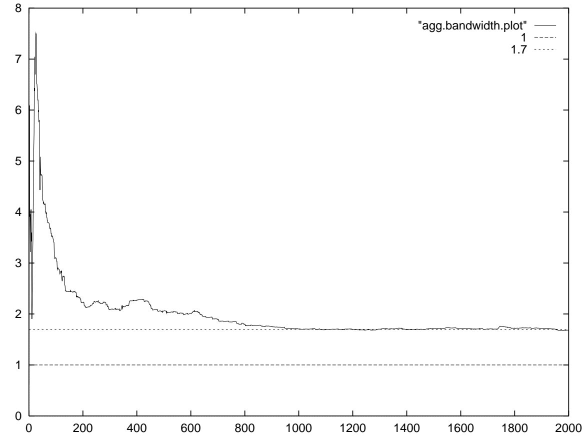

Figure 7: Ratio of HTTP bandwidths over 2000 URLs. X-axis represents the 2000 URLs. Y-axis represents the ratios of the HTTP bandwidth for all the previous URLs.

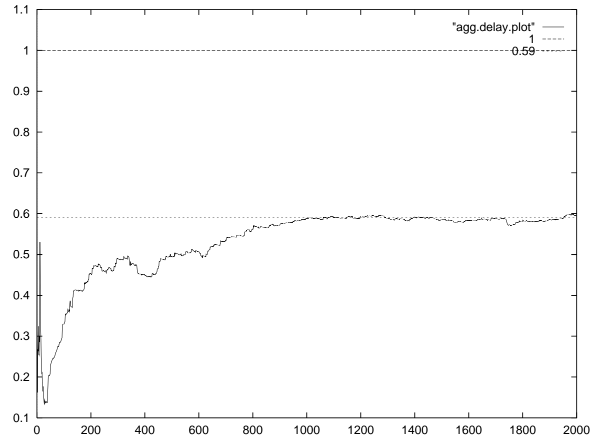

Figure 8: Ratio of average HTTP request delay over 2000 URLs. X-axis represents the 2000 URLs. Y-axis represents the ratios of the average HTTP request delay for all the previous URLs.

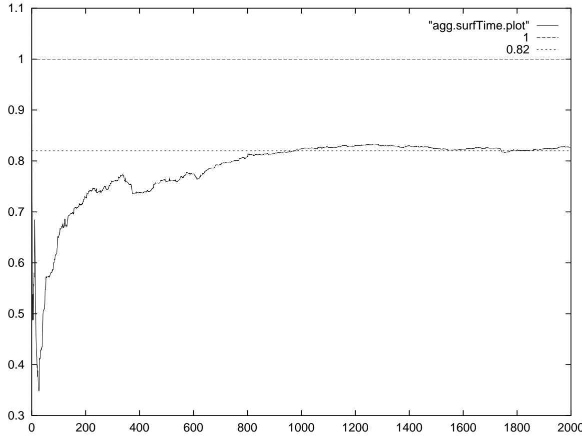

Figure 9: Ratio of Web session times. X-axis represents the 2000 URLs. Y-axis represents the ratios of the total session time for all the previous URLs.

We obtained 2000 random URLs from the Yahoo! Web site and divided them into 100 groups of 20 URLs to represent 100 different sessions. We then ran the *csurf* on the URLs with *squid* [4]. On a different client-server system, we ran *surf* with the proxy system (before the test, we ran surf 5 times for each of the 100 sessions over the proxy system to allow time for the proxy system to learn the access pattern). For both the tests, we set *readspeed* to be 128 chars/second and *impatience* to be 60 seconds. We synchronized the sessions between the two tests so that it cancels out the network activity that is time-based. We also cleared all caches (local and backbone) between sessions.

Figure 2 shows the ratio of each URL's delay time (HTML file together with image file times) plotted on a log scale. 79% of the URLs have a ratio of less than 1. This means that 79% of the URLs took less time with the proxy system. The 21% of the URLs that took equal or longer time can be attributed to the network variance in the Internet. The median ratio was 0.239572.

Figure 3 shows the ratio of each session's HTTP bandwidth. We see that on the average, we get twice the bandwidth, as far as the browser is concerned. Note that this is the mean value.

Figure 4 shows the ratio of the average URL request time per session. We see that on the average, each request takes only about 62% of the time it used to take. This means a faster response to the browser.

Figure 5 shows the ratio of the total network waiting time of each session. We notice that with the proxy system, on the average, we only wait 62.5% of the time we would normally wait in one session. Note that this graph is similar to Figure 4. It is only similar and not exact because of dynamic pages that return different dependents each time e.g. advertisements.

Figure 6 shows the ratios of the total time taken to complete each of the 100 sessions. The times included network waiting time, sleeping time (to simulate reading), and CPU overhead time. We see that even though 21% of the individual URL request times were equal or slower, grouping URLs into sessions amortizes the time, and only 8%-10% of the sessions are slower than before. We also see that on the average (over 100 sessions), we spend only 83% of the time we spend on a Web session when we are using the proxy system.

Finally, we collated the times of the 2000 URLs to see how the proxy system will perform in the long term. It must be noted that the value we get here is actually lower than what we would have gotten had we used a 2000 URL session. This is because we cleared all caches between sessions.

Figure 7 shows that, on the average, the browser observes a bandwidth increase of 1.7 times when the proxy system is being used.

Figure 8 shows that, on the average, for individual HTTP request delay, the browser waits only 59% of the time. As expected, this is roughly the inverse of the bandwidth improvement.

Figure 9 shows that, on the average, the user spends only 82% of the time he/she would have if he/she only used a single backbone proxy.

5.1 Caveats

Pre-fetching is pointless if the user can read at a rate faster than data can arrive at his/her browser. We assume that this is not the case.

In general, we have observed that the profiles are learned for the first time over 5 to 7 sessions. Learning changes in user patterns online is dependent on the weights assigned. In our case, we placed heavy emphasis on a user's history. This means it takes longer for the system to adapt to changes. At the same time, the system is less sensitive to random bursts of visits to certain sites that will not be visited again. That is, we set the system to do long

term adaptation. We found that this model suited our access patterns.

The cache hit ratio, defined as the number of hits over the number of user requests, for the above tests averaged at 62%. The accuracy of pre-fetches, defined as the number of hits over the number of pre-fetches made, averaged at 50%. However, note that *surf* does not visit previous pages. In normal browsing, we have found that the "previous" button is used rather often, and this increases the hit ratio as well as the pre-fetch accuracy. Note that the pre-fetch accuracy can actually be greater than one if a single pre-fetch can service multiple requests. In our own browsing, we found that the hit ratio hovers at around 75% and the pre-fetch accuracy hovers at around 70%. In this case, the low pre-fetch accuracy is due to visits of new sites in daily Web surfing.

Note that all these numbers are specific to our access patterns and for our learned profiles. We expect that these numbers may change for other usage patterns - in particular, the tuning of the constants in determining user profiles played a key role in improving the efficiency of pre-fetching. However, while we believe that significant work needs to be done in order to automatically tune the system to match user access patterns, we do believe that our system can provide perceptible improvements in the experience of Web browsing.

We also noted with interest that our own access patterns have changed as a result of using the proxy system. However, we felt that a user's access pattern will naturally change when the network condition changes. For example, if the network is slow, the user is usually apprehensive about clicking and then waiting. Since our proxy system basically changes the network conditions as far as the browser is concerned, we felt that a change in our access pattern is tolerable.

6 Related Work

Studies on techniques which aim to reduce the latency of Web accesses include LowLat [19] and WebExpress [13]. LowLat differs from our system in that it requires a process to be located near the Web server. WebExpress differs in that it uses file caching, forms differencing, protocol reduction, and the elimination of redundant HTTP header transmission to reduce the bandwidth used. Furthermore, WebExpress multiplexes multiple HTTP requests over a single link to reduce the TCP setup overhead. Currently, our system transmits HTTP documents through standard HTTP/1.0.

Studies into speculative pre-fetch of Web documents include work done by the OCEAN group [6, 9, 8], ICS-FORTH [15], Tenet [17, 18], and Wcol [5]. OCEAN's approach differs in that they use both server initiated pre-fetch as well as client-initiated pre-fetch. Further, they use a Random Walk User Model and a DSP User Model to model usage patterns. ICS-FORTH differs in that they employ a server initiated pre-fetch with the help of a Top-10 Approach. Tenet represents usage pattern on the server through dependency graphs. Similar to our pre-fetch with notification, their server makes the predictions and the client initiates the pre-fetches. Wcol differs from our profile-based pre-fetch in that they parse the HTML files and pre-fetch both the links and the inline images. Wachsberg [20] describes the use of a model similar to ours. A commercial product that does speculative pre-fetch is PeakJet [3].

Studies on geographical push caching [11, 12] by the VINO research group involves server initiated pushing and differs from our client initiated approach.

Studies into collaborative data filtering include Tapestry [10], and FIREFLY [1]. JunkBusters [2] is a proxy server that also filters HTTP requests. Our work is similar to the architecture that Zenel describes for intelligent filtering in low-bandwidth environment in that we make use of an intermediary (proxy).

7 Summary

Users surf the WWW in a regular fashion; thus, it is possible to exploit that information in caching systems. Furthermore, since users spend a non-trivial amount of time reading a page, the time can be used to pre-fetch documents rather than let the network stay idle. This paper described the use of usage profiling, pre-fetching, and filtering techniques in the context of WWW caching.

The usage profiles employed the use of a directed graph to represent the path a user takes in surfing the web. This information is later used by the pre-fetch engine to issue pre-fetch requests to the cache manager. Meanwhile, the HTTP requests and responses are filtered to ensure good use of the available bandwidth.

Using various heuristics described in the paper, we implemented a proxy system that improved the network performance from the perspective of the browser. This had the effect of reducing the overall time spent on web sessions.

References

[1] FireFly. http://www.firefly.com/.

[2] Junkbusters. http://www.junkbusters.com.

[3] PeakJet. http://www.peak-media.com/.

[4] Squid Internet Object Cache. http://squid.nlanr.net/Squid/.

[5] WWW Collector - The Prefetching Proxy Server for WWW. http://shika.aist-nara.ac.jp/products/wcol/wcol.html.

[6] Azer Bestavros. Using Speculation to Reduce Server Load and Service Time on the WWW. *Proceedings of CIKM'95: The 4th ACM International Conference on Information and Knowledge Management*, Nov 1995.

[7] Vaduvur Bharghavan and V. Gupta. A Framework for Application Adaptation in Mobile Computing Environments. *Proceedings of the IEEE Computer Software and Applications Conference, Washington D.C.*, Aug 1997.

[8] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW Client-Based Traces. *Technical Report TR-95-010, Boston University, CS Dept, Boston, MA 02215*, Apr 1995.

[9] Carlos R. Cunha and Carlos F. B. Jaccoud. Determining WWW User's Next Access and Its Application to Pre-Fetching. *Proceedings of ISCC'97: The Second IEEE Symposium on Computers and Communications*, Jul 1997. (Extended version).

[10] D. Goldberg, D. Nichols, B.M. Oki, and D. Terry. Using Collaborative Filtering to Weave an Information Tapestry. *Communications of the ACM, Volume 35, Number 12*, Dec 1992.

[11] James Gwertzman and Margo Seltzer. An Analysis of Geographical Push-Caching. http://www.eecs.harvard.edu/vino/web/server.cache/icdcs.ps.

[12] James Gwertzman and Margo Seltzer. The Case for Geographical Push-Caching. *Proceedings of the Fifth Annual Workshop on Hot Operating Systems*, May 1995.

[13] Barron C. Housel and David B. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. *MobiCom'96 Demonstration Session*, Nov 1996. http://www.networking.ibm.com/art/artwewp.htm.

[14] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the CODA File System. *ACM Transactions on Computer Systems, Vol. 10, No. 1*, Feb 1992.

[15] Evangelos P. Markatos and Catherine E. Chronaki. A Top-10 Approach to Prefetching on the Web. *Technical Report No. 173, ICS-FORTH, Heraklion, Crete, Greece.*, Aug 1996. http://www.ics.forth.gr/proj/arch-vlsi/www.html.

[16] B.D. Noble and M. Satyanarayanan et al. Agile Application-Aware Adaptation for Mobility. *Proceedings of the ACM Symposium on Operating Systems Principles, St. Malo, France*, Oct 1997.

[17] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving World Wide Web Latency. *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, Jul 1994.

[18] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *ACM SIGCOMM Computer Communication Review*, Jul 1996.

[19] Joe Touch. The LowLat Project. http://www.isi.edu/lowlat/, 1996.

[20] Stuart Wachsberg, Thomas Kunz, and Johnny Wong. Fast World-Wide Web Browsing over Low-Bandwidth Links. http://ccnga.uwaterloo.ca/˜ sbwachsb/paper.html, 1996.

[21] Bruce Zenel and Dan Duchamp. Intelligent Communication Filtering for Limited Bandwidth Environments. *Proceedings of the fifth Workshop on Hot Topics in Operating Systems Rosario, Washington*, May 1995.