Improving World Wide Web Latency *

Venkata N. Padmanabhan
padmanab@CS.Berkeley.EDU

Report No. UCB/CSD-95-875
Computer Science Division
University of California at Berkeley
May, 1995

Abstract

The HTTP protocol, as currently used in the World Wide Web, uses a separate TCP connection for each file requested. This adds significant and unnecessary overhead, especially in the number of network round trips required. We analyse the costs of this approach and propose simple modifications to HTTP that, while interoperating with unmodified implementations, avoid the unnecessary network costs. We have implemented our modifications, and our measurements show that they dramatically reduce latencies. We have also investigated the effectiveness of a scheme to mask network latency by prefetching files likely to be requested next, while the user is browsing through the currently displayed page. Our results indicate a significant benefit from prefetching at the cost of an increase in network traffic.

1 Introduction

People use the World Wide Web (WWW) because it gives quick and easy access to a tremendous variety of information in remote locations. Users do not like to wait for their results; they tend to avoid or complain about Web pages that take a long time to retrieve. That is, users care about Web latency.

Perceived latency comes from several sources. Web servers can take a long time to process a request, especially if they are overloaded or have slow disks. Web clients can add delay if they do not quickly parse the retrieved data and display it for the user. Latency caused by client or server slowness, however, can in principle be reduced simply by buying a faster computer, or faster disks, or more memory.

Web retrieval delay also depends on network latency. The Web is useful precisely because it provides remote access, and transmission of data across a distance takes time. Some of this delay depends on bandwidth; one cannot retrieve a 1 Mbyte file across a 1 Mbit/sec link in less than 8 seconds. You can in principle reduce this time by buying a higher-bandwidth link. But much of the latency seen by Web users comes from propagation delay: the speed of light is a constant. You cannot send

*This work was supported in part by Digital Equipment Corporation's Western Research Laboratory.

even a single bit of information over, say, 3000 miles in less than 16 msec, no matter how much money you have.

In practice, most retrievals over the World Wide Web result in the transmission of relatively small amounts of data. (A randomly chosen sample of 200,000 HTTP retrievals shows a mean size of 12925 bytes and a median size of just 1770 bytes; excluding 12727 zero-length retrievals, the mean was 13767 bytes and the median 1946 bytes.) This means that bandwidth-related delay may not account for much of the perceived latency. For example, transmission of 20 Kbytes over a T1 (1.544 Mbit/sec) link takes about 100 msec. For comparison, the best-case small-packet round-trip time (RTT) over a coast-to-coast (US) Internet path is about 70 msec; at least half of this delay depends on the speed of light and is therefore intrinsic. When the network path is congested, queuing delays can increase the RTT by large factors.

This means that, in order to avoid network latency, we must avoid the cost of round trips through the network. Unfortunately, the Hypertext Transport Protocol (HTTP) [1], as it is currently used in the Web, incurs many more round trips than necessary.

In the first part of this report, we analyse that problem, and show that almost all of the unnecessary round trips may be eliminated by surprisingly simple changes to the HTTP protocol and its implementations. We then present results measured using prototype implementations, which confirm that our changes result in significantly improved response times. This material is based on our earlier paper [11].

During the course of our work, Simon Spero published an analysis of HTTP [15], which reached conclusions similar to ours. However, we know of no other project, besides our own, that has implemented the consequent modifications to HTTP, or that has quantified the results.

Another approach for avoiding the cost of network round trips is to "hide" them from the user. One way of doing this is to prefetch Web pages that the user is likely to access next, while the user is browsing through the currently displayed page. Then, if the user does request one of the prefetched pages, it will probably already be in the local site's cache. So, the network round trips incurred while fetching the page from the server are hidden from the user.

While there have been studies of prefetching in other settings, we are not aware of any other work in the particular context of the WWW. It is clear that the effectiveness of prefetching critically depends on how good the predictions we make are. We use a scheme based on that proposed by Griffioen and Appleton [6] in the context of file systems, with a few modifications. Details of the our scheme and a discussion of our results are the subject of the second half of this report.

The rest of this report is organized as follows. In section 2, we briefly discuss the basics of HTTP that are needed to understand the rest of this report. Section 3 deals with the way HTTP uses network connections, and analyses some of its drawbacks. In sections 4 and 5, we present two different changes to HTTP that we have implemented. An experimental evaluation of the reduction in latency due to these modifications is presented in section 6. Then we move on to a discussion of prefetching in section 7. In section 8, we present an evaluation of our prefetching scheme based on log-driven simulations. We summarize our work in section 9, and present our conclusions in section 10.

2 HTTP protocol elements

We briefly sketch the HTTP protocol, to provide sufficient background for understanding the rest of this report. We omit a lot of detail not directly relevant to HTTP latency.

The HTTP protocol is layered over a reliable bidirectional byte stream, normally TCP [12]. Each HTTP interaction consists of a request sent from the client to the server, followed by a response sent from the server to the client. Requests and responses are expressed in a simple ASCII format.

The precise specification of HTTP is in a state of flux. Most existing implementations conform to [1]. A revision of the specification is in progress.

An HTTP request includes several elements: a *method* such as GET, PUT, POST, etc.; a Uniform Resource Locator (URL); a set of Hypertext Request (HTRQ) headers, with which the clients specifies things such as the kinds of documents it is willing to accept, authentication information, etc; and an optional Data field, used with certain methods such as PUT.

The server parses the request, then takes action according to the specified method. It then sends a response to the client, including a status code to indicate if the request succeeded, or if not, why not; a set of object headers, meta-information about the "object" returned by the server, optionally including the "content-length" of the response; and a Data field, containing the file requested, or the output generated by a server-side script.

Note that both requests and responses end with a Data field of arbitrary length. The HTTP protocol specifies three possible ways to indicate the end of the Data field, in order of declining priority:

1. If the "Content-Length" field is present, it indicates the size of the Data field and hence the end of the message.
2. The "Content-Type" field may specify a "boundary" delimiter, following the syntax for MIME multipart messages [2].
3. The server (but not the client) may indicate the end of the message simply by closing the TCP connection after the last data byte.

Later on we will explore the implications of the message termination mechanism.

3 Message and packet exchanges in HTTP

We now look at the way the interaction between HTTP clients and servers appears on the network, with particular emphasis on how this affects latency.

Figure 1 depicts the exchanges at the beginning of a typical interaction, the retrieval of an HTML document with at least one uncached inline image. In this figure, time runs down the page, and long diagonal arrows show packets sent from client to server or vice versa. These arrows are marked with TCP packet types; note that most of the packets carry acknowledgements, but the packets marked ACK carry *only* an acknowledgement and no new data. FIN and SYN packets in this example never carry data, although in principle they sometimes could.

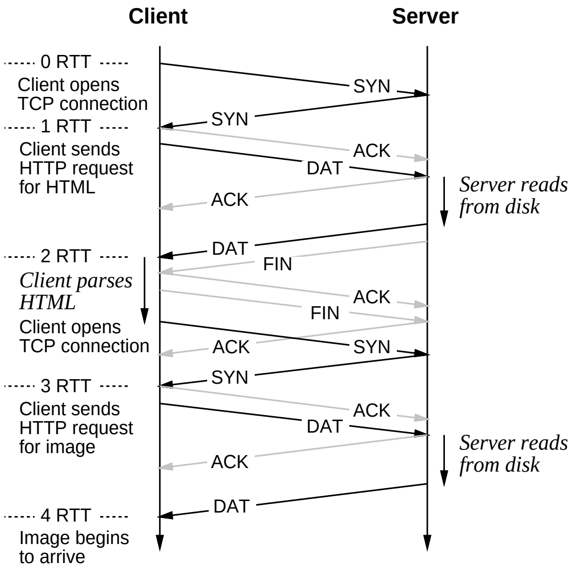

Figure 1: *Packet exchanges and round-trip times for HTTP*

Shorter, vertical arrows show local delays at either client or server; the causes of these delays are given in italics. Other client actions are shown in roman type, to the left of the Client timeline.

Also to the left of the Client timeline, horizontal dotted lines show the "mandatory" round trip times (RTTs) through the network, imposed by the combination of the HTTP and TCP protocols. These mandatory round-trips result from the dependencies between various packet exchanges, marked with solid arrows. The packets shown with gray arrows are required by the TCP protocol, but do not directly affect latency because the receiver is not required to wait for them before proceeding with other activity.

The mandatory round trips are:

1. The client opens the TCP connection, resulting in an exchange of SYN packets as part of TCP's three-way handshake procedure.
2. The client transmits an HTTP request to the server; the server may have to read from its disk to fulfill the request, and then transmits the response to the client. In this example, we assume that the response is small enough to fit into a single data packet, although in practice it might not. The server then closes the TCP connection, although if it has sent a Content-length field, the client need not wait for the connection termination before continuing.
3. After parsing the returned HTML document to extract the URLs for inline images, the client opens a new TCP connection to the server, resulting in another exchange of SYN packets.
4. The client again transmits an HTTP request, this time for the first inline image. The server obtains the image file, and starts transmitting it to the client.

Therefore, the earliest time at which the client could start displaying the first inline image would be four network round-trip times after the user requested the document. Each additional inline image requires at least two further round trips. In practice, with networks of finite bandwidth or documents larger than can fit into a small number of packets, additional delays will be encountered.

3.1 Other inefficiencies

In addition to requiring at least two network round trips per document or inline image, the HTTP protocol as currently designed has other inefficiencies.

Because the client sets up a new TCP connection for each HTTP request, there are costs in addition to network latencies:

- Connection setup requires a certain amount of processing overhead at both the server and the client. This typically includes allocating new port numbers and resources, and creating the appropriate data structures. Connection teardown also requires some processing time, although perhaps not as much.
- The Web clearly needs some form of authentication, and perhaps also encryption for privacy and data integrity. It would be quite expensive to re-authenticate principals on each HTTP request.
- Although the TCP connections may be active for only a few seconds, the TCP specification requires that the host which closed the connection remember certain per-connection information for four minutes [12] (although many implementations do violate this specification and use a much shorter timer.) A busy server could end up with its tables full of connections in this "TIME-WAIT" state, either leaving no room for new connections, or at least imposing excessive connection table management costs.

Current HTTP practice also means that most of these TCP connections carry only a few thousand bytes of data. As we noted earlier, one sample showed a mean document size of about 13K bytes, and a median of under 2K bytes. About 45% of these retrievals were for Graphics Interchange Format [4] (GIF) files, used for both inline and out-of-line images. This sub-sample showed a slightly larger mean and a slightly smaller median; our guess is that the very large GIF files were not inline images. The proposed use of JPEG for inline images will tend to reduce these sizes.

Unfortunately, TCP does not fully utilize the available network bandwidth for the first few round-trips of a connection. This is because modern TCP implementations use a technique called *slow-start* [7] to avoid network congestion. The slow-start approach requires the TCP sender to open its "congestion window" gradually, doubling the number of packets each round-trip time. TCP does not reach full throughput until the effective window size is at least the product of the round-trip delay and the available network bandwidth. This means that slow-start restricts TCP throughput, which is good for congestion avoidance but bad for short-connection completion latency.

3.2 Quantifying TCP connection overheads

We performed a set of simple experiments to illustrate this effect. We used a simple client program, which opens a connection to a server, tells the server how many bytes it wants, and then reads and discards that many bytes from the server. The server, meanwhile, generates the requested number of bytes from thin air, writes them into the connection, and then closes the connection. This closely approximates the network activity of a single-connection HTTP exchange.

We measured three configurations: a "local" server, with a round-trip time of under 1 msec, and 1460-byte TCP segments (packets); a "remote" server, across the width of the U.S., with a

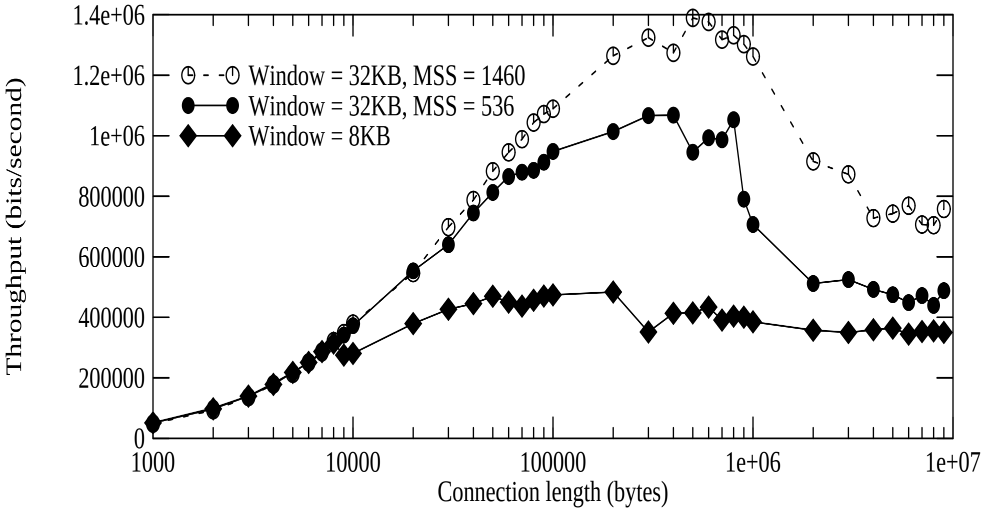

Figure 2: *Throughput vs. connection length, RTT = 70 ms*

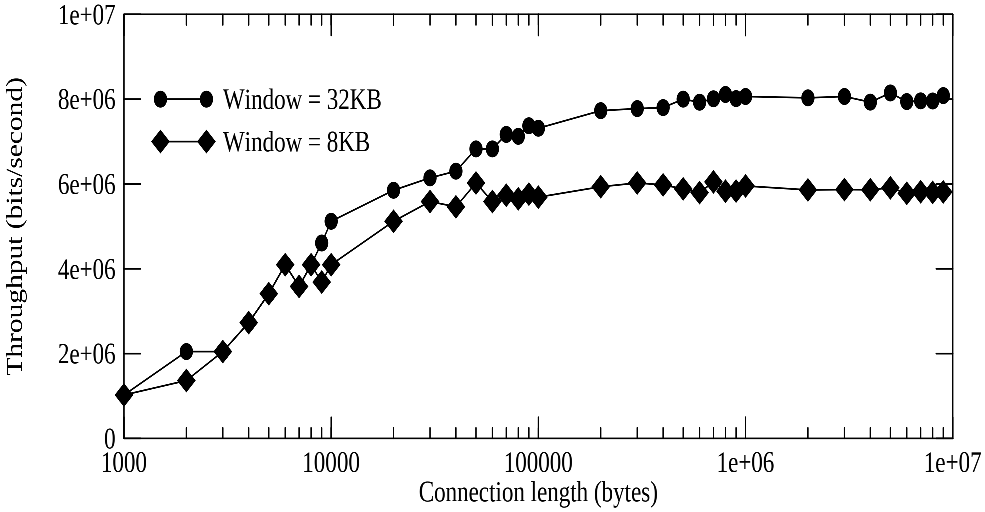

Figure 3: *Throughput vs. connection length, RTT near 0 ms*

best-case RTT of 70 msec, and 1460-byte TCP segments; and the same remote server, but using 536-byte TCP segments. This last configuration reflects a widely-used technique meant to avoid IP fragmentation [3]; more modern practice could use the full available packet size [9]. In each configuration, we measured throughput for a large variety of connection lengths and a few popular TCP buffer (maximum window) sizes. We did ten trials for each set of parameters, and plotted the throughput of the best trial from each set (to help eliminate noise from other users of the network). Figure 2 shows the results for the remote (70 msec) server; figure 3 shows the local-server results. Note that the two figures have different vertical scales.

Figure 2 shows that, in the remote case, using a TCP connection to transfer only 2 Kbytes results in a throughput less than 10% of best-case value. Even a 20 Kbyte transfer achieves only about 50% of the throughput available with a reasonable window size. This reduced throughput translates into increased latency for document retrieval. The figure also shows that, for this 70 msec RTT, use of too small a window size limits the throughput no matter how many bytes are transferred.

We also note a significant decline in throughput over this path for transfers longer than about

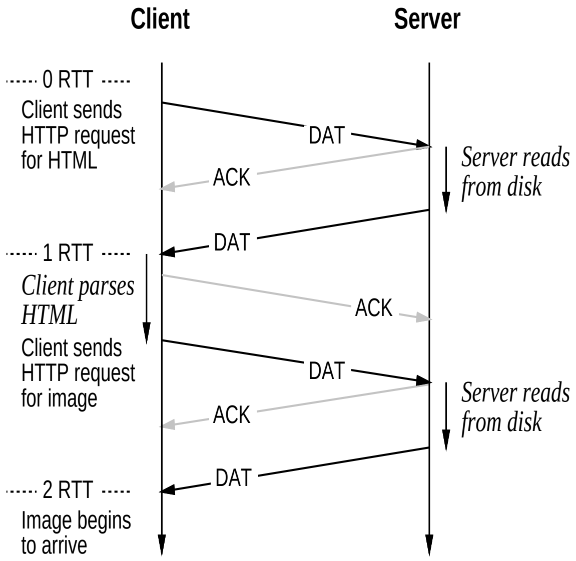

Figure 4: *Packet exchanges for HTTP with persistent connections*

500 Kbytes. This is caused by a breakdown in the TCP congestion-avoidance algorithm, as the congestion window becomes larger than the router's queue limit. Note, however, that this problem arises only for transfers orders of magnitude larger than typical HTML documents or inline images. The dotted curve shows that by using a larger maximum segment size (MSS) (and hence fewer packets for the same congestion window size), we can obtain somewhat better throughput for lengthy transfers.

Even in the local case, per-connection overhead limits throughput to about 25% of capacity for transfers of 2 Kbytes, and about 70% of capacity for transfers of 20 Kbytes. In this case, slow-start is not involved, because the ULTRIX implementation of TCP avoids slow-start for local-net connections.

4 Persistent Connections

Since the short lifetimes of HTTP connections causes performance problems, we tried the obvious solution: use a single, long-lived connection for multiple HTTP transactions. The connection stays open for all the inline images of a single document, and across multiple HTML retrievals. This avoids almost all of the per-connection overhead, and also should help avoid the TCP slow-start delays.

Figure 4 shows how this change affects the network latencies. This depicts the same kind of retrieval as did figure 1, except that the client already has a TCP connection open to the server, and does not close it at the end of an HTTP exchange. Note that the first image arrives after just two round trips, rather than four. Also, the total number of packets is much smaller, which should lead to lower server load. Finally, since the ratio of connection lifetime to the length of the TIME-WAIT state is higher, the server will have far fewer TCP connections (active or inactive) to keep track of.

In order to use long-lived connections, we had to make simple changes to the behavior of both client and server. The client can keep a set of open TCP connections, one for each server with which it has recently communicated; it can close connections as necessary to limit its resource consumption. Even a client capable of maintaining only one open connection can benefit, by simply not closing the connection until it needs to contact a different server. It is quite likely that two successive HTTP interactions from a single client will be directed to the same server (although we have not yet quantified this locality).

The server also keeps a set of open TCP connections. Some HTTP servers fork a new process to handle each new HTTP connection; these simply need to keep listening for further requests on the open connection after responding to a request, rather than closing the connection and terminating. This not only avoids connection overhead on each request; it also avoids the cost of forking a new process for each request. Other servers manage multiple threads within a single process; these need to keep a set of TCP connections open, and listen for new requests on all of them at once. Neither approach is especially hard to implement.

With either approach, the server may need to limit the number of open connections. For example, it could close the oldest connection when the number of open connections exceeds a threshold (preferably not in the middle of responding to a request on this connection). For multiple-process UNIX-based servers, for example, the parent process could send its oldest child a signal (interrupt) saying "exit when you next become idle." Since servers may terminate connections at arbitrary times, clients must be able to reopen connections and retry requests that fail because of this.

4.1 Detecting end-of-transmission

As we mentioned in section 2, HTTP provides three ways for the server to indicate the end of the Data field of its responses: a Content-length field, a boundary delimiter specified in the Content-type field, or termination of the TCP connection. This presents a problem when the response is generated by a script, since then the server process does not know how long the result will be (and so cannot use Content-length), nor does it know the format of the data (and so cannot safely use a predetermined delimiter sequence).

We considered several approaches in which the data stream from the script is passed through the server on its way to the client:

Boundary delimiter The server can safely insert a boundary delimiter (perhaps as simple as a single character) if it can examine the entire data stream and "escape" any instance of the delimiter that appears in the data (as is done in the Telnet protocol [13]). This requires both the server and client to examine each byte of data, which is clearly inefficient.

Blocked data transmission protocol The server could read data from the script and send it to the client in arbitrary-length blocks, each preceded by a length indicator. This would not require byte-by-byte processing, but it would involve a lot of extra data copying on the server, and would also require a protocol change.

Store-and-forward The server can read the entire output of the script into temporary storage, then measure the length and generate a response with a correct Content-length field. This requires extra copying and may be infeasible for large responses, but does not require a protocol change.

None of these approaches appealed to us, because they all imposed extra work on the server (and possibly the client).

We also considered using a separate control connection, as in FTP, via which the server could notify the client of the amount of data it had transmitted on the data connection. This, however, might be hard to implement and would double the amount of connection overhead, even in cases where it is not needed.

We chose to stick with a simple, hybrid approach in which the server keeps the TCP connection open in those cases where it can use the Content-length or boundary delimiter approaches, and closes the connection in other cases (typically, when invoking scripts). In the common case, this avoids the costs of extra TCP connections; in the less usual case, it may require extra connection overhead but does not add data-touching operations on either server or client, and requires no protocol changes.

4.2 Compatibility with older versions of HTTP

We wanted our modified client to transparently interoperate with both standard and modified HTTP servers, and we wanted our modified server to interoperate with both sorts of clients. This means that the modified client has to inform the server that the TCP connection should be retained, and in such a way that an unmodified server can ignore the request. This could be done by introducing a new field in the HTRQ headers (see section 2) sent in the client's request. For example, a future version of the HTTP specification could define a `hold-connection` field as a part of the HTTP request.

For our experiments, we simply encoded this information in a new HTRQ header field; such unrecognized fields must be ignored by unmodified servers.

5 Pipelining requests

Even with long-lived TCP connections, simple implementations of the HTTP protocol still require at least one network round trip to retrieve each inline image. The client interacts with the server in a stop-and-wait fashion, sending a request for an inline image only after having received the data for the previous one.

There is no need for this, since the retrieval of one image in no way depends on the retrieval of previous images. We considered several ways in which client requests could be pipelined, to solve this problem.

5.1 The GETALL method

When a client does a GET on a URL corresponding to an HTML document, the server just sends back the contents of the corresponding file. The client then sends separate requests for each inline image. Typically, however, most or all of the inline images reside on the same site as the HTML document, and will ultimately come from the same server.

We propose adding to HTTP a GETALL method, specifying that the server should return an

HTML document and all of its inline images residing on that server. On receiving this request, the server parses the HTML file to find the URLs of the images, then sends back the file and the images in a single response. The client uses the Content-length fields to split the response into its components.

The parsing of HTML documents is an additional load for the server. However, it is not expected to be too expensive, especially compared to the cost of parsing many additional HTTP requests. Or, the server could keep a cache of the URLs associated with specific HTML files, or even a precomputed database.

One can implement the GETALL method using an ordinary GET, using an additional field in the HTRQ header to indicate that the client wants to perform a GETALL. This allows a modified client to interoperate with an unmodified server; in this case, the client notes that it has not received all the images when the connection is closed, and simply retrieves them the traditional way.

5.2 The GETLIST method

HTTP clients typically cache recently retrieved images, to avoid unnecessary network interactions. A server has no way of knowing which of the inline images in a document are in the client's cache. Since the GETALL method causes the server to return all the images, this seems to defeat the purpose of the client's image cache (or of a caching relay [5]). GETALL is still useful in situations where the client knows that it has no relevant images cached (for example, if its cache contains no images from the server in question).

Therefore, we propose adding a GETLIST mechanism, allowing a client to request a set of documents or images from a server. A client can use a GET to retrieve an HTML file, then use the GETLIST mechanism to retrieve in one exchange all the images not in its cache. (On subsequent accesses to the same HTML file, the client can request the HTML and all images in one message.)

Logically, a GETLIST is the same as a series of GETs sent without waiting for the previous one to complete. We in fact chose to implement it this way, since it requires no protocol change and it performs about the same as an explicit GETLIST would.

Our client uses a simple heuristic to decide between using GETALL and GETLIST. When it accesses a document for the first time, it uses GETALL, even though there is a small chance that its cache contains some of the inline images. It keeps a cache, listing for each known image URL the URL of the document that contained it, so the client can distinguish between documents for which it definitely has cached images, and those for which it probably does not (some images may be referenced by several documents). We have not done sufficient studies of actual HTTP usage to determine if this heuristic results in excessive retrievals of cached images.

6 Experimental Results

In this section, we report on simple experiments to measure the effect of the new protocol on observed latency.

We implemented our protocol changes by modifying the Mosaic V2.4 client, and the NCSA `httpd` V1.3 server. Both client and server were run on MIPS-based DECstation systems, running the

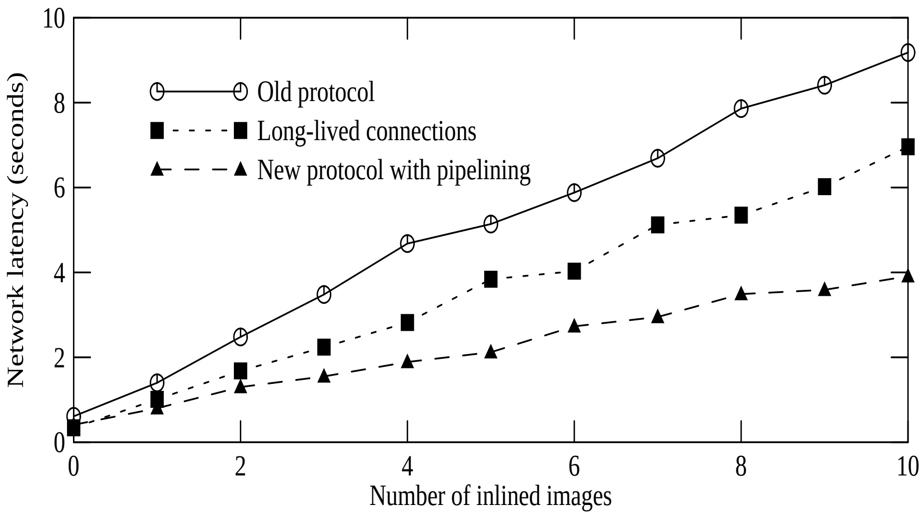

Figure 5: *Latencies for a remote server, image size = 2544 bytes*

ULTRIX operating system. The Mosaic client ran on a DECstation 3100 with 24M bytes of memory; this is a relatively slow system and so we measured network retrieval times, not including the time it took to render images on the display.

In our experiments, we measured the time required to load a document and all of its inline images. We created documents with different numbers of inline images, and with images of various sizes. We did these measurements for both a local server, accessed via a 10 Mbit/sec Ethernet with a small RTT, and a remote server, access via a 1.544 Mbit/sec T1 link with a best-case RTT of about 70 msec.

Figure 5 shows how load time depends on the number of images retrieved, using 2544-byte images and the remote server. Our modified HTTP protocol cuts the latency by more than half, about what we expected from the reduced number of round trips. These images are about the median size observed in our traces, and so we do expect to see this kind of speedup in practice. While more than half of the improvement comes from pipelining, even without pipelining long-lived connections do help.

Figure 6 shows that load time depends on the number of images retrieved. In this case, using 45566-byte images and the remote server, the new protocol improves latency by about 22%; less than in figure 5 but still noticeable. In this case, the actual data transfer time begins to dominate the connection setup and slow-start latencies.

We summarize our results for trials using the remote server and various image sizes in figure 7 and using the local server in figure 8. These graphs show the relative improvement from the modified protocol, including pipelining. In general, the benefit from the modified protocol is greatest for small images and for at least a moderate number of images.

Even though the round-trip time to the local server is much smaller than that to the remote server, the modified protocol still provides significant improvements for local retrievals. For the local case, long-lived connections without pipelining reduces latency by only about 5% to 15%; this implies that the reduction in round trips is more important that the per-connection overheads.

Note that for the relatively small transfers associated with the median image size, slow-start latencies cannot account for much of the delay; in these tests, the TCP MSS was 1460 bytes, and traces

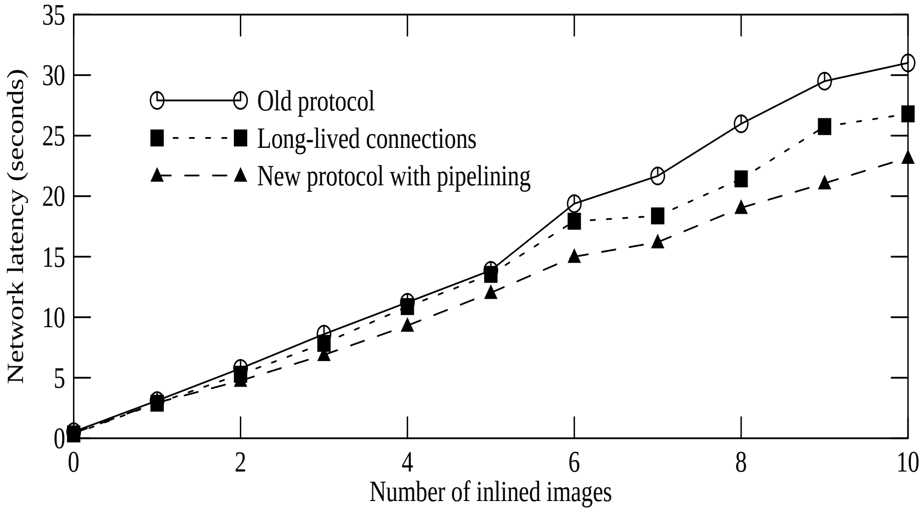

Figure 6: *Latencies for a remote server, image size = 45566 bytes*

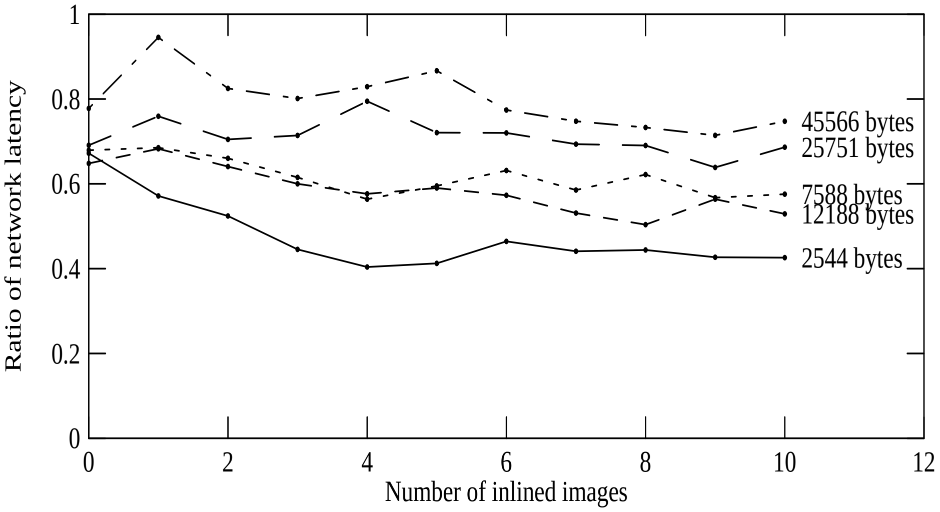

Figure 7: *Latency improvements for a remote server*

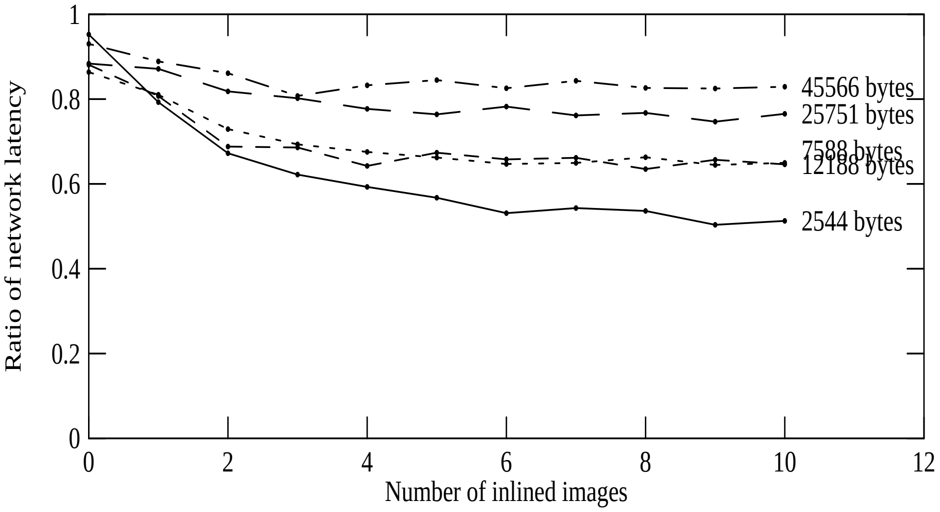

Figure 8: *Latency improvements for a local server*

showed that slow-start did not limit the window size.

7 Prefetching files

Thus far, we have discussed modifications to the HTTP protocol as a way of substantially reducing latency. There are, of course, other possibilities for latency reduction. We now discuss one idea which is motivated by the following observation. The usual way users browse the Web is to follow hyperlinks, going from one Web page to another, which is generally on the same server. Typically, there is a pause after each page is loaded, while the user is browsing through the displayed material. This time could be used by the client to prefetch files that are likely to be accessed soon, thereby avoiding network latency if and when those files are actually requested.

The basic idea is for the server to make predictions about the likelihood that a particular Web page will be accessed next, and convey this to the client. The client program can then decide whether or not to actually prefetch the page. This partitioning of work between the server and clients is natural because, on the one hand, the server has the opportunity to observe the pattern of accesses from several clients and use this information to make intelligent predictions, while on the other hand the clients are in the best position to decide whether or not they want to expend the resources (CPU time, memory, network bandwidth, etc.) needed to prefetch data. The human user is totally oblivious of all this.

As an aside, we note that the server could prefetch files from disk into memory, independent of clients. However, the benefit of this would be limited because of the dominance of network latency over disk latency, especially in a wide-area context. So in this study, we only investigated prefetching from the server to clients, across the network.

7.1 Model for prefetching

We now describe our model for how prefetching will be done. There are two types of user-level processes running on the server machine. One is the regular HTTP server process, `httpd`, with some additions as described below. This process forks off child processes to handle incoming requests from clients. The other process is the *prefetch daemon*, `prefetchd`, which makes prefetching decisions. There is only one `prefetchd` per server, not a new one for each client request or for each client.

On receiving a request from a client, `httpd` passes on the identity of the client and the files requested to `prefetchd`. In this context, we are only concerned with file accesses, so `prefetchd` only looks at client requests using the GET method or its variants (such as GETALL or GETLIST). The `prefetchd` uses the prediction algorithm described in the next subsection to determine candidate files for being prefetched based on the likelihood of their being accessed soon, and conveys this information to the concerned client. This can be piggy-backed on the reply sent by `httpd` to the client, in a special field.

The client, essentially an enhanced version of a browser such as Mosaic, looks at the reply sent by the server and decides whether or not to prefetch the files. It could base its decision on a variety of factors such as the contents of its local cache (which might already contain the file), the current CPU load, its current mode of operation (such as image loading being turned off), etc. In fact, the client process could use such feedback information from the system it is running on, to tune the

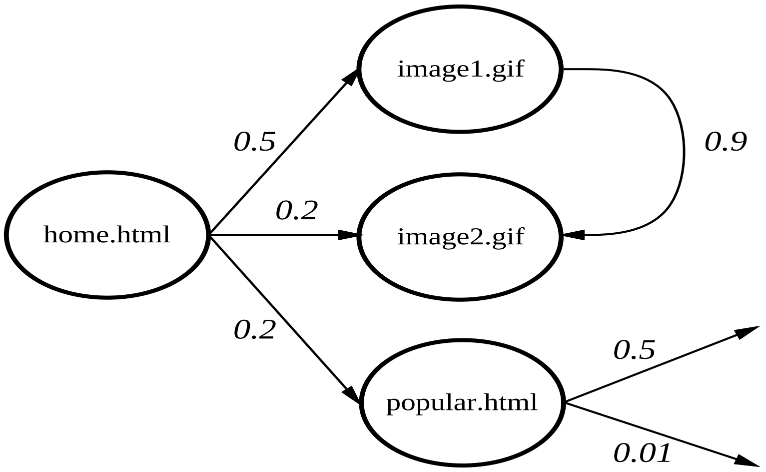

Figure 9: *A small portion of a dependency graph*

"aggressiveness" with which it prefetches data.

Once the client has decides to prefetch a file, it sends a prefetch request to the server. In the request it also indicates that it is prefetching data and not requesting data that the user has asked for. This information can potentially be used by the server in a variety of ways. `Prefetchd` could decide not to do any further prefetching computation based on this request since it is itself a prefetch request. Also if multiple requests are being scheduled in any way, this request could be assigned a lower priority than the more immediate fetches.

7.2 Prediction algorithm

Our prediction algorithm is based on that described by Griffioen and Appleton [6]. However, there are a few noteworthy differences. First, while their scheme was designed for use by the operating system to prefetch files from disk into the file system cache, our model is a distributed one with user-level processes at the server and clients managing prefetching across the network, into the client's cache. So our scheme does not require any kernel modifications.

Second, the scheme described in [6] does not try to maintain a distinction between accesses by different processes (the clients in the context of a file system). This could cause it to (incorrectly) think of independent accesses (by different processes) that just happen to occur close together in time, as related. As we explain below, our scheme avoids this problem of false dependencies.

The prediction algorithm is based on the idea of constructing a *dependency graph* that depicts the pattern of accesses to different files stored at the server. The graph has a node for every file that has ever been accessed. One could, however, decide to prune it by deleting nodes that have not been accessed for a long time. There is an arc from node A to B if and only if, at some point in time, B was accessed within w accesses after A, where w is the *lookahead window* size. The weight on the arc is the ratio of the number of accesses to B within a window after A, to the number of accesses to A itself. Figure 9 depicts a small portion of a dependency graph.

The dependency graph is dynamically updated as the server receives new requests. This is done by the prefetch daemon, `prefetchd`, which receives information about requests from each child `httpd` forked off by the server. It maintains a circular buffer of size equal to the window size w for each client (more precisely, each client host, because we believe that it is a reasonable approximation to assume that there is not more that one Web client running on a host) that is currently connected to this server. When it receives a new request from one of the server processes, it inserts the id of the file accessed into the corresponding circular buffer. Only the entries within a ring-buffer are considered related, so the corresponding arcs in the dependency graph are updated. This logically separates out accesses by different clients and thereby avoids the problem of false dependencies.

The prefetch daemon bases its prefetching decisions on the dependency graph. If the arc from A to B has a high weight, it means that, whenever A is accessed, there is a good chance of B being accessed soon afterwards. So in such a case it would make sense to prefetch B. In general, the daemon would declare B to be a candidate for prefetching if the arc from A to B has a weight higher than the *prefetch threshold* p.

7.3 Some Issues

We have implemented the prefetch daemon, and have made the necessary changes to `httpd` to communicate information on accesses to `prefetchd` through a UNIX pipe. In case of a GETALL or GETLIST request, the modified `httpd` will convey this fact to `prefetchd` so that the latter is aware that *all* the files corresponding to the GETALL or GETLIST have already been sent to the client and hence need not be considered as candidates for being prefetched at this time. We have not yet implemented the client-server communications interface and the client-side support for prefetching.

There is an issue of how the lookahead window is to be managed when there are multiple accesses to the same file within a window. For instance, consider a window size of 10 and the sequence of accesses $ABB \cdots A \cdots A \cdots ABB$, where $\cdots$ denotes gaps much larger than the window size. If we counted the multiple occurences of B within a window, then the weight of the arc from A to B would be $4/4 = 1$. However, this does not reflect the dependency between accesses to A and B correctly because, in fact, 50% of the time, B does *not* follow A within a window. Caching at the clients should eliminate such multiple accesses, but they happen sometimes, typically because the data pointed to by a URL (B in this case) is updated regularly. We ignored such multiple accesses to the same file within a window while computing the weights on arcs.

8 Experimental Evaluation of Prefetching

In this study, we have chosen to evaluate the usefulness of our prefetching scheme using log-driven simulation. The prefetch daemon can run in a simulation mode where it uses access logs of Web servers as input, and computes metrics like the miss-rate for a client (i.e., the fraction of accesses that are to blocks (of files) not already in the client's cache), the average retrieval time per file, the increase in the amount of network traffic due to prefetching, and so on. The simulator allows changing various parameters, such as the prefetch threshold, p, the lookahead window size, w, the maximum number of URL's that `prefetchd` can predict as candidates for prefetching at any one time, i (standing for the amount of "intelligence" the server can convey to the clients), the size of the client cache, c, and so on.

We used access logs from the commercial Web server of Digital Equipment Corporation for driving our simulations. This was a regular `httpd` server from NCSA, so there were no GETALL or GETLIST accesses. For each set of parameter values, the simulator used the first 50000 access log entries just to gain sufficient "intelligence" by building up its dependency graph, without simulating prefetching. It then used the next 70000 entries to simulate the working of a real system with prefetching predictions and updates to the dependency graph. It also simulated an LRU cache at each client to determine the miss rates, average retrieval time, and the fractional increase in network traffic due to prefetching. We assumed that a client always prefetches files which the server advises it to, except when the file is already present in the client's local cache.

The parameters were varied as follows. The prefetch threshold, p, took values from 0.0 through 1.1, increasing in steps of 0.1. The weight on an arc in the dependency graph can never exceed 1.0 (since, as explained earlier, duplicates within a window are ignored), so setting it to 1.1 corresponded to no prefetching. The lookahead window, w, was varied from the minimum value 2 (corresponding to looking ahead just one step) to 10. The parameter i was varied from 1 to 2, and was also set to ∞, which corresponded to there being no limit on the number of URLs that can be predicted for being prefetched at any one time.

The *miss rate* was computed by counting the total number of block misses on real accesses (not prefetches), and dividing by the total number of blocks accessed. The block size was set to 8 KB, and was used as the smallest unit for caching at the clients. The size of the cache at each client host was set to 4 MB. The *average access time per file* was computed using the model described in section 8.2.1. The ratio of the amount of network traffic with prefetching to that without prefetching (which we call the *fractional traffic increase*) was used as a measure of increased network traffic due to prefetching.

We now discuss the results we obtained based on two different models of the system.

8.1 A simple model

In this case, we model the prefetching as happening instantaneously in the following sense:

- Prefetching activity does not delay the servicing of regular fetches from the server.
- A file that is prefetched appears instantly in the client's cache, so if the very next request from the client is for that file, it will result in a hit.

While these assumptions are certainly not realistic, they simplify the model and allow us to determine a useful upper bound on the benefit derived from prefetching.

Figure 10 plots the aggregate miss rate for all client caches for different values of the prefetch threshold, p, and the lookahead window size, w. We see the general trend that the miss rate increases as p increases, because of the decreasing aggressiveness with which prefetching is done. However, for large w and small p, the miss rate worsens with more aggressive prefetching, presumably because a large number of files, even those with very little likelihood of being accessed soon, are prefetched, and knock out more useful ones from the cache. The other trend is that the miss rate decreases with increasing w because a larger lookahead window captures dependencies between accesses that are not necessarily successive.

With this model, the behaviors of the miss rate and the average access time metrics would be identical because both are linear functions of the amount of data corresponding to misses in the

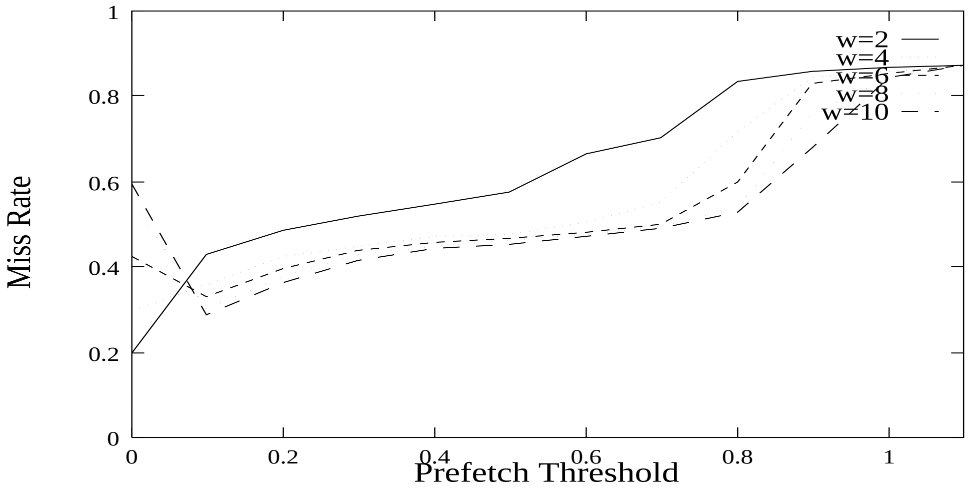

Figure 10: *Aggregate cache miss rate for all clients (each with a 4 MB cache) versus the prefetch threshold. There is no limit on the amount of prefetching information the server can pass on to a client (i.e., $i = \infty$). w is the lookahead window size.*

clients' caches.

Figure 11 shows the fractional increase in network traffic due to prefetching. We see that for smaller values of p and larger values of w (which corresponds to more aggressive prefetching), network traffic increases rapidly. Note that in the figure we have only plotted values of p larger than 0.3 because the curves shoot up for smaller values of p.

Our goal is to determine whether there are values of p and w for which there is a substantial reduction in miss rate over the case of no prefetching ($p > 1$), while the increase in network traffic is tolerable. From figures 10 and 11, we see that $w = 2$ results in a miss rate significantly higher than for other values of w, but also the lowest increase in network traffic. Among the other values of w, $w = 4$ looks most promising with a miss rate almost as low as for higher values of w, and the lowest increase in network traffic. So in the rest of this discussion we shall concentrate on $w = 2$ and $w = 4$.

The miss rate when there is no prefetching is about 0.88. For $w = 2$, the miss rate curve does *not* show a marked upswing at any particular value of p, rather it increases quite steadily. So choosing a value of p near the middle, say $p = 0.5$, the miss rate decreases to about 0.58 (an improvement of about 34% over no prefetching) at the cost of a 38% increase in network traffic. For $w = 4$, there a significant point of inflection in the miss rate curve at $p = 0.7$. At this point, the miss rate is about 0.55 (an improvement of 38%) , while the increase in network traffic is about 43%. Thus we see that, if we are ready to pay for about a 40% increase in network traffic, the miss rate can be reduced by about the same fraction, thereby resulting in a significant reduction in latency.

Thus far we assumed that the server, at any one time, could predict an arbitrary number of files as candidates for being prefetched, and convey this to the clients (i.e., $i = \infty$). It would be interesting to investigate whether a restriction on the amount of this "intelligence" can still result in significant gains over no prefetching. In particular, we consider the cases when the server, after receiving each fetch request, passes prefetch advice on at most one or two ($i = 1$ or $i = 2$) files. Figures 12 and 13 compare these cases with that when $i = \infty$, for $w = 2$. We note that the three curves merge for $p > 0.3$, which in any case is the portion that we are interested in. The reason is that, for larger

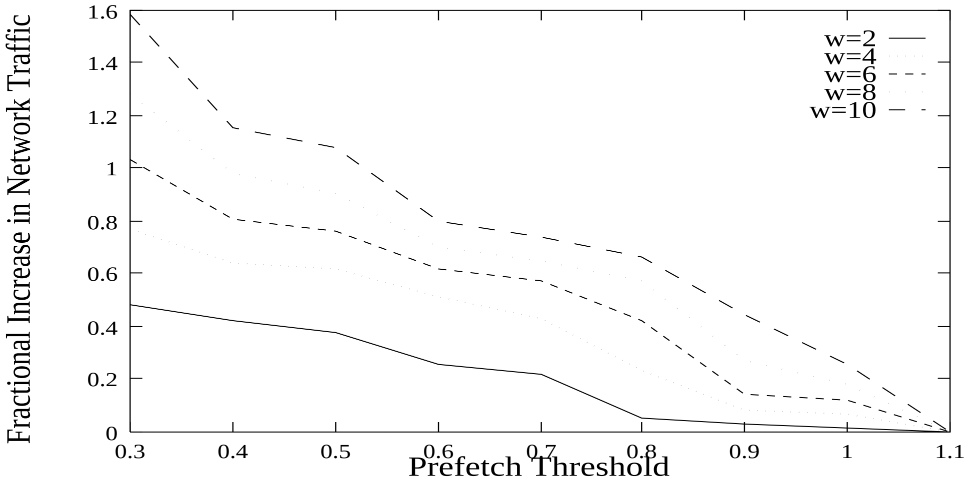

Figure 11: *The fractional increase in network traffic due to prefetching versus the prefetch threshold. Again, $i = \infty$.*

values of p, fewer files qualify for being prefetched. In fact, the figures indicate that the number is typically one or two, which is encouraging because it means that the server does not have to piggyback much prefetching information onto replies sent to the clients.

8.2 A more realistic model

One of the drawbacks of the model used in the previous subsection was that it did not take into consideration the interarrival times between requests. It is possible that requests for files that the prefetch daemon is very good at predicting arrive before the client is able to prefetch them. In this case, the previous model would incorrectly consider these as hits.

In this more realistic model, we use the interarrival times and an estimate of the time it will take to complete the prefetch, to decide whether there is enough time for the prefetch to be completed before the file is actually needed by the client. It is possible that, while there is not sufficient time, prefetching still reduces the latency to a certain extent because the prefetch request is sent by the client before the need to fetch that file actually arises. Since, in this case, the miss rate metric does not reflect such partial benefits, we report the average access time per file instead.

8.2.1 Estimating the time for prefetching

We needed to be able to estimate the time it takes for a client to prefetch a file, i.e., the duration from when it sends the prefetch request to the server till the time it receives all the data. In order to develop a model for this, we needed data points. Unfortunately, the server's log only contained local timing information, so we needed another way of getting data points. We instrumented the NCSA Mosaic browser to record the time taken to fetch each file and the size of the file. Running the browser on a host at Berkeley, we connected to the same Web server whose logs we used for all of our simulations. About 230 data points were obtained by randomly accessing Web pages on that server.

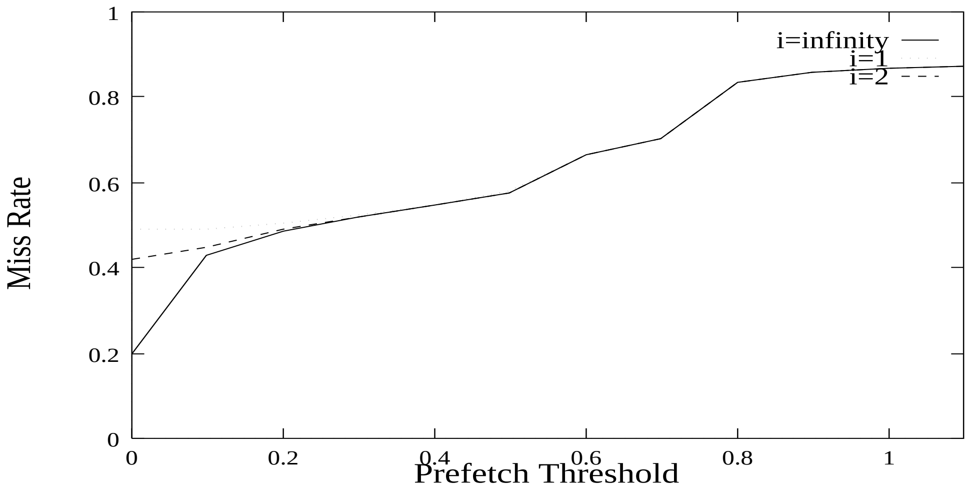

Figure 12: *Aggregate cache miss rate for all clients versus the prefetch threshold for $w = 2$ and various values of i.*

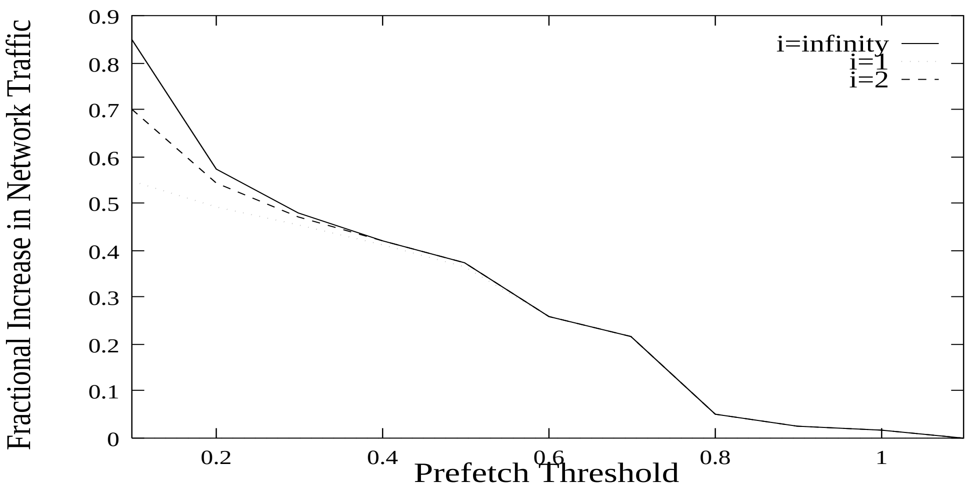

Figure 13: *The fractional increase in network traffic due to prefetching versus the prefetch threshold for $w = 2$ and various values of i.*

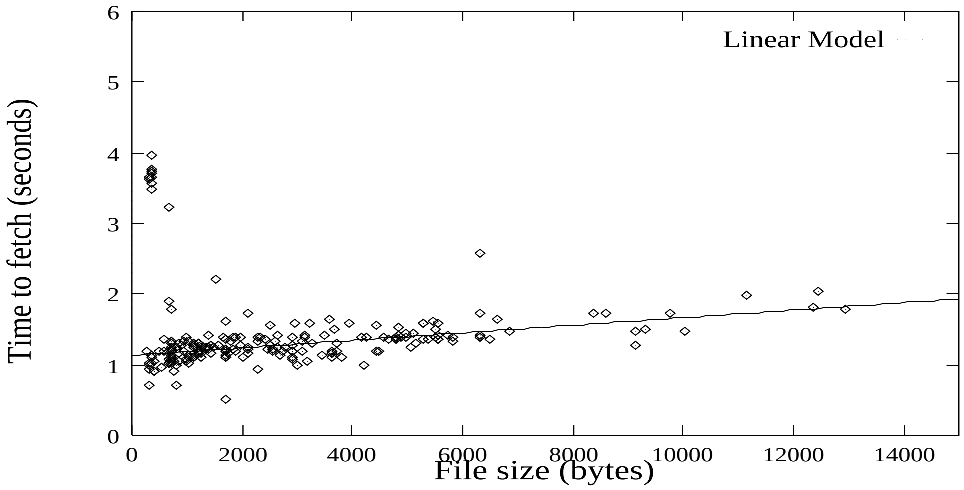

Figure 14: *A scatter plot of the times to fetch files (Web pages, inline images, etc.) of different sizes from DEC's Commercial Web Server, and the line corresponding to the linear regression model.*

Since the time to prefetch a file would be roughly the same as the time to just fetch the file on demand (i.e., when the user clicks on the corresponding URL), we used the data points obtained above to build a model for the time to prefetch a file. We hypothesized that this time depends mainly on the size of the file, and ignored other factors. One major concern is that our experiments were conducted from a client host at Berkeley, so our data points would not, in general, match those obtained from another site. We were, however, constrained by logistics to conduct experiments only at Berkeley. It is likely, though we are not sure, that most accesses to the Web server in question are made from hosts with similar network connectivity as Berkeley, so our model would still be a good approximation.

We use a linear regression technique [8] to model the time to prefetch a file. The basic idea is to come up with a linear model that minimizes the sum of squared errors, while ignoring outliers. A scatter plot of the data points and the line obtained from the regression model is shown in figure 14. The line corresponds to the equation $y = 1.13 + 5.36 * 10^{-5}x$.

8.2.2 Simulation Results

We now present the results of simulations that use the interarrival time information from the server log and estimations from the regression model to filter out prefetches that happen too late, and consequently do not help bring down the miss-rate. Note that there is no additional network traffic corresponding to these "wasteful" prefetches because a server would not try to fetch a file for which a prefetch is pending completion, and vice versa.

From figures 15 and 16, we see that, with our more realistic model for prefetching, the average access times are higher and the network traffic is lower for the same values of p and w. This is what we would expect given that some of the prefetches in the previous model will be filtered out here.

Since the price we pay for prefetching is essentially the increase in network traffic, it would be interesting to compare the average access times with the two models for the same increase in network traffic, say 40%. With the simple model, the average access time drops to about 0.78

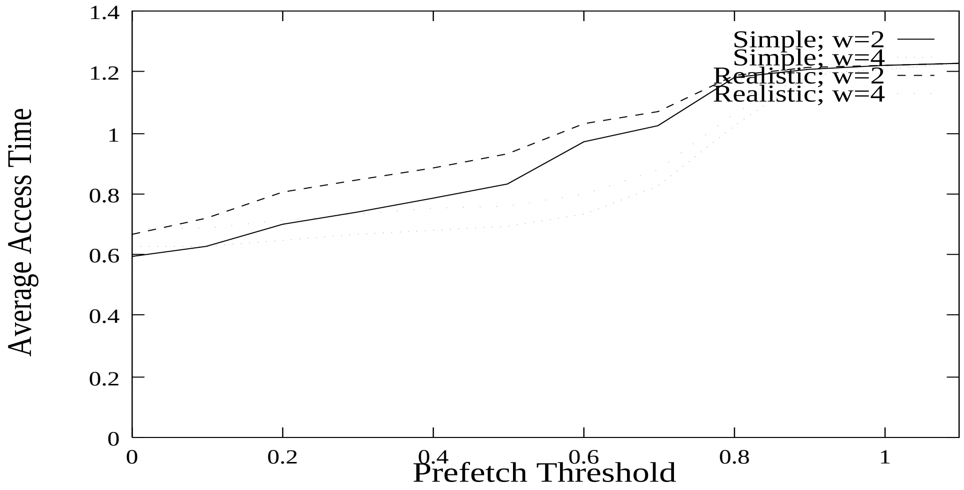

Figure 15: *Comparison of the average access time per file for this model (labeled "regression") with that for the simple model from section 8.1.*

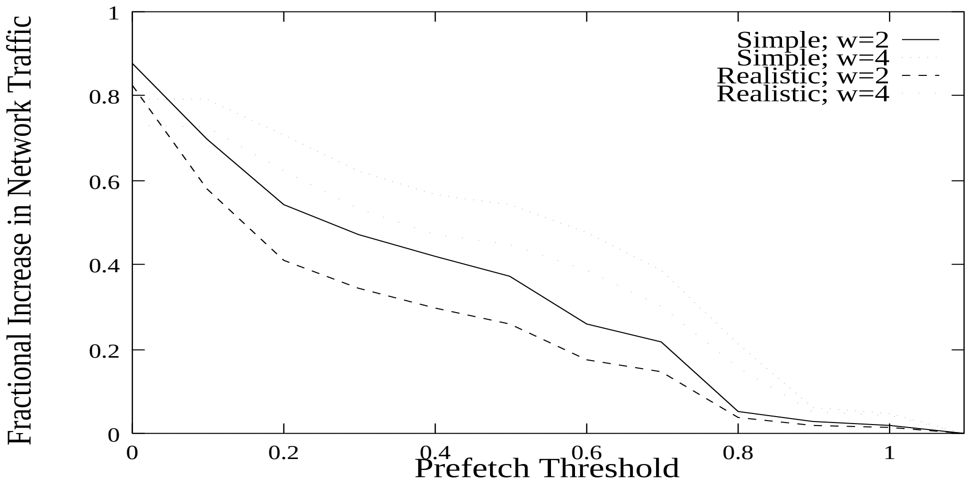

Figure 16: *Comparison of the cache miss rates for this model with that for the simple model.*

seconds (an improvement of 36% over no prefetching). With the more realistic model, the average access time is about 0.78-0.80 (an improvement of 34.5% to 36%) depending on w. Thus there is still a significant reduction in the miss rate as a result of prefetching.

9 Summary

We have analysed and quantified several sources of significant latency in the World Wide Web, problems that are inherent in the way HTTP is currently used. We have proposed several simple changes in HTTP that, individually or together, substantially reduce latency, while interoperating with unmodified servers and clients. These changes may also help reduce server loading.

We have also presented a prefetching scheme for the WWW aimed at masking latencies, in which the servers tell clients which files are likely to be requested next by the user, and each client decides whether or not to prefetch those files. We have analysed the benefits of prefetching based on two different models. The first is a simple one that assumes that prefetches happen infinitely fast, so the prefetched data is always available at the client host before the next access by that client. Then we refined this model by using the inter-arrival times between accesses and an estimate for how long a prefetch would take to complete, to filter out prefetches that happen with not enough time left till the next access.

The results from our simulations show that a substantial reduction in the latency perceived by a client (quantified in terms of the average time to access a file) can achieved at the cost of a significant increase in the network traffic. For instance, the reduction in latency is about 36% at the cost of a 40% increase in network traffic, with both the simple and the more refined models.

10 Conclusions

We have investigated several techniques for reducing latency in the World Wide Web. Based on this, we would suggest the following for future versions of the HTTP protocol:

1. There is a mismatch between the byte-oriented service of TCP and the message-based interface needed by HTTP. The ideal solution would be a session layer on top of TCP that would provide a message-based interface over a single TCP connection. However, in the absence of this, the HTTP protocol should have support for persistent connections built into it.
2. Primitives that are able to operate on a group of files/URLs at a time (such as GETLIST or GETALL) should be supported in order to reduce the number of network round trips.
3. Prefetching might be worthwhile, especially when increasing bandwidth demands does not significantly degrade service for other users nor increase the cost for service. An example would be a ground station downloading data over a direct broadcast satellite (DBS) downlink that provides a channel exclusively to that station. The HTTP protocol could have a facility that allows servers to piggyback prefetching hints on replies to clients. Also, it would help scheduling at a server if prefetches could be distinguished from regular fetches (for instance, to give them lower priority).

There have been other approaches suggested for reducing WWW latency, most notably that used by Netscape. From the networking standpoint, Netscape derives its speed from having several simultaneous TCP connections to the server, to retrieve data. The advantage of this approach over ours is that it does not require any modifications to existing servers. However, there are several drawbacks. Using multiple connections would be unfair to other protocols, such as the file transfer protocol (FTP) [14], that use a single connection to retrieve data (ignoring the control connection of FTP). Further, a bunch of TCP connections would be less regulated than a single connection. For instance, a packet loss on one connection would not cause the other connections to reduce their window sizes, though it is likely that the packet loss is an early symptom of congestion along the route which is, in fact, common to all the connections. Finally, Netscape does not avoid the paying the cost of slow start repeatedly.

There is at least one other work [10] that discusses these issues, and makes a strong case for persistent HTTP connections.

Acknowledgements

I would like to thank Dr. Jeffrey Mogul very much for supervising this project during the summer of 1994 while I was an intern at Digital Equipment Corporation's Western Research Lab, and for the useful discussions and suggestions even afterwards. I am grateful to my advisor, Professor Domenico Ferrari, for his constant support and encouragement, and for the opportunity to work as a part of the Tenet research group. I would also like to thank Digital's Western Research Lab, Network Systems Lab, and Cambridge Research Lab for allowing me to use their computing facilities for the purpose of this research.

References

[1] Tim Berners-Lee. "Hypertext Transfer Protocol (HTTP)", *Internet Draft draft-ietf-iiir-http-00.txt, IETF,* November, 1993. This is a working draft.

[2] N.Borenstein, N.Freed. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", *RFC 1521, Internet Engineering Task Force*, September, 1993.

[3] R.Braden. "Requirements for Internet Hosts – Communication Layers", *RFC 1122, Internet Enginnering Task Force*, October, 1989.

[4] CompuServ, Incorporated. *Graphics Interchange Format Standard*, 1987.

[5] Steven Glassman. "A Caching Relay for the World Wide Web", *Proceedings of the First International World Wide Web Conference, Geneva*, pages 314-329, May, 1994.

[6] James Griffioen and Randy Appleton. "Reducing File System Latency using a Predictive Approach", *Proceedings of the 1994 Summer USENIX Technical Conference*, Boston MA, June, 1994.

[7] Van Jacobson. "Congestion Avoidance and Control", *Proceedings of the ACM SIGCOMM Conference*, Stanford, CA, August, 1988.

[8] Raj Jain "The Art of Computer Systems Performance Analysis", *John Wiley & Sons, Inc.*, 1991.

[9] Jeffrey C. Mogul and Stephen Deering. "Path MTU Discovery", *RFC 1191, Network Information Center, SRI International*, November, 1990.

[10] Jeffrey C. Mogul. "The Case for Persistent-Connection HTTP", *Proceedings of the ACM SIGCOMM Conference* (to appear), Boston, MA, August, 1995.

[11] Venkata N. Padmanabhan and Jeffrey C. Mogul. "Improving HTTP Latency", *Proceedings of the Second International World Wide Web Conference, Chicago, IL*, pages 995-1005, October, 1994.

[12] J.Postel. "Transmission Control Protocol", *RFC 793, Network Information Center, SRI International*, September, 1981.

[13] J.Postel, J.Reynolds. "Telnet Protocol Specification", *RFC 854, Network Information Center, SRI International*, May, 1983.

[14] J.Postel, J.Reynolds. "File Transfer Protocol (FTP)", *RFC 959, Network Information Center, SRI International*, October, 1985.

[15] Simon E. Spero. "Analysis of HTTP Performance Problems", *URL http://elanor.oit.unc.edu/http-prob.html*, July, 1994.