The following paper was originally published in the
Proceedings of the USENIX 1997 Annual Technical Conference
Anaheim, California, January 6-10, 1997

Optimistic Deltas for WWW Latency Reduction

Gaurav Banga; Rice University
Fred Douglis, Michael Rabinovich; AT&T Labs - Research

Optimistic Deltas for WWW Latency Reduction

Gaurav Banga
Rice University
gaurav@cs.rice.edu, http://www.cs.rice.edu/˜gaurav/

Fred Douglis
AT&T Labs – Research
douglis@research.att.com, http://www.research.att.com/˜douglis/

Michael Rabinovich
AT&T Labs – Research
misha@research.att.com, http://www.research.att.com/˜misha/

Abstract

When a machine is connected to the Internet via a slow network, such as a 28.8 Kbps modem, the cumulative latency to communicate over the Internet to World Wide Web servers and then transfer documents over the slow network can be significant. We have built a system that optimistically transfers data that may be out of date, then sends either a subsequent confirmation that the data is current or a delta to change the older version to the current one. In addition, if both sides of the slow link already store the same older version, just the delta need be transferred to update it.

Our mechanism is optimistic because it assumes that much of the time there will be sufficient idle time to transfer most or all of the older version before the newer version is available, and because it assumes that the changes between the two versions will be small relative to the actual document. Timings of retrievals of random URLs in the Internet support the former assumption, while experiments using a version repository of Web documents bear out the latter one. Performance measurements of the optimistic delta system demonstrate that deltas significantly reduce latency when both sides cache the old version, and optimistic deltas can reduce latency, to a lesser degree, when content-provider service times are in the range of seconds or longer.

1 Introduction

The Internet, and particularly the World Wide Web (W^3), consists of an ever-increasing number of servers, networks, and personal machines with dramatically varying qualities of service. Individuals often access the W^3 via modems with bandwidth of 14.4–28.8 Kbps, while their provider might have a T1 link or better to the rest of the Internet. But then the actual access may be to another site with low bandwidth, high server load, or both. Thus the latency to respond to the user's request for a page is unpredictable: often fairly low, but sometimes extremely high. The unpredictability and generally slow response may be exacerbated in environments with even poorer quality of service, such as cellular telephony or wide-area wireless networks.

A number of techniques have been implemented or proposed to deal with HTTP latency. A browser can direct its request to a proxy-caching server (henceforth referred to as a *server proxy*) on the other end of the low-speed connection, and then the latency for retrieving pages elsewhere in the Internet can be eliminated when someone else has retrieved those pages in the recent past. (Recency is a function of the size of the cache, any expiration dates in the pages, and any constraints passed from the browser to the cache [5, 12]. Also, some pages are flagged as uncacheable, and the proxy-caching server is obliged to pass those requests through to the content provider. Caching is discussed further in Section 2.) America Online (AOL) uses a proprietary protocol between the browser on a user's machine and an enormous cache of W^3 pages within the AOL server cluster. Prefetching pages during periods when the modem would otherwise be idle can reduce or eliminate the latency of following a link, if the prefetch is accurate and the user thinks between clicks long enough for prefetching to complete [18, 22]. Padmanabhan and Mogul's study of persistent HTTP connections [17] indicates that a persistent connection between a client and proxy, or between one of them and a content provider, will eliminate TCP connection setup and slow-start overhead.

We look at the problem of latency from another perspective: using computation to improve end-to-end network latency. The idea of trading off computation for I/O bandwidth has appeared numerous times in past systems. Examples include application-specific deltas and compression, such as Low-bandwidth X[15]; compressed network or disk I/O [3, 6, 7]; replicated file systems [2]; shared memory [16]; and checkpointing [9, 19]. It seems that the same tradeoffs apply in the domain of the W^3.

We address the issue of latency from the perspective of sending the differences between versions of a page, or *deltas*, in order to avoid sending entire pages. Briefly, deltas are used in two ways. First, if both ends of the slow link store the same version of a page, and the server proxy obtains a new copy of the page, it can send a delta to the user's machine. This hopefully will reduce the transfer time on the slow link. Second, if the user's machine does not store the older version, the server proxy can send a potentially obsolete version of the page immediately and request the new version in parallel. It follows this with a delta against the obsolete version if necessary. Thus, the idle time on the slow link when the server proxy is waiting for a response from the end server is not wasted.

The size of the delta may turn out to be large, in which case the server proxy may have to abort sending the stale version (if one is being sent) and just send the current page received. In this case, work for sending the stale data and calculating the delta is wasted. Worse, as the server proxy does not pipeline the data from the content provider to the client but instead waits till the whole page has been received (since it needs to compute a delta), the end latency as perceived by the client may be somewhat larger. Our approach optimistically assumes that this case is uncommon; hence we refer to the case where stale data is transferred as an *optimistic delta*. The experiments described in this paper support this assumption. In contrast, we refer to the case where both sides share a cached version as a *simple delta*.

As we were going to press, we found that the "simple deltas" case is similar to a system from IBM called WebExpress [13]. WebExpress is geared toward a low-bandwidth wireless environment, where bandwidth is precious, so it has similar goals but makes different trade-offs. It focusses on small changes to dynamic data (CGI output), sending deltas between a base version that is shared by the client- and server-side "intercepts" (similar to the client and server proxies described here). However, WebExpress has apprently not been used so far for arbitrary W^3 pages, nor does it send stale data optimistically. The limited bandwidth, high error rate, and contention in a wireless environment suggest that transferring data that may not be used could have more negative consequences than over a single-user modem.

Our work also has some similarity to Dingle and Partl [5], who proposed that a hierarchy of proxy-caching servers could be used to send stale data as a MIME multipart document, causing the browser to display the stale data immediately and to replace it with more recent versions as they become available. The main difference with our work is that Dingle and Partl do not address the latency of obtaining the final version of the data. Our approach attempts to reduce this latency by sending the the current version as a delta. In addition, we do not display the stale page: it is intercepted by a *client proxy* [20], typically co-located with the browser on the same machine, and passed to the browser once it is updated or known to be current. In the case where the "stale" page is actually current, our system behaves like the proposal in [5]. Finally, intercepting the stale data by a client-side proxy permits the transmission of the stale data to be aborted transparently once it becomes clear that simply sending the current version is more efficient (e.g., if the delta turns out to be too large relative to the page, or if the current page became available very quickly).

The rest of this paper is organized as follows. Section 2 provides some background into caching in the HTTP protocol and an analysis of HTTP latency. Section 3 discusses some data analysis to support our hypotheses that delta sizes will be small and that there will be sufficient idle time to transfer stale copies. Next, Section 4 describes the design of our system, and Section 5 covers experimental results. Finally, Section 6 discusses the status of the system and future work, and Section 7 offers some conclusions.

2 Background

In this section we briefly describe caching in HTTP and analyze the dynamics of a typical HTTP transfer. This description provides background for the discussion in the following sections.

2.1 Caching in the HTTP Protocol

The HyperText Transfer Protocol (HTTP) [4] supports caching in clients (i.e. browsers) and intermediate servers known as proxy-caching servers. To display a page, a client without a cached copy will unconditionally send a page request to the content-provider or a proxy-caching server. A client with the page already cached may return the cached copy or contact another host to determine whether the page has changed. This check for page currency is done by sending an HTTP GET request with a header specifying If-Modified-Since followed by the timestamp of the cached copy. If a proxy-caching server is used, it can respond to a page request or currency check request if it has a cached copy that is deemed usable and if the client has not specified that the cache must not be used (with a Pragma: no-cache[1] directive, most commonly sent when a user tells a browser to reload a fresh copy of a page).

The proxy-caching server decides whether or not the cached page is usable and, if so, whether or not the client's cached copy is current, based on the optional extra information that the clients can send with their requests and which HTTP servers can send back with a page. Beyond the Pragma: no-cache directive mentioned above, the client may specify a bound on the age of the cached copy it is willing to accept (the Cache-control: max-age directive). Content-providers can specify when the page was last modified, whether or not the page can be cached (the Cache-control: no-cache field), and how long a client or proxy-caching server should cache the page (the Expires field). Dynamic data, often the output of a CGI script, is typically sent with no Last-Modified timestamp and set up to expire from the cache immediately (equivalent to disabling caching).

Currently, if a page has no Last-Modified timestamp, checking for the freshness of a cached copy requires retrieving the fresh copy from the content provider and shipping it all the way to the client browser. Similarly, changes to a page with the timestamp will require the proxy to obtain the file from the content provider and transmitting the entire file to the client; the transmission is elided only if the page has not been modified at all.

2.2 Analysis of HTTP Latency

We now present an analysis of the timing dynamics of a typical HTTP transaction. We assume a setting where the Web browser (client) talks to a server proxy on the other side of a low bandwidth and high latency link, which in turn talks to HTTP servers (content-providers) on the Internet. We use this analysis to make a case for the usefulness of optimistic deltas.

Consider Figure 1, which depicts the timeline corresponding to an HTTP transaction between a client, a server proxy, and a content provider. Packets 1, 2 and 3 correspond to the SYN, SYN/ACK and ACK packets that are exchanged as part of the TCP 3-way handshake in the connection establishment phase between the client and the server proxy. The client sees this connection as established after a delay of approximately $2 * L_{slow}$ where L_{slow} is one half of the round-trip latency of the link between the client and the server proxy. At this point the client sends packet 4 which contains the HTTP request. After the server proxy gets this packet (at approximately time $3 * L_{slow}$), if it needs to go to the end server to satisfy this request or to validate its cached copy, it initiates a connection with the content provider (packets 5, 6 and 7). The latter connection is established after a further delay of $2 * L_{fast}$, where L_{fast} is one half of the round-trip latency of the link between the server proxy and the content provider.

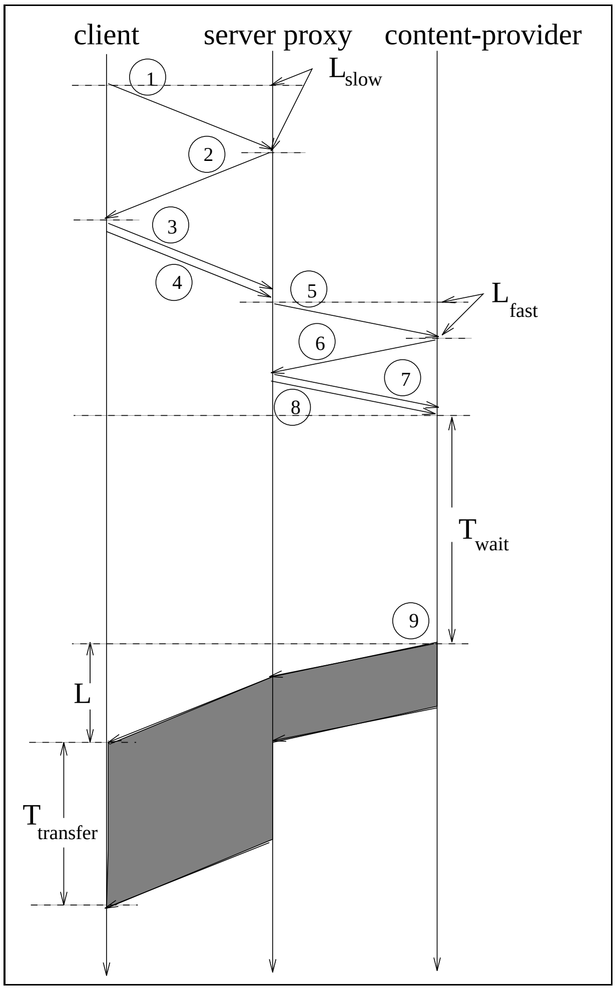


Figure 1: Timeline for typical HTTP request.

[1] HTTP 1.1 alternatively supports a Cache-control: no-cache directive, which when sent by the client is equivalent to Pragma: no-cache. In this paper we refer to the pragma to mean either header.

The server proxy then forwards the client's HTTP request to the content provider (packet 8).

On receipt of this HTTP request the content provider does whatever is necessary to produce the response. This may include several relatively slow activities, such as forking off a child server and/or passing descriptors to it, forking off a script to produce the response document, invoking and waiting for the response from a search engine, etc. For our purposes we will view these activities as a certain latency shown as time T_{wait} in the figure. The magnitude of T_{wait} is thus a function of the server in question, the particular URL being served and the load on the server. This time can be highly variable. Some simple experiments on the Internet indicate that it varies from a couple of hundred milliseconds to several seconds. After the response (packet 9) is available and sent off it takes another $L = L_{slow} + L_{fast}$ time before the client starts seeing it. The amount of time $T_{transfer}$ that it takes to transfer the response is dependent on the size of the response and speed of the link.

If we put in typical values for the various time parameters in the figure (L_{slow} = 160ms, L_{fast} = 60ms, T_{wait} varying from 200ms to several seconds, $T_{transfer}$ for a 2-KByte response at 20 Kbps is 800ms[2]), we notice that if T_{wait} is large, the low bandwidth link is idle for a substantial portion of the time. Also, regardless of the magnitude of T_{wait} we want to minimize the amount of data we transfer over the slow link. These two observations lead to the idea of optimistic deltas. We want to effectively utilize the slow link in its idle time, and if possible, reduce the amount of data transferred, in order to reduce the total latency perceived by the client. If the client and server proxy both cache the same older version, only a delta needs to be sent over the slow link decreasing $T_{transfer}$. If the client and server proxy do not cache the same version, the server proxy can transfer its cached version while waiting for data from the content-provider and subsequently send a delta. Here again, $T_{transfer}$ is shorter if T_{wait} is long enough.

3 Data Analysis

Simple deltas benefit by trading off computation of the deltas for a reduction in bandwidth and latency over the slow link when both sides store the same old version of a page. Optimistic deltas trade off an *increase* in the amount of data transferred, by sending an older version during an idle time of the slow link followed by a delta, for a reduction in end-to-end latency. The viability of either form of deltas is thus dependent on "smallness" of deltas, which we evaluate in Section 3.1. Optimistic deltas depend additionally upon long idle times on the slow link, which we consider in Section 3.2.

3.1 Delta Sizes

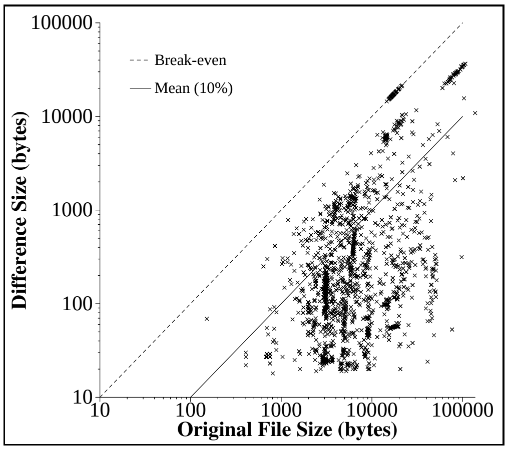
To test the hypothesis that deltas would be sufficiently small, we wanted to take a sample of W^3 pages and see how large the differences were between two versions of the same page, relative to the page itself. We considered two sources of sample pages: the version archive of the AT&T Internet Difference Engine (AIDE) [8], which stores versions of pages for future visual comparison of their changes, and a set of random URLs obtained from AltaVista [1]. The random pages were tracked by AIDE as well, but they were considered separately from those pages that were actually registered explicitly, either by individuals or by inclusion in a list of popular URLs collected from a set of bookmark files within AT&T. The random URLs and popular URLs were archived daily if changes were detected, while the pages tracked for individual users were typically archived upon explicit request.

Throughout our experiments, we computed deltas using *vdelta*, a program that generates compact deltas by essentially compressing the deltas in the process of computing them, and which can be used as a stand-alone compression program as well [10].[3] We must consider the possibility that W^3 pages that are compressed in a stand-alone fashion will compress so well that the deltas between two versions of a page are not much smaller than the compressed page. In this case the client and server proxies could merely compress every page (or rely on compression in the modems) without using deltas and have the same benefit. We will see that in practice, however, deltas are substantially smaller than simple compression.

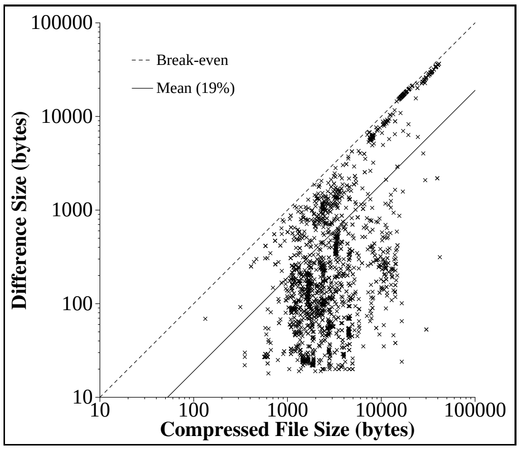
Considering first the non-random pages, of a total of 380 pages in the archive, 181 had more than one version, with a mean of 4.9 versions/page ($\sigma = 10.3$). Figure 2(a) shows a plot of delta size against original file size. The delta size is usually a small fraction of the original file size. By comparison, Figure 2(b) plots delta size against the size of the newer file once compressed. Figure 2(b) indicates that the delta is consistently much smaller than the compressed file, though in some cases it is approximately the same; this usually happens when a file has changed completely from one version to the next. Even if the file does not compress

[2] This number is very approximate and will in practice be larger because of TCP's congestion control algorithms.

[3] In fact, one can consider a delta of B compared to A as compressing B with the strings of A already in the compressor's table of prefices: if B is similar to A then it will compress well, and if not, it will still compress well if it has internal similarity (as most ASCII text does).



(a) Deltas compared to original file sizes.



(b) Deltas compared to compressed output sizes.

Figure 2: Comparison of sizes of deltas and original or compressed pages, using *vdelta*. While deltas have a greater benefit when simple compression is not taken into account, they help above and beyond the benefits of compression. In each graph, the dashed line indicates the break-even point and the solid line depicts the mean across all files.

well (for instance, it is a GIF file), the worst that *vdelta* will do is to reproduce the original file with a few bytes of overhead. The mean across over 2200 comparisons of delta/compressed-file ratios was 19% ($\sigma = 27.6\%$).

The outlying points in Figure 2, which are due to one GIF file that has been archived automatically each day and a compressed postscript file with two versions in the archive, might be a concern in practice if the system were to send stale copies and compute deltas regardless of file type. Fortunately, file types are identifiable, both from the $\mathrm{Content\text{-}type}$ HTTP header and data within the files, so it is possible to treat images and other non-textual data specially. One might instead use a distillation technique to send a version of an image that is more appropriate for a low-bandwidth link [11].

Our study of 1000 random URLs from AltaVista [1] found that 861 URLs were actually accessible at the time we started tracking them, and the vast majority (79%) of those URLs were not modified in the next two months of daily checks. Figure 3 graphs the distribution of the number of versions detected for the remaining 21% that were modified. Just 43 of the 861 URLS (5%) had 40 or more versions over the two months of the study; the minor variations in the number of versions of the frequently-changed pages may be due to transient errors while contacting those hosts. We also performed the above analysis of delta sizes for these pages, and found that the mean delta size was just 3.7% of the original page size ($\sigma = 6.9\%$), and 10.4% of

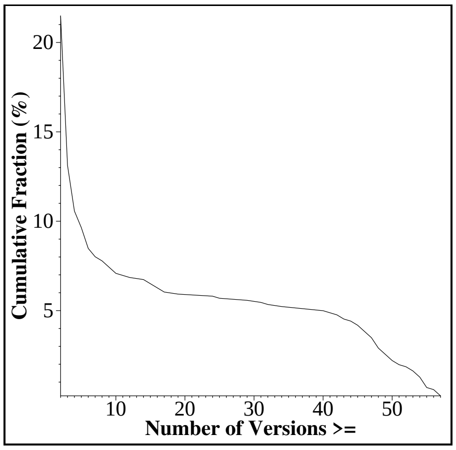


Figure 3: Distribution of the number of versions detected by daily checks of 861 randomly selected URLs over a two-month period, for the 21% of pages that were modified.

the compressed page size ($\sigma = 14.0\%$). The pages themselves compressed to an average of 12.1% ($\sigma = 18.0\%$) of their original size. A possible explanation for the small deltas is that the sample was dominated by pages that changed daily, and those changes may often have been the inclusion of timestamps or other small, simple modifications.

Another consideration is what sorts of data can be compared. Even dynamic pages, which aren't cacheable, might have a lot of overlap between versions of the same page, or pages with the same base URL but different parameters to a CGI script. (Determining when to compare one URL against a slightly different URL for differencing is an open question, but as long as both the client and server proxies agree on the versions being compared the system will act correctly.)

For example, a query to the AltaVista search engine [1] might result in a page containing several links to content and several more links to other URLs within AltaVista. The "boilerplate" can dominate the content that changes from page to page, because each page contains the same form at the top and, at the bottom, a set of links to each other page generated by the query. Figure 4 graphs the sizes of deltas from two queries, compared to the size of the page if it were just compressed. The first, a name lookup, returned 9 pages; the second, a query with many terms ("storage management mobile computing flash memory nvram") that generated thousands of matches, returned 20 pages, of which 10 were compared. In each case the deltas from one page to the next, within a given search result, were much smaller even than the compressed pages.

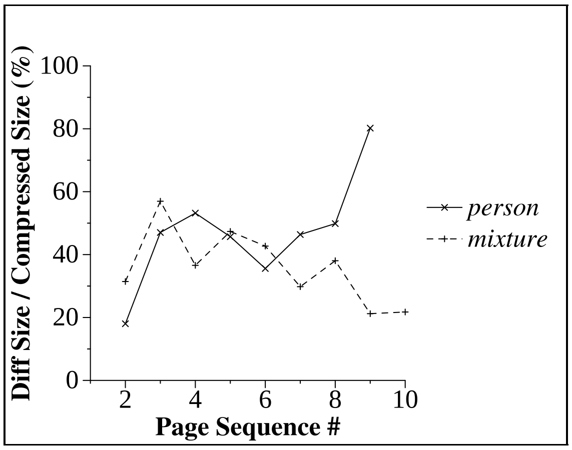


Figure 4: Example of deltas for two AltaVista queries, one for a person and one a mixture of computer science terms. All comparisons were pairwise in sequential order, starting with a delta between the first two pages of a query result. The URL of each page varied slightly because it specified the range of responses to return (1-10, 11-20, etc.).

3.2 HTTP Latency

To get a sense for the likelihood that a request would take a long time to start receiving data, we collected 1000 random URLs from AltaVista [1] and timed their responses. This study differs somewhat from Viles and French [21], who studied the availability of random HTTP servers and the time to *connect* to them; here we are seeing how long it takes to collect the first data from a W^3 page. Figure 5 shows the results of this experiment, based on the 722 URLs that returned data within the first minute. We found that about a third of pages responded within a second, assuming they responded at all, and half responded within about 1.6s. However, it takes 5s to cover $\frac{3}{4}$ of the pages and 10% took 10–30s or more for the first data to arrive. As more pages on the W^3 are dynamically generated, we expect the fraction of pages with sluggish response to increase.

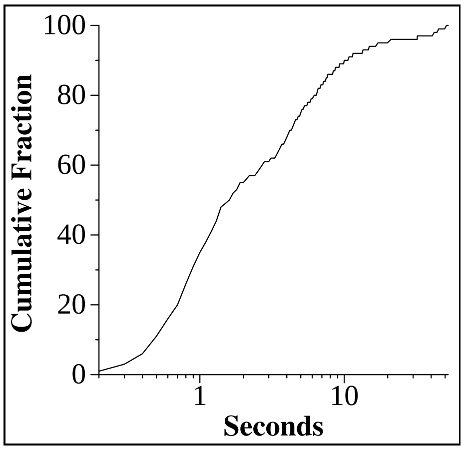


Figure 5: Distribution of response times to receive first data, based on a sampling of 722 URLs. Roughly $\frac{1}{3}$ of the sample received data within a second, and $\frac{3}{4}$ received data within 5 seconds.

4 System Design

4.1 Design Considerations

Once we were confident that sending deltas could often reduce bandwidth requirements and/or client-observed latency of Web access, we had to consider two issues. One was what the architecture of a delta-based system would be. We rejected any scheme that would require changes to HTTP or to existing content providers, though we later learned that HTTP/1.1 will support a PATCH directive to allow efficient uploading of changes to shared documents. Instead, we settled on a proxy-based architecture in which the browser connects to a proxy on the same machine (the client proxy), and that proxy in turn connects to a server proxy on a well-connected host. We have control over both of these proxies; in fact we use a common source code base for them, though that is not necessary. The design of the proxies is covered in greater detail in Section 4.2.

The second open issue was how the deltas would help when the user's machine does not store an old version of the page. If the well-connected proxy has many clients, or can talk to a nearby proxy that does, it may have fast access to a cached copy of the requested page. When that page is *current*, nothing need be done other than sending the cached copy over the phone line. If it is stale, however, there might be an opportunity to use deltas after sending the stale data.

In fact, one might expect the server proxy to handle a number of clients and to keep multiple versions of each document in its cache. Sometimes a client's request will specify a version which the caching proxy has, and then only a delta needs to go back over the slow link. Other times, the server proxy will not have the same version cached as advertised by the client in which case it will try to utilize the T_{wait} time by sending in the stale copy it has and subsequently sending the delta. Since in the latter scenario the benefit is potentially lower, especially when T_{wait} is not very large, we may bias the system so that it hits the first scenario more often than the second. This may be brought about by auxiliary mechanisms like prefetching during idle times to keep the client and server proxies' caches in sync.

In total, there are numerous ways in which the optimistic delta mechanism can improve the efficiency of W^3 access:

- The client and server proxies may share an out-of-date version. Consider the case when the client proxy sends an If-modified-since request with a No-cache directive to the proxy, which caches the same version, and assume that the page has been modified on the content provider site. In this case, the proxy obtains the page, computes the delta and, if it is smaller than the whole page, sends the delta instead of the whole page to the client. Thus, the demand for bandwidth of the (slow) link to the client is decreased. Again, we refer to this case as a "simple delta," which is less "optimistic" than others: it only relies on the delta being small enough to be beneficial but does not risk transferring useless data.

- The server proxy may have the current version, but the client proxy wants to check the validity of its own cached copy with the content provider. Assume the client sends an If-modified-since request with No-cache to the proxy server, which caches a newer version that is the same as the version on the content provider site. In this case, the proxy immediately sends its copy to the client (marked as Stale); in parallel, it sends an If-modified-since request to the content provider, verifies that its copy is actually current and sends a null delta to the client. The browser can display the page as soon as the conditional GET returns via the server proxy, rather than having the newer contents of the page transferred starting then.

  The same latency reduction applies if the client has no cached copy but requests the most current version of a page, since the server proxy can send a Stale copy and then confirm that the copy is current after its own If-modified-since request.

- The server proxy may have a newer version than the client, as well as the client's cached version. Assume the client proxy asks the server proxy for a page that the client has cached, and the server's copy is more recent but is not necessarily the most current version. The server can respond with a delta against the client's version. If the page is out of date or the client specifies that the cache be bypassed, the content provider is consulted and a second delta can be sent if needed.

- The client and server proxies may share a current version of an "uncacheable" page, one that must be retrieved directly from the content provider on each access. Our system permits the client and server proxies to cache such pages as a basis for deltas between versions of a page, while ensuring correctness by providing the browser with the most current version every time. The server proxy can determine that the page is unchanged (using a regular GET over the high-speed network and comparing the contents with the cached version) and notify the client proxy to use the cached version rather than transferring the page over the low-speed network. Note that while other systems do this for cacheable pages, ours does this even for uncacheable ones as far as the slow link is concerned, while taking advantage of deltas when the differences are small.

4.2 Architecture

Figure 6 shows the architecture of our system. The browser connects to a local (client) proxy using the usual HTTP proxy-caching mechanism, by sending it requests containing the full URL of a desired page. We assume that the browser does not cache pages, but instead relies on the client proxy; however, this does not affect correctness, only storage utilization and caching effectiveness. The client proxy serves the request out of its cache if possible, or forwards it to the server proxy on the other side of the slow connection. The added overhead of a client proxy on the same machine is minimal by comparison to network delays and so the analysis in Section 2.2 still holds.

The server proxy can respond using its cache if the request is for a cached page, the page is not out of date with respect to its expiration date (if any), and the client has not issued the nocache pragma. Otherwise, it forwards the request upstream, either directly to the content provider or to yet another proxy-caching server. In either case, if the client and server proxies share a cached version, a delta from that version to the current version can be sent once the current version is available. If they do not share a cached version but the server proxy has some version cached, the server can send the possibly stale version followed by a delta from that version to the current version.

The server proxy determines what the client proxy has cached via some extra headers in the HTTP request. First, an Accepts: multipart/attdelta field indicates that the client understands the delta format. This way, a browser or other unmodified client can talk to our proxy without getting back something it cannot interpret. Second, a Current-version: [signature] field informs the server which version the client has cached, if any. The signature can in principle be anything that can distinguish different versions of a document, such as an MD5 checksum. We make the simplifying assumption that any client proxy that requests a new version from a server proxy will have received the previous version from the same proxy, and we use a monotonically increasing version number (instead of a checksum) that the server generates and passes to its clients.

Table 1 summarizes the possible combinations of client/server proxy states and the procedures that are followed.

4.3 Detecting Non-cost-effective Transfers

As was mentioned in the introduction, one difference between our system and the proposal of Dingle and Partl [5] is the ability to abort the transfer of stale data. In fact, there are two cases when it is more appropriate to behave like standard proxies and just send the current version of a page to the client without delta processing.

In the "simple delta" case, the client and server proxies cache the same stale version of the page, and only the delta need be sent. If the delta is as large as the current page, the current page rather than the delta must be sent. If the delta is somewhat smaller, one could use heuristics to decide whether the cost of recreating the current version from the delta exceeds the benefits due to bandwidth reduction. We currently transfer the delta any time it is smaller than the original.

The other case occurs when an optimistic transfer is in progress and the new version of the page starts to arrive at the server proxy. If most of the stale version has been sent and the delta is small, it pays to finish sending the stale version; if there is a lot of data yet to be transferred and/or the delta is large, the transfer should be aborted and the current version should be sent as it becomes available.

If the cost of recreating the page from the stale version and the delta is negligible, and we assume that the two versions are the same size, then we should continue to send the stale version any time the remaining stale data plus the size of the delta is less than the size of the current version. (In practice, we will know the size of the current version if the Content-length header field is present.) Assume the length is L and the size of the delta is δL. If we have already transferred δL bytes of the stale version, then transferring the remainder plus δL will require no more bytes than sending all L bytes of the current version.

Since we have no way of knowing how large a particular delta will be, any scheme that depends on computing the delta only after the whole response has been received by the server proxy can sometimes perform badly. However, there is an entire family of increasingly sophisticated abort schemes that one can think of, which can be integrated into the process of receiving the current version, producing the delta on the fly, and aborting if the delta appears large.

5 Experiments

Ideally, we would like to perform a long-running experiment that would compare end-to-end performance of our optimistic delta mechanism with existing proxy

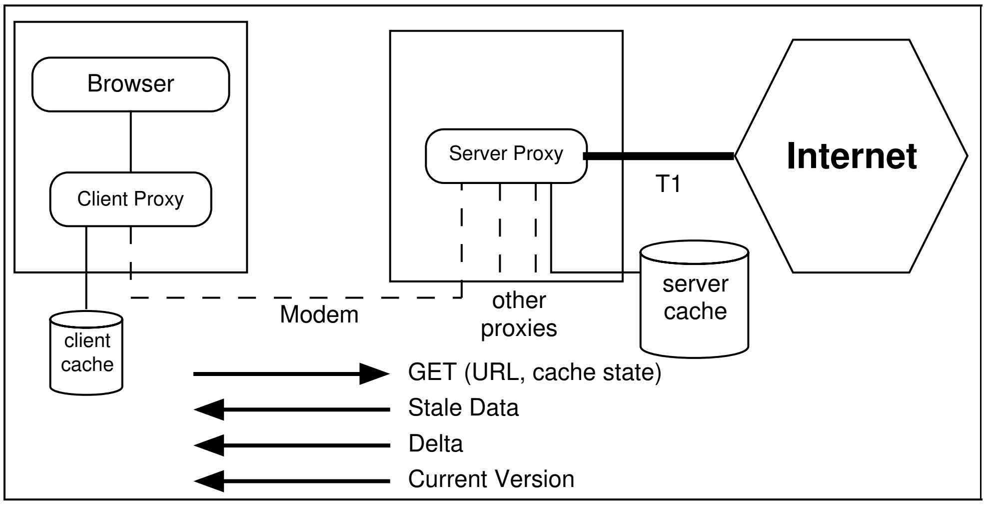


Figure 6: System architecture.

<table>
<tr><th>Client proxy has cached copy?</th><th>Server proxy has cached copy?</th><th>Cached copy current?</th><th>Content provider's copy Modified?</th><th>Server proxy's action</th></tr>
<tr><td>irrelevant</td><td>no</td><td>irrelevant</td><td>irrelevant</td><td>retrieve current version and send to client proxy</td></tr>
<tr><td rowspan="3">no</td><td rowspan="6">yes</td><td>yes</td><td>irrelevant</td><td>send cached copy</td></tr>
<tr><td rowspan="2">no</td><td>no</td><td>send cached copy upon request, confirm current after GET If-Modified-Since to content provider</td></tr>
<tr><td>yes</td><td>send cached copy upon request, send delta or new copy after GET If-Modified-Since to content provider</td></tr>
<tr><td rowspan="3">yes</td><td>yes</td><td>irrelevant</td><td>confirm current</td></tr>
<tr><td rowspan="2">no</td><td>no</td><td>confirm current after GET If-Modified-Since to content provider</td></tr>
<tr><td>yes</td><td>send delta or new copy after GET If-Modified-Since to content provider</td></tr>
</table>

Table 1: Possible states when requesting a URL. “Cached copy current?” refers to whether the server proxy can respond using its cached copy without consulting the content provider.

caching. One way to perform such measurements would be to get a set of random URLs, and then request each URL periodically for an extended time using the optimistic delta approach and the existing approach, and compare the average response time. We plan to perform such experiments in the future.

To date, we have focussed on "microbenchmarks" to study the two extremes of interest: the case when the client does not have a page cached, and must obtain a full copy of the page (possibly a stale copy followed by a delta or confirmation that it is current); and the case when the client and server have the same copy cached and the server can send a delta. We compare each of these against the response time of an unmodified system that connects directly to the server proxy and does not use deltas.

5.1 Experimental Setup

We performed our tests using a pair of Intel-based systems running the proxy code and a Sun SparcStation providing content. Specifically, the client system was a Pentium 133Mhz based machine with 32MB of RAM, running BSDI's BSD/OS v2.1. (Our proxy code is very portable across Unix platforms and currently compiles without any change on SunOS, Solaris, Linux, FreeBSD and BSD/OS. Its main system dependencies are BSD sockets and *vdelta* so we expect it should be easily portable to any system that has BSD sockets and some kind of difference library that supports binary files.) It was connected using an AT&T Paradyne Comsphere 3820Plus modem at 28.8 Kbps to a dial-in server on the AT&T corporate network. The server proxy ran on an identical machine in the AT&T network one hop away from the dial-in server. The server connected to an HTTP daemon (*htd*, an internally developed server) on a uniprocessor SparcStation 20 on the same Ethernet segment as the server proxy. This provided relatively fine-grained control over the latency between the server proxy and the content provider.

The "browser" was a simple C program that fetched a series of URLs specified in a control file by communicating with the client proxy, which also ran on the client machine. The "browser" did not do any caching.

5.2 Test Data

Here we report a performance evaluation using a synthetic workload based on the multi-version archive of W^3 pages collected by the AT&T Internet Difference Engine (described above in Section 3.1). This archive reflected the actual evolution of the pages, although it did not contain copies of every version of every page: some pages were archived automatically once per day when changes were detected, while the majority were archived upon the explicit instruction of a user of the system.

Slightly over half of the pages had only one version archived; these reflected pages that were registered with the system but had either never changed or (more likely) were not archived automatically and had not been selected for subsequent archival by a user. We excluded these pages from the benchmark because no deltas were available. On the other hand, about 10% of the 380 pages had 10 or more versions archived, and several had 50 or more versions (the latter were all pages that were archived automatically).

5.3 Benchmark

The purpose of the benchmark was to examine the effect of several parameters on end-to-end latency in the optimistic delta system: delta size, server latency, and cache contents. We considered delta size by retrieving many pages with different characteristics. We examined the effect of server latency by varying the response time prior to sending data to the server proxy (see below). Finally, we evaluated the difference between sending deltas to a client with the past version of the page cached and one without it cached. (In the case of the unmodified proxy, without deltas, caching was irrelevant because each version of the page was retrieved exactly once.) All requests were made with Pragma: no-cache, and the pages always differed upon each retrieval. Other cases, such as when the page has not changed or the server proxy can return its cached copy, are relatively uninteresting: they either favor the optimistic approach or are equivalent between the two systems.

In each run of the benchmark, the client system retrieved each URL repeatedly, once for each version that existed. A CGI script mapped the URL into a local filename that is dependent on the next version for that URL: the first GET on /deltatest/pageN returned /deltatest/pageN/1, the second returned /2, and so on. Thus to the "browser" (actually the benchmark program) and the proxies, the same URL mapped to new versions of the page upon each request.

In addition, the CGI script read a file to determine how much of a delay it should insert before responding, which was used to simulate delay in the Internet and/or on the content provider. Longer delays permit more of an opportunity for optimistic transfers of large documents but also place a larger lower bound on end-to-end latency: if a server takes a minute to respond, then it will be at least a minute before the client proxy knows it has the current version of the page. We

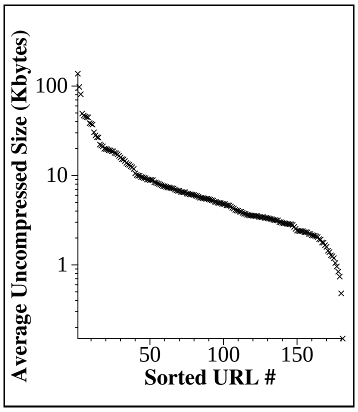


Figure 7: Average uncompressed data size across all versions of each page, sorted by size, shown on a log scale.

therefore expected that the runs with no added latency would show a high benefit from cached deltas and suffer delays from ill-placed optimistic transfers of the old copy that could not fit into the T_{wait} idle period, while runs with significant added latency would favor the optimistic approach but gain less from sending cached deltas (the amount of time saved as a fraction of total latency would decrease due to the fixed overhead).

For the current set of experiments, we were forced to restrict the range of parameters to keep the experiments tractable in a limited time frame. We did this in two respects:

- We ran experiments using fixed delays of 0s and 5s, to show extreme cases: what happens when data is nearly immediately available, and what happens when old data can be transferred during idle times.
- We restricted the maximum number of versions for a given page to 10, under the assumption that the behavior for the first 10 versions of a page would be representative of the entire set.

5.4 Results

To make it easier to interpret the results, we sorted URLs by the average uncompressed file size. Figure 7 graphs the average size (across all versions of a page) as a function of the sorted URL numbers, which are used in the other graphs below.

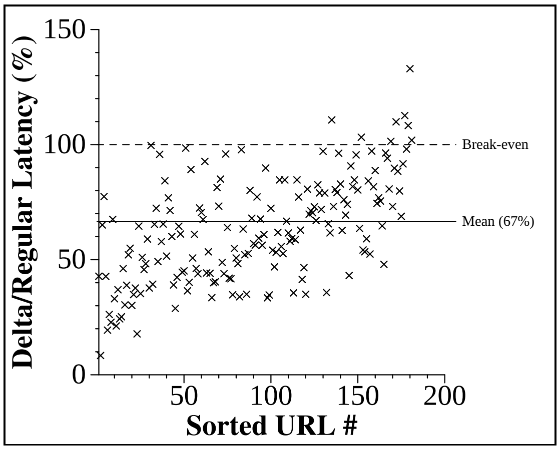
Figure 8 shows our results. Each graph plots the average ratio of end-to-end latency using the modified system to latency using the unmodified system[4]. The left column shows cases where the client proxy caches the previous version (simple deltas), while the right column shows the use of optimistic deltas. The first row shows no added content provider latency, and the second row shows 5s of added latency. The URLs are sorted in the same sequence as in Figure 7. The solid line in each graph indicates the mean of all the points in the graph, while the dashed line indicates the break-even point.

The cost of computing deltas and patching was negligible (1-2%) compared to the network transfer time and protocol processing overhead in all our experiments. Moreover, the largest measured value of overhead from computing a delta and applying the delta on the client was much less than the typical variation in the total URL fetch times.

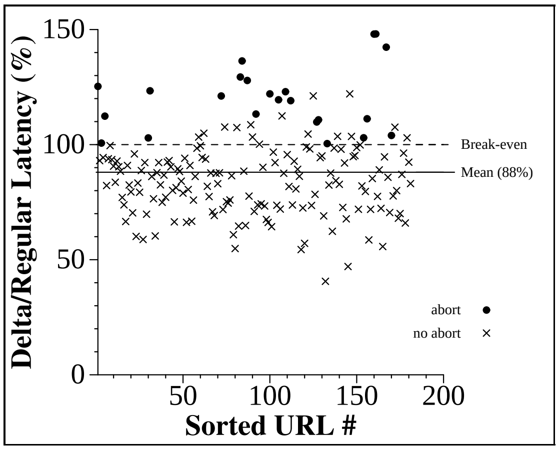
From Figure 8 we draw the following conclusions:

- The pages with the lowest index, which have the largest original file size, tended to show more improvement than the smaller pages, but although the general trend is upward as one moves right along the X-axis, there are great variations from page to page.
- As expected, without added latency, many of the pages took longer using the optimistic approach than without it. The measurements in Figure 8(b) were taken with a simple abort strategy in place. This strategy aborted only when the server proxy had finished computing the delta and the amount of remaining stale data plus the delta was more than the size of the regular response. We expect that a smarter abort strategy, such as aborting an ongoing optimistic transfer of stale data and "cutting through" new data even as it is being received if it appears that it is very different from the cached stale data, would cap the latency of our system at close to 100% of the unmodified system.
- With 5s added latency, most pages were received faster by the client using optimistic deltas, with a mean improvement of 27%. In fact, the latency for optimistic deltas with 5s added delay was consistently somewhat less than that for simple deltas with the same delay. We attribute the better performance of optimistic deltas to TCP's slow-start algorithm [14]. In the case of the optimistic deltas the transfer of the stale data opened up the TCP congestion window, so the deltas were transferred faster in this case than in the case of simple deltas,

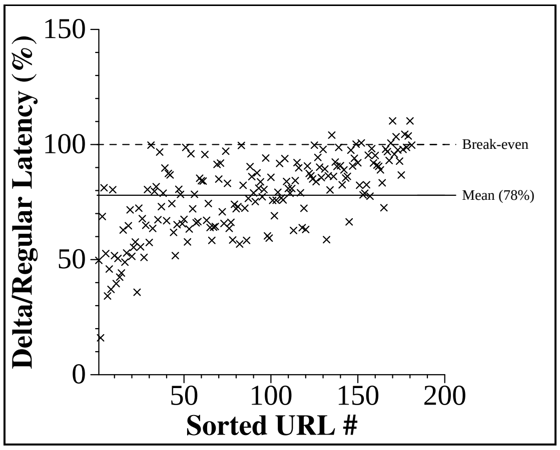
[4] In the unmodified system, there were no proxies involved and the client talked directly to the content-provider.



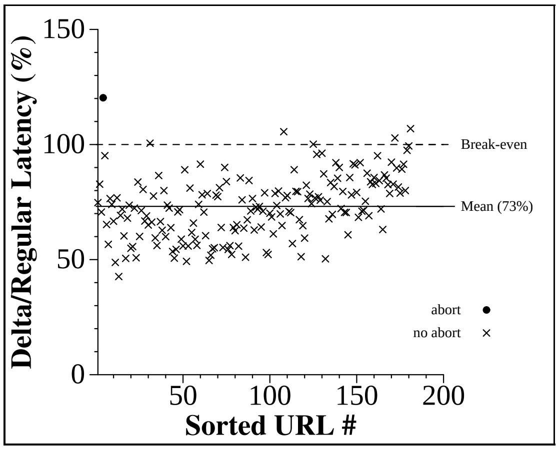
(a) Simple deltas, 0s added latency.



(b) Optimistic deltas, 0s added latency. The circles represent the 23 pages where aborts occurred.



(c) Simple deltas, 5s added latency.



(d) Optimistic deltas, 5s added latency. The circle represents the one page where aborts occurred.

Figure 8: Experimental results, showing ratios of end-to-end latency for modified versus unmodified system, varying whether old versions are cached on the client or sent optimistically by the server, and whether the content provider adds 0s or 5s of latency before returning content. Each data point represents the average across all versions for the corresponding page. The solid line in each graph indicates the mean of all the points in the graph, while the dashed line indicates the break-even point. The "simple delta" case never experienced aborts, while the "optimistic delta" case experienced aborts that are indicated with a different symbol.

where the transfer of the delta had to open up the congestion window itself.

- Nearly all of the simple deltas improved performance regardless of added latency, which one would expect. As predicted, the relative gain was generally better when the fixed overhead was lower. In the case of 0s added latency, most of the points that showed degradation were cases of only two versions being available (hence a greater likelihood of variability due to external factors) and where the deltas were 40–60% of the original file size. The overall improvement was 33%, which was the best of the four configurations.

6 Status and Future Work

At present, all of the functionality described in this paper has been implemented except for the ability of the server-side proxy to store multiple versions of the same URL. We plan to implement these features and test the system in a multi-user environment, where the same server proxy handles requests for multiple users. We believe that in this environment there will be many more cases where the server proxy has content that a particular client proxy does not, resulting in more optimistic transfers than would occur in a single-user context. One must evaluate policies for determining how many versions to keep and how many concurrent clients can be supported by a server proxy.

We plan to expand the URL comparison logic to handle the case of variants on the same URL (including the part of a GET URL that specifies CGI parameters), as described above in Section 3.1. In fact, it might be possible to hash the contents of pages to find other pages that are substantially similar and would generate small deltas.

Currently, each communication between the browser and the client proxy, or between the client and server proxies, requires a TCP setup. Persistent HTTP [17] should improve performance further, but we have not yet implemented a persistent connection in our proxy.

In the current system, deltas are generated only when the current version of a document has been received in its entirety. We intend to add *incremental* delta generation so that the delta can be sent over the slower link as content is received, and so that it is possible to abort optimistic transfers early if the delta appears to be large. It is also possible to use historical data to estimate the usefulness of sending stale data: if T_{wait} for a particular host or page is usually very small, then one might not bother with the optimistic transfer.

Finally, it should be useful to integrate prefetching into the optimistic delta system. In addition to prefetching new pages through the server proxy to the client (similar to the studies mentioned above [18, 22]), we can prefetch deltas to keep the proxies' caches better synchronized.

7 Conclusion

We have proposed an *optimistic deltas* approach to reduce the latency of accessing W^3 pages. This approach involves sending the differences between versions of a page, or deltas, to the client, instead of sending entire pages. It also permits stale data to be sent during periods of inactivity. Our approach is optimistic because it sends data that may not be needed; instead, it optimizes for the common case when pages change incrementally, at the expense of a slight overhead in the rare cases when a modification drastically changes the content of the page. In other words, we assume that in most cases when a copy cached by the proxy is deemed unusable, it is either still current, or, if it has been modified, the size of the modification is considerably smaller than the page itself.

Our study of an AT&T multi-version archive of W^3 pages confirmed the above assumption. In fact, by examining the extent to which the results of AltaVista queries with slightly different parameters differ, we showed that this assumption may even hold for dynamically generated pages. However, in general we expect that other sorts of data, such as images, should be handled specially rather than processed as deltas.

A study of the latency to obtain W^3 pages confirmed that the latency in obtaining data may often be sufficient to send stale data, for the purpose of sending a small delta once the data is available. However, performance may be degraded when latency is low and more sophisticated techniques for deciding when to abort the transfer of stale data are required.

We implemented our approach without changing the browser. Instead, we configure the browser to connect to a *client proxy* on the same machine, which in turn connects to a server proxy. These proxies have been modified to follow the optimistic deltas approach. We compared the performance of this configuration with the original system. This performance study, based on microbenchmarks, showed a significant latency reduction achieved by our approach: an average of 12-33% improvement across all pages in the study, depending on system parameters, with some transfers improved by an order of magnitude. One particularly surprising result was the effect that transferring potentially stale data had on the TCP slow-start algorithm when a link is otherwise idle, consistently improving end-to-end latency.

While a long-term experiment that would compare the performance of our approach with existing proxy caching systems on real-life workloads is needed, the experiments described in the paper strongly suggest that the optimistic delta mechanism results in a considerable reduction of W^3 latency.

Acknowledgments

Ankur Jain assisted with the analysis of HTTP latencies. Herman Rao wrote the initial version of the bimodal proxy used in these experiments, Dave Korn and Kiem-Phong Vo wrote *vdelta*, and Dave Kristol wrote *htd*. H. V. Jagadish initially suggested the use of stale pages in conjunction with deltas. Robin Chen, Tony DeSimone, Dave Korn, Bala Krishnamurthy, Ankur Jain, Jeff Mogul, Doug Monroe, Sandeep Sibal, and the anonymous USENIX referees provided comments on earlier drafts of this paper.

References

[1] Altavista. http://www.altavista.digital.com. Random URL selection at http://www.altavista.digital.com/cgi-bin/query?pg=s&target=0.

[2] Dave Belanger, David Korn, and Herman Rao. Infrastructure for wide-area software development. In *Proceedings of Sixth International Workshop on Software Configuration Management*, March 1996.

[3] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–9. ACM, October 1992.

[4] World Wide Web Consortium. Hypertext transfer protocol. http://www.w3.org/pub/WWW/Protocols/.

[5] Adam Dingle and Tomas Partl. Web cache coherence. In *Proceedings of the Fifth International WWW Conference*, May 1996. Available as http://www5conf.inria.fr/fich_html/papers/P2/Overview.html.

[6] Fred Douglis. On the role of compression in distributed systems. In *Proceedings of the Fifth ACM SIGOPS European Workshop*, Mont St.-Michel, France, September 1992. ACM.

[7] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, CA, January 1993.

[8] Fred Douglis and Thomas Ball. Tracking and viewing changes on the web. In *Proceedings of 1996 USENIX Technical Conference*, pages 165–176, San Diego, CA, January 1996.

[9] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable and Distributed Systems*, pages 39–47, October 1992.

[10] Glenn S. Fowler, David G. Korn, Steven C. North, Herman Rao, and K. Phong Vo. Libraries and file system architecture. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 2. John Wiley & Sons, New York, January 1995.

[11] Armando Fox and Eric A. Brewer. Reducing www latency and bandwidth requirements by real-time distillation. In *Proceedings of the Fifth International WWW Conference*, May 1996.

[12] James Gwertzman and Margo Seltzer. World-wide web cache consistency. In *Proceedings of 1996 USENIX Technical Conference*, pages 141–151, San Diego, CA, January 1996. Also available as http://www.eecs.harvard.edu/˜vino/web/usenix.196/.

[13] Barron C. Housel and David B. Lindquist. WebExpress: A system for optimizing web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 108–116, Rye, New York, November 1996. ACM.

[14] Van Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM Conference*, Stanford, CA, August 1988.

[15] Christopher A. Kantarjiev, Alan Demers, Ron Frederick, Robert T. Krivacic, and Mark Weiser. Experiences with X in a wireless environment. In *Proceedings USENIX Symposium on Mobile & Location-Independent Computing*, pages 117–128. USENIX, August 1993.

[16] P. Keleher, S. Dwarkadas, A.L. Cox, , and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of 1994 Winter USENIX Conference*, pages 115–131, San Francisco, CA, January 1994.

[17] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving http latency. *Computer Networks and ISDN Systems*, 28(1–2):25–35, December 1995.

[18] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *Computer Communication Review*, 26(3):22–36, 1996.

[19] James S. Plank, Jian Xu, and Rob Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[20] Bill N. Schilit, Fred Douglis, David M. Kristol, Paul Krzyzanowski, James Sienicki, and John A. Trotter. Teleweb: Loosely connected access to the world wide web. In *Proceedings of the Fifth International World Wide Web Conference*, Paris, France, May 1996.

[21] Charles L. Viles and James C. French. Availability and latency of world wide web information servers. *Computing Systems*, 8(1):61–91, Winter 1995.

[22] Stuart Wachsberg, Thomas Kunz, and Johnny Wong. Fast world-wide web browsing over low-bandwidth links. Available as http://ccnga.uwaterloo.ca/˜sbwachsb/paper.html, June 1996.