Potential benefits of delta encoding and data compression for HTTP

Jeffrey C. Mogul (Digital Equipment Corporation Western Research Laboratory)
250 University Avenue, Palo Alto, CA 94301; mogul@wrl.dec.com

Fred Douglis, **Anja Feldmann**, **Balachander Krishnamurthy** (AT&T Labs - Research)
180 Park Avenue, Florham Park, NJ 07932-0971; {douglis,anja,bala}@research.att.com

Abstract

Caching in the World Wide Web currently follows a naive model, which assumes that resources are referenced many times between changes. The model also provides no way to update a cache entry if a resource does change, except by transferring the resource's entire new value. Several previous papers have proposed updating cache entries by transferring only the differences, or ''delta,'' between the cached entry and the current value.

In this paper, we make use of dynamic traces of the full contents of HTTP messages to quantify the potential benefits of delta-encoded responses. We show that delta encoding can provide remarkable improvements in response size and response delay for an important subset of HTTP content types. We also show the added benefit of data compression, and that the combination of delta encoding and data compression yields the best results.

We propose specific extensions to the HTTP protocol for delta encoding and data compression. These extensions are compatible with existing implementations and specifications, yet allow efficient use of a variety of encoding techniques.

1. Introduction

The World Wide Web is a distributed system, and so often benefits from caching to reduce retrieval delays. Retrieval of a Web resource (such as document, image, icon, or applet) over the Internet or other wide-area network usually takes enough time that the delay is over the human threshold of perception. Often, that delay is measured in seconds. Caching can often eliminate or significantly reduce retrieval delays.

Many Web resources change over time, so a practical caching approach must include a coherency mechanism, to avoid presenting stale information to the user. Originally, the Hypertext Transfer Protocol (HTTP) provided little support for caching, but under operational pressures, it quickly evolved to support a simple mechanism for maintaining cache coherency.

In HTTP/1.0 [3], the server may supply a ''last-modified'' timestamp with a response. If a client stores this response in a cache entry, and then later wishes to re-use the response, it may transmit a request message with an ''if-modified-since'' field containing that timestamp; this is known as a *conditional* retrieval. Upon receiving a conditional request, the server may either reply with a full response, or, if the resource has not changed, it may send an abbreviated reply, indicating that the client's cache entry is still valid. HTTP/1.0 also includes a means for the server to indicate, via an ''expires'' timestamp, that a response will be valid until that time; if so, a client may use a cached copy of the response until that time, without first validating it using a conditional retrieval.

The proposed HTTP/1.1 specification [6] adds many new features to improve cache coherency and performance. However, it preserves the all-or-none model for responses to conditional retrievals: either the server indicates that the resource value has not changed at all, or it must transmit the entire current value.

Common sense suggests (and traces confirm), however, that even when a Web resource does change, the new instance is often substantially similar to the old one. If the difference (or *delta*) between the two instances could be sent to the client instead of the entire new instance, a client holding a cached copy of the old instance could apply the delta to construct the new version. In a world of finite bandwidth, the reduction in response size and delay could be significant.

One can think of deltas as a way to squeeze as much benefit as possible from client and proxy caches. Rather than treating an entire response as the ''cache line,'' with deltas we can treat arbitrary pieces of a cached response as the replaceable unit, and avoid transferring pieces that have not changed.

In this paper, we make use of dynamic traces of the full contents of HTTP messages to quantify the potential benefits of delta-encoded responses. Although previous papers [2, 8, 17] have proposed the use of delta encoding, ours is the first to use realistic traces to quantify the benefits. Our use of traces from two different sites increases our confidence in the results.

We show that delta encoding can provide remarkable improvements in response-size and response-delay for an important subset of HTTP content types. We also show the added benefit of data compression, and that the combination of delta encoding and data compression yields the best results.

We propose specific extensions to the HTTP protocol for delta encoding and data compression. These extensions are compatible with existing implementations and specifications, yet allow efficient use of a variety of encoding techniques.

2. Related work

The idea of delta-encoding to reduce communication or storage costs is not new. For example, the MPEG-1 video compression standard transmits occasional still-image frames, but most of the frames sent are encoded (to oversimplify) as

changes from an adjacent frame. The SCCS and RCS [15] systems for software version control represent intermediate versions as deltas; SCCS starts with an original version and encodes subsequent ones with forward deltas, whereas RCS encodes previous versions as reverse deltas from their successors. Jacobson's technique for compressing IP and TCP headers over slow links [10] uses a clever, highly specialized form of delta encoding.

In spite of this history, it appears to have taken several years before anyone thought of applying delta encoding to HTTP, perhaps because the development of HTTP caching has been somewhat haphazard. The first published suggestion for delta encoding appears to have been by Williams et al. in a paper about HTTP cache removal policies [17], but these authors did not elaborate on their design until later [16].

The possibility of compressing HTTP messages seems to have an longer history, going back at least to the early drafts of the HTTP/1.0 specification. However, until recently, it appears that nobody had attempted to quantify the potential benefits of loss-free compression, although the GloMop project [7] did explore the use of lossy compression. Also, our traces suggest that few existing client implementations offer to accept compressed encodings of arbitrary responses (apparently, Lynx is the one exception). (Before the Web was an issue, Douglis [4] wrote generally about compression in distributed systems.)

The WebExpress project [8] appears to be the first published description of an implementation of delta encoding for HTTP (which they call ''differencing''). WebExpress is aimed specifically at wireless environments, and includes a number of orthogonal optimizations. Also, the WebExpress design does not propose changing the HTTP protocol itself, but rather uses a pair of interposed proxies to convert the HTTP message stream into an optimized form. The results reported for WebExpress differencing are impressive, but are limited to a few selected benchmarks.

Banga et al. [2] describe the use of *optimistic* deltas, in which a layer of interposed proxies on either end of a slow link collaborate to reduce latency. If the client-side proxy has a cached copy of a resource, the server-side proxy can simply send a delta. If only the server-side proxy has a cached copy, it may optimistically send its (possibly stale) copy to the client-side proxy, followed (if necessary) by a delta once the server-side proxy has validated its own cache entry with the origin server. The use of optimistic deltas, unlike delta encoding, actually increases the number of bytes sent over the network, in an attempt to improve latency by anticipating a ''Not Modified'' response from the origin server. The optimistic delta paper, like the WebExpress paper, did not propose a change to the HTTP protocol itself, and reported results only for a small set of selected URLs.

We are also analyzing the same traces to study the rate of change of Web resources [5].

3. Motivation and methodology

Although two previous papers [2, 8] have shown that compression and delta encoding could improve HTTP performance for selected sets of resources, these did not analyze traces from ''live'' users to see if the benefits would apply in practice. Also, these two projects both assumed that the HTTP protocol could not be modified, and so relied on interposing proxy systems at either end of the slowest link. This approach adds extra store-and-forward latency, and may not always be feasible, so we wanted to examine the benefits of end-to-end delta encoding and compression, as an extension to the HTTP protocol.

Although we propose such an extension in section 7, we have not yet finished an implementation. In this paper, we use a trace-based analysis to quantify the potential benefits from both proxy-based and end-to-end applications of compression and delta encoding. Both of these applications are supported by our proposed changes to HTTP. We also analyze the utility of these techniques for various different HTTP content-types (such as HTML, plain text, and image formats), and for several ways of grouping responses to HTTP queries. We look at several different algorithms for both delta encoding and data compression, and we examine the relative performance of high-level compression and modem-based compression algorithms.

We used two different traces in our study, made at busy Internet connection points for two large corporations. One of the traces was obtained by instrumenting a proxy; the other was made by capturing raw network packets and reconstructing the data stream. Both traces captured only references to Internet servers outside these corporations, and did not include any ''inbound'' requests. Because the two traces represent different protocol levels, time scales, user communities, and criteria for pre-filtering the trace, they give us several views of ''real life'' reference streams, although certainly not of all possible environments.

Since the raw traces include a lot of sensitive information, for reasons of privacy and security the authors of this paper were not able to share the traces with each other. That, and the use of different trace-collection methods, led us to do somewhat different analyses on the two trace sets.

3.1. Obtaining proxy traces

Some large user communities often gain access to the Web via a proxy server. Proxies are typically installed to provide shared caches, and to allow controlled Web access across a security firewall. A proxy is a convenient place to obtain a realistic trace of Web activity, especially if it has a large user community, because (unlike a passive monitor) it guarantees that all interesting activity can be traced without loss, regardless of the offered load. Using a proxy server, instead of a passive monitor, to gather traces also simplifies the task, since it eliminates the need to reconstruct data streams from TCP packets.

3.1.1. Tracing environment

We were able to collect traces at a proxy site that serves a large fraction of the clients on the internal network of Digital Equipment Corporation. Digital's network is isolated from the Internet by firewalls, and so all Internet access is mediated by proxy relays. This site, located in Palo Alto, California, and operated by Digital's Network Systems Laboratory, relayed more than a million HTTP requests each weekday. The proxy load was spread, more or less equally, across two AlphaStation 250 4/266 systems running Digital UNIX V3.2C.

To collect these traces, we modified version 3.0 of the CERN httpd code, which may be used as either a proxy or a server. We made minimal modifications, to reduce the risk of introducing bugs or significant performance effects. The modified proxy code traces a selected subset of the requests it receives:

- Only requests going to HTTP servers (i.e., not FTP or Gopher).
- Only those requests whose URL does *not* end in one of a set of suffixes, such as ''.gif'', ''.jpeg'', ''.au'', ''.mpeg'', etc. These URLs were omitted in order to reduce the size of the trace logs.

This pre-filtering considered only the URL in the request, not the HTTP Content-type in the response; therefore, many responses with unwanted content-types leaked through.

For each request that is traced, the proxy records in a disk file the client and server IP addresses, timestamps for various events in processing the request, and the complete HTTP header and body of both the request and the response.

This particular proxy installation was configured not to cache HTTP responses, for a variety of logistical reasons. This means that a number of the responses in the trace contained a full body (i.e., HTTP status code = 200) when, if the proxy had been operating as a cache, they might have instead been ''Not Modified'' responses with no body (i.e., HTTP status code = 304). The precise number of such responses would depend on the size of the proxy cache and its replacement policy. We still received many ''Not Modified'' responses, because most of the client hosts employ caches.

3.1.2. Trace duration

We collected traces for almost 45 hours, starting in the afternoon of Wednesday, December 4, 1996, and ending in the morning of December 6. During this period, the proxy site handled about 2771975 requests, 504736 of which resulted in complete trace records, and generated almost 9 GBytes of trace file data. (Many requests were omitted by the pre-filtering step, or because they were terminated by the requesting client.) While tracing was in progress, approximately 8078 distinct client hosts used the proxy site, which (including the untraced requests) forwarded almost 21 GBytes of response bodies, in addition to HTTP message headers (whose length is not shown in the standard proxy log format).

3.2. Obtaining packet-level traces

When a large user community is not constrained to use a proxy to reach the Internet, the option of instrumenting a proxy is not available. Instead, one can passively monitor the network segment connecting this community to the Internet, and reconstruct the data stream from the packets captured.

We collected a packet-level trace at the connection between the Internet and the network of AT&T Labs -- Research, in Murray Hill, New Jersey. This trace represents a much smaller client population than the proxy trace. All packets between internal users and TCP port 80 (the default HTTP server port, used for more than 99.4% of the HTTP references seen at this site) on external servers were captured using *tcpdump* [13]. A negligible number of packets were lost due to buffer overruns. The raw packet traces were later reassembled into individual TCP streams. (This is a complex process, described in more detail in [14].) These streams were then split into files representing the body of each successful request and a log containing information about URLs, timestamps, and request and response headers.

Between Friday, November 8 and Monday, November 25, 1996, (17 days) we collected a total of 51,100,000 packets, corresponding to roughly 19 Gbytes of raw data. Unlike the proxy-based trace, this one was not pre-filtered to eliminate requests based on their content-type or URL extension.

4. Trace analysis software

Because the two traces were obtained using different techniques, we had to write two different systems to analyze them.

4.1. Proxy trace analysis software

We wrote software to parse the trace files and extract relevant HTTP header fields. The analysis software then groups the references by unique resource (URL), and to *instances* of a resource. We use the term *instance* to describe a snapshot in the lifetime of a resource. In our analyses, we group responses for a given URL into a single instance if the responses have identical last-modified timestamps and response body lengths. There may be one or more instances per resource, and one or more references per instance.

The interesting references, for the purpose of this paper, were those for which the response carried a full message body (i.e., HTTP status code = 200), since it is only meaningful to compute the difference between response bodies for just these references. Once the analysis program has grouped the references into instances, it then iterates through the references, looking for any full-body reference which follows a previous full-body reference to a different instance of the same resource. (If two references involve the same instance, then presumably the server should have sent a ''Not Modified'' response, with status = 304 and no response body, rather than two identical responses.)

For each such pair of full-body responses for different instances of a resource, the analysis program computes a delta encoding for the second response, based on the first response. This is done using several different delta-encoding algorithms; the program then reports the size of the resulting response bodies for each of these algorithms.

The delta computation is done by extracting the relevant response bodies from the trace log files into temporary files, then invoking one of the delta-encoding algorithms on these files, and measuring the size of the output.

The delta-encoding algorithms that we applied include:

- *diff -e*: a fairly compact format generated by the UNIX ''diff'' command, for use as input to the ''ed'' text editor (rather than for direct use by humans).[1]
- compressed *diff -e*: the output of *diff -e*, but compressed using the *gzip* program.
- *vdelta*: this program inherently compresses its output [9].

[1]Because HTML files include lines of arbitrary length, and because the standard *ed* editor cannot handle long lines, actual application of this technique would require use of an improved version of *ed* [11].

We used *diff* to show how well a fairly naive, but easily available algorithm would perform. We also used *vdelta*, a more elaborate algorithm, because it was identified by Hunt et al. as the best overall delta algorithm, based on both output size and running time [9].

The UNIX *diff* program does not work on binary-format input files, so we restricted its application to responses whose Content-type field indicated a non-binary format; these included ''text/html'', ''application/postscript'', ''text/plain'', ''application/x-javascript'', and several other formats. *Vdelta* was used on all formats.

4.2. Packet-level trace analysis software

We processed the individual response body files derived from the packet trace (see section 3.2) using a *Perl* script to compute the size of the deltas between pairs of sequentially adjacent full-body responses for the same URL, and the size of a compressed version of each full-body response.

While the proxy-based trace, by construction, omitted many of the binary-format responses in the reference stream, the packet-based trace included all content types. We classified these into ''textual'' and ''non-textual'' responses, using the URL extension, the Content-type HTTP response-header, or (as a last resort) by scanning the file using a variant of the UNIX *file* command.

In our traces we saw 1,366,401 requests, of which 26,501 (1.9%) had gaps, due to packet losses and artifacts of the techniques required to process 19 Gbytes of trace data (see [14] for more details on trace-processing). Another 38,589 (2.8%) of the requests were detected as duplicates created by artifacts of the processing techniques. Both these sets were excluded from further analysis. To further restrict our analysis only to those references where the client received the complete HTTP response body, we included only those TCP streams for which we collected SYN and FIN packets from both client and server, or for which the size of the reassembled response body equaled the size specified in the Content-length field of the HTTP response. This left us with 1,080,143 usable responses (79% of the total).

5. Results of trace analysis

This section describes the results of our analysis of the proxy and packet-level traces.

5.1. Overall response statistics for the proxy trace

The 504736 complete records in the proxy trace represent the activity of 7411 distinct client hosts, accessing 22034 distinct servers, referencing 238663 distinct resources (URLs). Of these URLs, 100780 contained ''?'' and are classified as query URLs; these had 12004 unique prefixes (up to the first ''?'' character). The requests totalled 149 MBytes (mean = 311 bytes/message). The request headers totalled 146 MBytes (mean = 306 bytes), and the response headers totalled 81 MBytes (mean = 161 bytes). 377962 of the responses carried a full body, for a total of 2450 MB (mean = 6798 bytes); most of the other types of responses do not carry much (or any) information in their bodies. 17211 (3.4%) of the responses carried a status code of 304 (Not Modified).

Note that the mean response body size for all of the references handled by the proxy site (7773 bytes) is somewhat larger than the mean size of the response bodies captured in the traces. This is probably because the data types, especially images, that were filtered out of the trace based on URL extension tend to be somewhat larger than average.

5.2. Overall response statistics for the packet-level trace

The 1090025 usable records in the packet-level trace represent the activity of 465 clients, accessing 20956 servers, referencing 625657 distinct URLs. Of these URLs, 105010 contained ''?'' and are classified as query URLs; these had 15438 unique prefixes (up to the first ''?'' character). 26216 of the URLs contained ''cgi'', and so are probably references to CGI scripts.

The mean request and response header sizes were 281 bytes and 173 bytes, respectively. 828837 of the responses carried a full body, for a total of 6239 MB of response bodies (mean = 7882 bytes for full-body responses). 145273 (13.4%) of the responses carried a status code of 304 (Not Modified). We omitted from our subsequent analyses 8839 full-body responses for which we did not have trustworthy timing data, leaving a total of 819998 fully-analyzed responses.

The mean response size for the packet-level trace is higher than that for the proxy trace, perhaps because the latter excludes binary-format responses, some of which tend to be large. The difference may also simply reflect the different user communities.

5.3. Characteristics of responses

Figure 5-1 shows cumulative distributions for total response sizes, and for the response-body size for full-body responses, for the proxy trace. The distributions for the packet-level trace are similar, and omitted for reasons of space. The median full-response body size was 3976 bytes for the proxy trace, and 3210 bytes for the packet-level traces, which implies that the packet-level trace showed larger variance in response size. Note that over 99% of the bytes carried in response bodies, in this trace, were carried in the status-200 responses; this is normal, since HTTP responses with other status codes either carry no body, or a very small one.

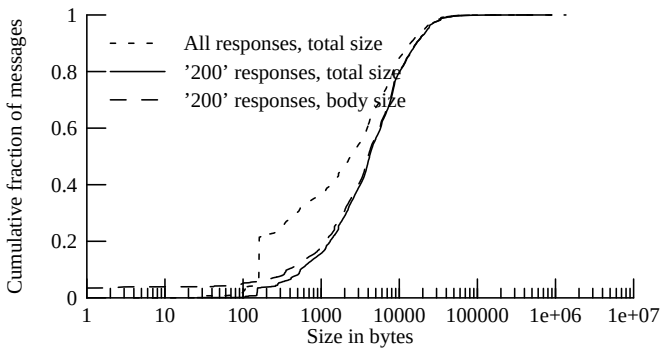

Figure 5-1: Cumulative distributions of response sizes (proxy trace)

Delta encoding and/or caching are only useful when the reference stream includes at least two references to the same URL (for delta encoding), or two references to the same *(URL, last-modified-date)* instance (for caching). Figure 5-2 shows the cumulative distributions in the proxy trace of the number of references per URL, and per instance. Curves are shown both for all traced references, and for those references that

resulted in a full-body response. We logged at least two full-body responses for more than half (57%) of the URLs in the trace, but only did so for 18% of the instances. In other words, resource values seem to change often enough that relatively few such values are seen twice, even for URLs that are referenced more than once. (An alternative explanation is that the values do not change, but the origin servers provide responses that do not allow caching.)

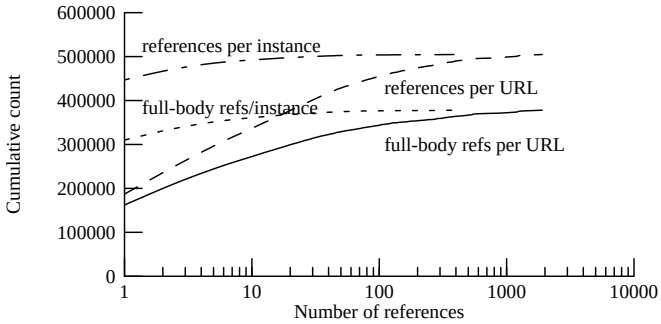

Figure 5-2: Cumulative distributions of reference counts (proxy trace)

5.4. Calculation of savings

We define a response as *delta-eligible* if the trace included at least one previous status-200 response for a different instance of the same resource. (We did not include any response that conveyed an instance identical to the previous response for the same URL, which probably would not have been received by a caching proxy.) In the proxy trace, 142913 of the 377962 status-200 responses (37.8%) were delta-eligible. In the packet-level trace, 83991 of the 819998 status-200 responses (10.2%) were delta-eligible. In the proxy trace, only 18% of the status-200 responses were excluded from consideration for being identical, compared to 32% for the packet-level trace.

We attribute much of the difference in the number of delta-eligible responses to the slower rate of change of image responses, which were mostly pre-filtered out of the proxy trace. In the packet-level trace, 66% of the status-200 responses were GIF or JPEG images, but only 3.5% of those responses were delta-eligible; in contrast, 25% of the status-200 HTML responses were delta-eligible. Some additional part of the discrepancy may be the result of the smaller client population in the packet-level traces, which might lead to fewer opportunities for sharing.

Our first analysis is based on the assumption that the deltas would be requested by the proxy, and applied at the proxy to responses in its cache; if this were only done at the individual clients, far fewer of the responses would be delta-eligible. In section 5.5.1, we analyze the per-client reference streams separately, as if the deltas were applied at the clients.

For each of the delta-eligible responses, we computed a delta using the *vdelta* program, based on the previous status-200 instance in the trace, and two compressed versions of the response, using *gzip* and *vdelta*. For those responses whose HTTP Content-type field indicated an ASCII text format (‘‘text/html’’, ‘‘text/plain’’, ‘‘application/postscript’’, and a few others), we also computed a delta using the UNIX *diff -e* command, and a compressed version of this delta, using *gzip*. 71446 (50%) of the delta-eligible responses in the proxy trace were text-format responses, as were 54856 (65%) of the delta-eligible responses in the packet-level trace.

For each response, and for each of the four computations, we measured the number of response-body bytes saved (if any). We also estimated the amount of retrieval time that would have been saved for that response, had the delta or compression technique been used. (We did not include the computational costs of encoding or decoding; see section 6 for those costs.)

Our estimate of the improvement in retrieval time is simplistic, but probably conservative. We estimated the transfer time for the response from the timestamps in our traces, and then multiplied that estimate by the fraction of bytes saved to obtain a prediction for the improved response transfer time. However, in the proxy traces it is not possible to separate the time to transmit the request from the time to receive the first part of the response, so our estimate of the original transfer time is high. We compensated for that by computing two estimates for the transfer time, one which is high (because it includes the request time) and one which is low (because it does not include either the request time, or the time for receiving the first bytes of the response). We multiplied the fraction of bytes saved by the latter (low) estimate, and then divided the result by the former (high) estimate, to arrive at our estimate of the fraction of time saved.

For the packet-level traces, we were able to partially validate this model. We measured the time it actually took to receive the packets including the first *N* bytes of an *M*-byte transfer, where *N* is the number of bytes that would have been seen if delta encoding or compression had been used. The results agree with our simpler model to within about 10%, but are still conservative (because we did not model the reduction in the size of the last data packet).

Figure 5-3 shows the distribution of latencies for the important steps in the retrieval of full-body (status-200) responses from the proxy trace. The four steps measured are: (1) the time for the proxy to read and parse the client’s request, (2) the time to connect to the server (including any DNS lookup cost), (3) the time to forward the request and to receive the first bytes of response (i.e., the first read() system call), and (4) the time to receive the rest of the response, if any. (The spikes at 5000 msec may represent a scheduling anomaly in the proxy software; the spike at 10000 msec simply represents the sum of two 5000-msec delays.) We used the sum of steps 3 and 4 as the high estimate for transfer time, and step 4 by itself as the low estimate.

Figure 5-4 shows a similar view of the packet-level trace. The individual steps are somewhat different (the packet-level trace exposes finer detail), and the latencies are all measured from the start of the connection (the client’s SYN packet). The steps are (1) arrival of the server’s SYN, (2) first packet of the HTTP request, (3) first packet of the response header, (4) first packet of the response body, and (5) end of the response. The figure also shows the transfer time for the response body, which is similar to (but smaller than) the transfer-time estimate used in figure 5-3.

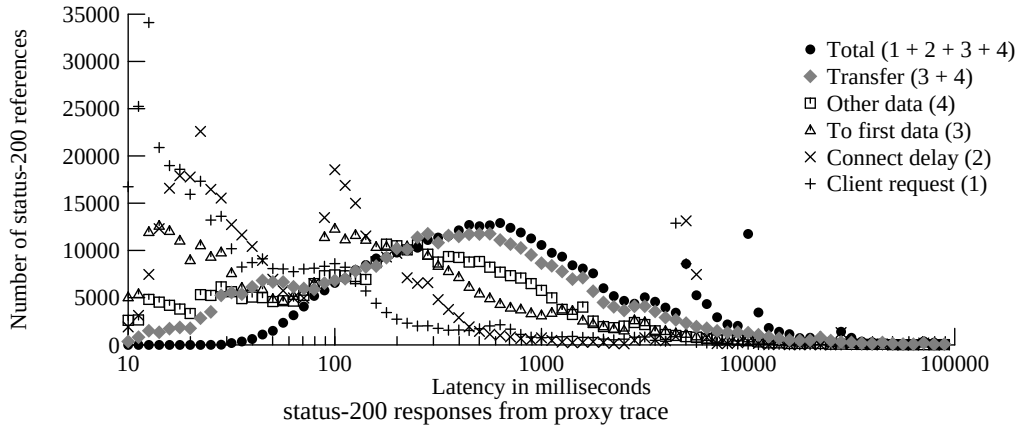

Figure 5-3: Distribution of latencies for various phases of retrieval (proxy trace)

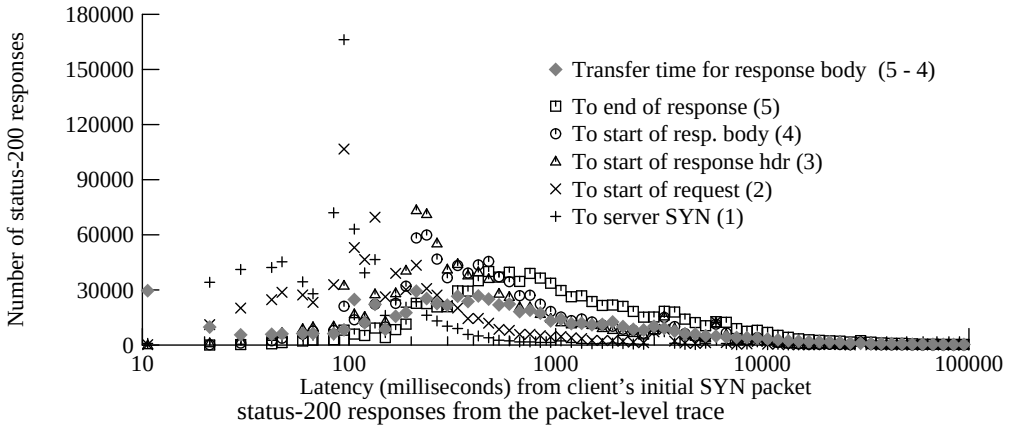

Figure 5-4: Distribution of cumulative latencies to various phases (packet-level trace)

5.5. Net savings due to deltas and compression

Table 5-1 shows (for the proxy trace) how many of the responses were improved, and by how much. The left-hand columns show the results relative to just the delta-eligible responses; the right-hand columns show the same results, but expressed as a fraction of all full-body responses. Because these account for more than 99% of the response-body bytes in the traces, this is also nearly equivalent to the overall improvement for all traced responses.

In table 5-1, the row labeled ‘‘unchanged’’ shows how many delta-eligible responses would have resulted in a zero-length delta. (These ‘‘unchanged’’ responses are delta-eligible because their last-modified time has changed, but their The rows labelled ‘‘diff -e’’, ‘‘diff -e | gzip’’, and ‘‘vdelta’’ show the delta-encoding results only for those responses where there is at least some difference between a delta-eligible response and the previous instance. Two other lines show the results if the unchanged responses are included. The rows labelled ‘‘vdelta compress’’ and ‘‘gzip compress’’ show the results for compressing the responses, without using any delta encoding. The final row shows the overall improvement (not including unchanged responses), assuming that the server uses whichever of these algorithms minimizes each response.

It is encouraging that, out of all of the full-body responses, table 5-1 shows over 30% of the response-body bytes could be saved by using *vdelta* to do delta encoding. This implies that the use of delta encoding would provide significant benefits for textual content-types. It is remarkable that over 83% of the response-body bytes could be saved for delta-eligible responses; that is, in those cases where the recipient already has a cached copy of a prior instance. And while it appears that the potential savings in transmission time is smaller than the savings in response bytes, the response-time calculation is quite conservative (as noted earlier).

For the 112354 delta-eligible responses where the delta was not zero-length, *vdelta* gave the best result 94% of the time. *diff -e* without compression and with compression was best for about 3% and 2% of the cases, respectively, and simply compressing the response with *gzip* worked best in 1% of the cases. The *vdelta* approach clearly works best, but just using *diff -e* would save 42% of the response-body bytes for delta-eligible responses. That is, almost half of the bytes in ‘‘new’’ responses are easily shown to be the same as in their predecessors.

Table 5-2 shows, for the responses in the packet-level trace, how much improvement would be available using deltas if one introduced a proxy at the point where the trace was made. The results in table 5-2 are somewhat different from those in table 5-1, for several reasons. The packet-level trace included a larger set of non-textual content types, which leads to a reduction in the effectiveness of delta encoding and compression (see section 5.7). Because the packet-level trace analysis uses a somewhat more accurate (and so less conservative) model for the savings in transfer time, similar reductions in the number of bytes transferred lead to different reductions in transfer time.

	Relative to delta-eligible responses *N* = 142913, 906 MBytes, 173416 seconds						Relative to all status-200 responses *N* = 377962, 2462 MBytes, 557373 seconds					
Computation	**Improved references**		**MBytes saved**		**Retrieval time saved**		**Improved references**		**MBytes saved**		**Retrieval time saved**	
unchanged	30559	(21.4%)	164	(18.2%)	14902	(8.6%)	30559	(8.1%)	164	(6.7%)	14902	(2.7%)
diff -e	37637	(26.3%)	214	(23.6%)	22179	(12.8%)	37637	(10.0%)	214	(8.7%)	22179	(4.0%)
diff -e (inc. *unchanged*)	68196	(47.7%)	378	(41.8%)	37082	(21.4%)	68196	(18.0%)	378	(15.5%)	37082	(6.7%)
diff -e | gzip	39649	(27.7%)	265	(29.3%)	31441	(18.1%)	39649	(10.5%)	265	(10.8%)	31441	(5.6%)
vdelta	111096	(77.7%)	587	(64.8%)	53330	(30.8%)	111096	(29.4%)	587	(24.0%)	53330	(9.6%)
vdelta (inc. *unchanged*)	141655	(99.1%)	751	(83.0%)	68232	(39.3%)	141655	(37.5%)	751	(30.7%)	68232	(12.2%)
vdelta compress	110872	(77.6%)	319	(35.3%)	461059	(26.6%)	320799	(84.9%)	918	(37.5%)	120999	(21.7%)
gzip compress	105529	(73.8%)	365	(40.3%)	53294	(30.7%)	284551	(75.3%)	947	(38.7%)	121886	(21.9%)
best algorithm above	141693	(99.1%)	790	(83.2%)	68435	(39.5%)	352912	(93.4%)	1452	(59.3%)	158712	(28.5%)

Table 5-1: Improvements assuming deltas are applied at proxy (proxy trace)

	Relative to delta-eligible responses *N* = 83991, 645 MBytes, 195814 seconds						Relative to all status-200 responses *N* = 819998, 6216 MBytes, 2053775 seconds					
Computation	**Improved References**		**MBytes saved**		**Retrieval time saved**		**Improved References**		**MBytes saved**		**Retrieval time saved**	
unchanged	26489	(31.5%)	184	(28.5%)	37028	(18.9%)	26489	(3.1%)	184	(2.9%)	37028	(1.8%)
diff -e	37318	(44.4%)	203	(31.5%)	47063	(24.0%)	37318	(4.4%)	203	(3.2%)	47063	(2.2%)
diff -e (inc. unchanged)	63807	(76.0%)	387	(60.0%)	84091	(42.9%)	63807	(7.5%)	387	(6.0%)	84091	(4.0%)
diff -e | gzip	40063	(47.7%)	246	(38.2%)	62511	(31.9%)	40063	(4.7%)	246	(3.8%)	62511	(3.0%)
vdelta	57151	(68.0%)	362	(56.2%)	81572	(41.7%)	57151	(6.8%)	362	(5.7%)	81572	(3.9%)
vdelta (inc. unchanged)	83640	(99.6%)	546	(84.7%)	118600	(60.6%)	83640	(9.9%)	546	(8.5%)	118600	(5.6%)
vdelta compress	73414	(87.4%)	270	(41.9%)	66411	(33.9%)	589819	(71.9%)	1054	(17.0%)	248565	(12.1%)
gzip compress	70859	(84.4%)	307	(47.6%)	75321	(38.5%)	596949	(72.8%)	1230	(19.8%)	291300	(14.2%)

Table 5-2: Improvements assuming deltas are applied at a proxy (packet-level trace)

Taken together, the results in tables 5-1 and 5-2 imply that if delta encoding is possible, then it is usually the best way to transmit a changed response. If delta encoding is not possible, such as the first retrieval of a resource in a reference stream, then data compression is usually valuable.

5.5.1. Analysis assuming client-applied deltas

Table 5-3 shows (for the proxy trace) what the results would be if the deltas were applied individually by each client of the proxy, rather than by the proxy itself. For delta-eligible responses, client-applied deltas perform about as well as proxy-applied deltas. However, a much smaller fraction of the responses are delta-eligible at the individual clients (19% instead of 37.8%), and so the overall improvement from delta encoding is also much smaller. In other words, the success of delta encoding depends somewhat on the large, shared cache that a proxy would provide. Alternatively, a reference stream longer than our two-day trace might show a larger fraction of per-client delta-eligible responses.

5.6. Distribution of savings

Tables 5-1, 5-2, and 5-3 report mean values for improvements in the number of bytes saved, and the amount of time saved. One would not expect delta encoding to provide the same improvement for every delta-eligible response. In some cases, especially for small responses or major changes, delta encoding can save only a small fraction of the bytes. In other cases, such as a small change in a large response, delta encoding can save most of the response bytes. Figure 5-5 shows the distribution of the fraction of response bytes saved, for all delta-eligible responses in the proxy trace. (Note that the vertical axis is a log scale.)

Although delta encoding saves few or no bytes for many of the delta-eligible responses, the bimodal distribution in figure 5-5 suggests that when delta encoding does work at all, it saves most of the bytes of a response. In fact, for delta-eligible responses in the proxy trace, the median number of bytes saved per response by delta encoding using *vdelta* is 3051 bytes (compared to a mean of 5517 bytes). For half of the delta-eligible responses, *vdelta* saved at least 98.5% of the response-body bytes (this includes cases where the size of the

	Relative to delta-eligible responses *N* = 71804, 358 Mbytes, 110273 seconds						Relative to all status-200 responses *N* = 377962, 2450 MBytes, 557373 seconds					
Computation	**Improved references**		**MBytes saved**		**Retrieval time saved**		**Improved references**		**MBytes saved**		**Retrieval time saved**	
unchanged	17660	(24.6%)	67	(18.8%)	7162	(6.5%)	17660	(4.7%)	67	(2.7%)	7162	(1.3%)
diff -e	23809	(33.2%)	129	(36.1%)	14895	(13.5%)	23809	(6.3%)	129	(5.3%)	14895	(2.7%)
diff -e (inc. *unchanged*)	41469	(57.8%)	196	(54.9%)	22057	(20.0%)	41469	(11.0%)	196	(8.0%)	22057	(4.0%)
diff -e | gzip	25323	(35.3%)	162	(45.4%)	22117	(20.1%)	25323	(6.7%)	162	(6.6%)	22117	(4.0%)
vdelta	53170	(74.0%)	246	(68.7%)	33017	(29.9%)	53170	(14.1%)	246	(10.0%)	33017	(5.9%)
vdelta (inc. *unchanged*)	70830	(98.6%)	313	(87.4%)	40179	(36.4%)	70830	(18.7%)	313	(12.8%)	40179	(7.2%)

Table 5-3: Improvements assuming deltas are applied at individual clients (proxy trace)

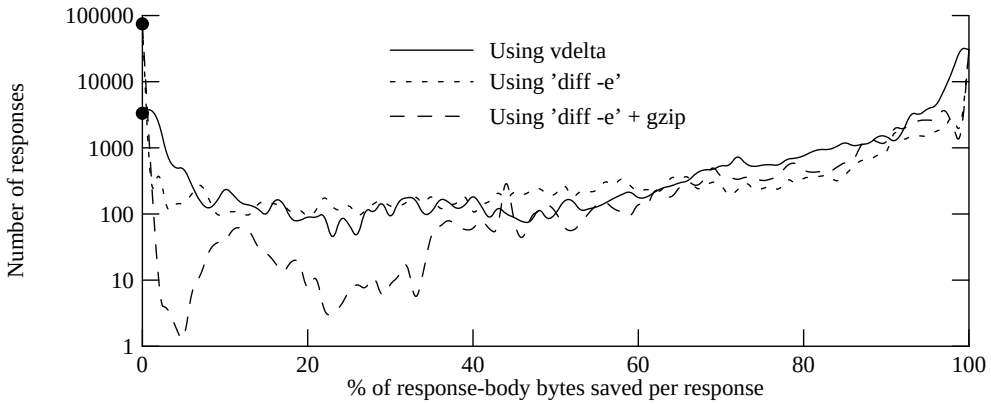

Figure 5-5: Distribution of response-body bytes saved for delta-eligible responses (proxy trace)

delta is zero, because the response value was unchanged). This is encouraging, since it implies that the small overhead of the extra HTTP protocol headers required to support delta encoding will not eat up most of the benefit.

5.7. Influence of content-type on coding effectiveness

The output size of delta-encoding and data-compression algorithms depends on the nature of the input [9, 12], and so, in the case of HTTP, on the content-type of a response. The effectiveness of delta encoding also depends on the amount by which the two versions differ, which might also vary with content-type. We subdivided the packet-level traces by content-type and analyzed each subset independently, to see how important these dependencies are in practice.

Table 5-4 shows, first of all, what fraction of the delta-eligible responses had bodies that were entirely unchanged from the previous instance. This might happen because the two requests came from separate clients, or because the server was unable to determine that an ‘‘If-Modified-Since’’ request in fact refers to an unmodified resource, or because while the resource body was not modified, some important part of the response headers did change. The table also shows other type-specific differences in the data; for example, ‘‘text/html’’ responses change more often than ‘‘text/plain’’ responses, but the ‘‘text/plain’’ responses that remain unchanged are smaller than the ‘‘text/plain’’ responses that do change. The last column shows a conservative estimate for the amount of time wasted in the transmission of unchanged responses.

Table 5-5 show the delta-encoding effectiveness, broken down by content-type, for *vdelta*. This table also shows a dependency on content-type; for example, delta encoding of changed responses seems to be more effective for ‘‘application/octet-stream’’ resources than for ‘‘text/html’’ resources. (Most ‘‘octet-stream’’ resources seem to be associated with the PointCast application.) Somewhat surprisingly, the *vdelta* algorithm improved about half of the ‘‘image/gif’’ and ‘‘image/jpeg’’ responses, albeit not reducing the byte-counts by very much (both these image formats are already compressed). We suspect that the savings may come from eliding redundant header information in these formats.

The apparent scarcity of delta-eligible images greatly reduces the utility of delta encoding when it is viewed in the context of the entire reference stream. However, we believe that in many bandwidth-constrained contexts, many users avoid the use of images, which suggests that delta encoding would be especially applicable in these contexts.

Table 5-6 shows the effectiveness of compression, using the *gzip* program, broken down by content-type. Although a majority of the responses overall were improved by compression, for some content-types compression was much less effective. It is not surprising that ‘‘image/gif’’ and ‘‘image/jpeg’’ responses could not be compressed much, since these formats are already compressed when generated. The ‘‘application/x-msnwebqt’’ responses (used in a stock-quote application) compressed nicely, but doing so would not save much transfer time at all, because the responses are already quite short.

Content-type	Delta-eligible Refs	MBytes	Total time	Refs unchanged	Bytes unchanged	Time wasted
All delta-eligible	83991	645	195814	31.5%	28.5%	18.9%
text/html	45812	400	130378	32.1%	33.4%	22.3%
application/octet-stream	17797	124	31767	0.1%	0.1%	0.1%
image/gif	17024	88	26117	60.5%	42.0%	22.8%
image/jpeg	2101	25	5397	52.5%	43.1%	32.7%
text/plain	493	2	459	22.1%	21.1%	13.0%
application/x-msnwebqt	400	0	93	36.0%	28.3%	0.0%
application/*other*	107	2	440	43.9%	36.4%	12.2%
image/*other*	85	0	117	7.1%	14.5%	0.3%
other or unknown	170	4	1045	32.9%	30.4%	6.1%

Table 5-4: Summary of unchanged response bodies by content-type (packet-level trace)

	All status-200		All delta-eligible			Not including unchanged			All delta-eligible		
Content-type	Refs	MBytes	Refs	MBytes	Total time	Refs improved	Bytes saved	Time saved	Refs improved	Bytes saved	Time saved
All content-types	819998	6216	83991	645	195814	68.0%	56.2%	41.7%	99.6%	84.7%	60.6%
text/html	185636	1276	45812	400	130378	67.9%	61.7%	44.8%	100.0%	95.1%	67.1%
application/octet-stream	77531	819	17797	124	31767	99.9%	85.7%	63.6%	100.0%	85.8%	63.8%
image/gif	434476	2222	17024	88	26117	37.6%	5.4%	6.7%	98.0%	47.4%	29.5%
image/jpeg	106039	1513	2101	25	5397	47.5%	6.8%	6.1%	100.0%	49.9%	38.8%
text/plain	7023	67	493	2	459	77.7%	70.6%	23.6%	99.8%	91.6%	36.6%
application/x-msnwebqt	401	0	400	0	93	64.0%	57.5%	0.3%	100.0%	85.8%	0.3%
application/*other*	1301	116	107	2	440	48.6%	7.5%	8.6%	92.5%	43.9%	20.8%
image/*other*	2319	9	85	0	117	92.9%	76.0%	12.4%	100.0%	90.5%	12.7%
other or unknown	5002	105	170	4	1045	66.5%	40.2%	68.4%	99.4%	70.6%	74.5%

Table 5-5: Summary of savings by content-type, for *vdelta* (packet-level trace)

5.8. Effect of clustering query URLs

A significant fraction of the URLs seen in the proxy trace (42% of the URLs referenced) contained a ‘‘?’’ character, and so probably reflect a query operation (for example, a request for a stock quote). By convention, responses for such URLs are uncachable, since the response might change between references (HTTP/1.1, however, provides explicit means to mark such responses as cachable, if appropriate). In this trace, 23% of the status-200 responses were for query URLs. (There are fewer status-200 responses for query URLs than distinct query URLs in the trace, because many of these requests yield a status-302 response, a redirection to a different URL.)

Housel and Lindquist [8], in their paper on WebExpress, point out that in many cases, the individual responses to different queries with the same ‘‘URL prefix’’ (that is, the prefix of the URL before the ‘‘?’’ character) are often similar enough to make delta encoding effective. Since users frequently make numerous different queries using the same URL prefix, it might be much more effective to compute deltas between different queries for a given URL prefix, rather than simply between different queries using an identical URL. Banga et al. [2] make a similar observation. We will refer to this technique as ‘‘clustering’’ of different query URLs with a common prefix. (Such clustering is done implicitly for POST requests, since POST requests carry message bodies, and so the response to a POST may depend on input other than the URL.)

The WebExpress paper did not report on the frequency of such clustering in realistic traces. We found, for the proxy trace, that the 100780 distinct query URLs could be clustered using just 12004 prefix URLs. Further, of the 86191 status-200 responses for query URLs, only 28395 (33%) were delta-eligible if the entire URL was used, but 77314 (90%) were delta-eligible if only the prefix had to match.

Tables 5-7 and 5-8 show that clustering not only finds more cases where deltas are possible, but also provides significantly more reduction in bytes transferred and in response times. In fact, a comparison of tables 5-8 and 5-1 shows that when queries are clustered, delta encoding improves query response transfer efficiency more than it does for responses in general.

Content-type	Refs	MBytes	Total time	Refs improved	Bytes saved	Time saved
All status-200	819998	6216	2053775	72.8%	19.8%	14.2%
image/gif	434476	2222	823893	55.7%	4.6%	3.0%
text/html	185636	1276	516791	99.7%	68.8%	41.4%
image/jpeg	106039	1513	434467	99.1%	2.8%	2.6%
application/octet-stream	77531	819	196252	65.9%	10.4%	12.3%
text/plain	7023	67	20160	95.2%	55.7%	30.6%
image/*other*	2319	9	2173	98.6%	47.0%	25.3%
application/*other*	1301	116	19423	83.2%	35.2%	20.3%
application/x-msnwebqt	401	0	94	99.5%	56.3%	0.4%
video/*	225	88	13365	93.3%	12.6%	11.0%
text/*other*	45	0	66	100.0%	71.7%	38.8%
other or unknown	5002	105	27088	62.5%	26.2%	19.1%

Table 5-6: Summary of *gzip* compression savings by content-type (all status-200 responses in packet-level trace)

(We note, however, that because most query responses are generated on the fly, and are somewhat shorter on average than other responses, the query processing overhead at the server may dominate any savings in transfer time.)

Computation	Improved References		MBytes saved		Retrieval time saved	
unchanged	9169	(10.6%)	11	(2.9%)	1477	(1.0%)
diff -e	5052	(5.9%)	26	(6.7%)	3078	(2.2%)
diff -e | gzip	5241	(6.1%)	35	(8.9%)	5130	(3.6%)
vdelta	19197	(22.3%)	62	(15.6%)	12325	(8.7%)

N = 86191, 419 MBytes, 141076 seconds

Table 5-7: Improvements relative to all status-200 responses to queries (no clustering)

Comput.	Improved References		MBytes saved		Retrieval time saved	
unchanged	13905	(16.1%)	5	(1.3%)	1036	(0.7%)
diff -e	39063	(45.3%)	95	(24.0%)	9488	(6.7%)
diff -e|gzip	40664	(47.2%)	226	(56.7%)	17920	(12.7%)
vdelta	61858	(71.8%)	262	(65.7%)	25005	(17.7%)
diff -e ♣	52968	(61.5%)	101	(25.3%)	10525	(7.5%)
vdelta ♣	75763	(87.9%)	268	(67.1%)	26042	(18.5%)

♣: including unchanged responses

N = 86191, 419 MBytes, 141076 seconds

Table 5-8: Improvements when clustering queries (all status-200 responses to queries)

6. Including the cost of end-host processing

The time savings calculation described in section 5.4 omits any latency for creating and applying deltas, or for compressing and decompressing responses. Since these operations are not without cost, in this section we quantify the cost of these operations for several typical hardware platforms. We chose three systems: a 50 MHz 80486 (running BSD/OS, SPECint92 = 30), which would now be considered very slow; a 90 MHz Pentium (running Linux, SPECint95 = 2.88); and a 400 MHz AlphaStation 500 (running Digital UNIX V3.2G), SPECint95 = 12.3). The 90 MHz Pentium might be typical for a home user, and the 400 MHz AlphaStation is typical of a high-end workstation, but by no means the fastest one available.

Table 6-1 shows the results, which were computed from 10 trials on files (or, for deltas, pairs of instances) taken from the packet-level trace. For the delta experiments, we used 65 pairs of text files and 87 pairs of non-text files; for the compression experiments, we used 685 text files and 346 non-text files. The files were chosen to be representative of the entire set of responses. (We sorted the responses in order of size, and chose every *n*th entry to select 1% of the pairs, and 0.1% of the single-instance responses.) We express the results in terms of the throughput (in KBytes/sec) for each processing step, and for the sequential combination of the server-side and client-side processing steps. (Deltas created by *diff* are applied using the *ed* program; deltas and compressed output created by *vdelta* are fed to the *vupdate* program.) For deltas, the throughput is calculated based on the average size of the two input files.

We also show the standard deviations of these values. The deviations are large because there is a large fixed overhead for each operation that does not depend on the size of the input, and so throughputs for the larger files are much larger than the means. Much of this fixed overhead is the cost of starting a new process for each computation (which ranges from 15 to 34 msec. on the systems tested). However, since several of the delta and compression algorithms already exist as library func-

	50 Mhz 80486 BSD/OS 2.1				90 MHz Pentium Linux 2.0.0				400 MHz AlphaStn 500/DUNIX 3.2G			
	Text		Non-text		Text		Non-text		Text		Non-text	
Computation	**Mean**	**Std. dev.**	**Mean**	**Std. dev.**	**Mean**	**Std. dev.**	**Mean**	**Std. dev.**	**Mean**	**Std. dev.**	**Mean**	**Std. dev.**
diff -e	72.2	57.2	∅	∅	134.6	136.5	∅	∅	406.9	305.2	∅	∅
ed	90.1	64.4	∅	∅	101.5	94.3	∅	∅	1294.9	1420.4	∅	∅
both steps above	39.7	29.7	∅	∅	55.7	52.2	∅	∅	281.8	210.3	∅	∅
diff -e | gzip	33.6	29.3	∅	∅	90.7	83.3	∅	∅	153.4	135.9	∅	∅
gunzip | ed	14.8	14	∅	∅	37.9	33.6	∅	∅	472.3	407.9	∅	∅
both steps above	9.6	8.6	∅	∅	26.6	23.7	∅	∅	114.4	99.4	∅	∅
vdelta	63.4	46.4	89.4	60.8	183.1	160.1	193.4	133.2	149.6	331.9	188	226.9
vupdate	100.3	97.3	176.7	175.6	272	301.6	460.3	570.7	331.9	529	341.3	406.3
both steps above	37.7	29.2	55.9	40.2	106.4	102.2	122	90.1	100.7	133.6	117.1	136
gzip	72.9	43.4	106	78.4	100.5	78.6	106	78.4	252	151.6	189	139.4
gunzip	145.4	124.2	218.6	216.4	199.6	220.6	218.6	216.4	412.9	563.5	374.9	407.4
both steps above	47.6	31.2	70.1	56.8	64.2	54	70.1	56.8	147.9	103.2	121.8	100.3
vdelta (compress)	102.4	58.6	86.2	45.6	134.2	105.5	130.4	100.2	121.5	117.4	133.3	118.5
vupdate (decomp)	181.4	154.9	250.1	264.3	172.7	197.3	259.8	394.7	155.5	87.1	125.9	136.9
both steps above	63.8	40.9	60.7	38.5	73.1	64.6	79.9	74.2	66.2	46.7	63.3	60.7
vdelta *(library)*					924.8	781.6	1579.6	1660.6	2640.3	1879.5	3713.2	3460.6
vupdate *(library)*									5189.5	5325.3	7939.3	10647.5
both steps above									1606.1	1208.5	2245.6	2338.6

Values are in Kbytes/sec., based on elapsed times ∅: not applicable

Table 6-1: Overheads for compression and delta encoding

tions, an implementation could easily avoid this overhead[2]. The last three lines in table 6-1 show preliminary measurements of a library version of the *vdelta* and *vupdate* algorithms on two of the tested platforms. The results of these tests suggest that simply eliminating the use of a separate process reduces overheads by an order of magnitude. Although the Alpha's performance for the non-library versions of *vdelta* and *vupdate* are poor, relative to the much slower Pentium, the results for the library version of *vdelta* imply that the Alpha's poor performance on the non-library code is due to some aspect of the operating system, not the CPU.

We did not make an attempt to include these costs when calculating the potential net savings in section 5.5, because (1) we have no idea of the actual performance of the end systems represented in the trace, (2) some of the computation could be done in parallel with data transfer, since all of the algorithms operate on streams of bytes (3) it would not be always necessary to produce the delta-encoded or compressed response ''on-line''; these could be precomputed or cached at the server, and (4) historical trends in processor performance promise to quickly reduce these costs.

However, we make several observations. First, the throughputs for almost all of the computations (except, on the slowest machine, for ''gunzip | ed'') are faster than a Basic-rate ISDN line (128 Kbits/sec, or 16KBytes/sec), and the library implementations of *vdelta* and *vupdate* computations are significantly faster than the throughput of a T1 line (1.544 Mbits/sec, or 193 KBytes/sec.) This suggests that delta encoding and compression would certainly be useful for users of dialup lines (confirming [2]) and T1 lines, would probably be useful for sites with multiple hosts sharing one T3 line, and might not be useful over broadband networks (at current levels of CPU performance).

Second, computation speed often scales with CPU performance, but not always. For example, the cost of using *ed* to apply a delta appears to depend on factors other than CPU speed. Generally, *vdelta* seems to be the most time-efficient algorithm for both delta encoding and compression, except sometimes when compared against ''*diff -e*'' (which produces much larger deltas).

[2]The existing versions of the ''*diff -e*'' command generates output that is not entirely compatible with the *ed* command. *ed* requires one additional line in its input stream, which is normally generated by running another UNIX command. This adds significant overhead on some versions of UNIX, and since there is a simple, efficient fix for this problem, our measurements do not include the execution of this additional command.

Finally, the cost of applying a delta or decompressing a response is lower than the cost of creating the delta or compressed response (except for some uses of *ed*), for a given CPU. This is encouraging, because the more expensive response-creation step is also the step more amenable to caching or precomputation.

6.1. What about modem-based compression?

Many users now connect to the Internet via a modem; in fact, most of the slowest links, and hence the ones most likely to benefit from data compression, are modem-based. Modern modems perform some data compression of their own, which could reduce the benefit of end-to-end (HTTP-based) compression. However, we believe that a program which can see the entire input file, and which has available a moderate amount of RAM, should be able to compress HTML files more effectively than a modem can.

We conducted a simple experiment to test this, transferring both plain-text and compressed versions of several HTML files via FTP over both 10 MBit/sec Ethernet LAN and modem connections. URLs for these files are listed in table 6-2; our measurements used local copies of these URLs, made in January, 1997. Table 6-3 shows the measurements. The modems involved were communicating at 28,800 bps, and used the V.42bis compression algorithm (a form of the Lempel-Ziv-Welch algorithm; *gzip* uses the Lempel-Ziv algorithm). We used FTP instead of HTTP for a number of reasons, including the lack of caching or rendering in FTP clients; the retrieved files were written to disk at the client (a 75 MHz Intel 486 with Windows 95).

File	URL
A	http://www.w3.org/pub/WWW/Protocols/
B	http://www.w3.org/pub/WWW/
C	http://www.specbench.org/osg/cpu95/results/results.html
D	http://www.specbench.org/osg/cpu95/results/rint95.html

Table 6-2: URLs used in modem experiments

Table 6-3 shows that while the modem compression algorithms do work, and the use of high-level compression algorithms reduce the link-level bit rate, the overall transfer time for a given file is shorter with high-level compression than with modem compression. For example, the achieved transfer rate for file C using only modem compression was 55.3 Kbps (over a nominal 28.8 Kbps link), while the transfer rate for the vdelta-compressed version of the same file was only 16.3 Kbps. But, ignoring the costs of compression and decompression at the client and server, the overall transfer time for the file was 68% shorter when using high-level compression.

We found that although *vdelta* provided greater savings for large files (C and D), for the smaller files (A and B) the *gzip* algorithm apparently provides better results. It might be useful for an HTTP server generating compressed responses to choose the compression algorithm based on both the document size and the characteristics of the network path, although it could be difficult to discover if the path involves a compressing modem. In any case, using high-level compression seems almost always faster than relying on modem compression, particularly for large files.

When the costs of compression and decompression, shown in table 6-4, are included, the overall transfer time for the longer files (A, C, and D) is still much better using high-level compression. For the measurements in table 6-4, we used the slowest available system (the 50 MHz 80486 running BSD/OS); the results in table 6-1 imply that a more modern CPU would reduce these costs substantially.

7. Extending HTTP to support deltas

We have proposed a simple extension to HTTP to support the use of deltas [14], but space here permits only a brief description. We assume the use of HTTP/1.1 [6], which (while not yet widely deployed) provides better control over caching than does HTTP/1.0.

When an HTTP client wishes to check the validity of a cache entry, it sends a ‘‘conditional GET’’ to the server. This request indicates the identity of the cached response (using the ‘‘entity-tag’’ and ‘‘If-None-Match’’ features of HTTP/1.1). We extend this by adding an optional ‘‘Delta’’ header, so that the client may express to the server the set of delta-encoding algorithms it understands.

If resource has changed, and if the server supports at least one of the delta encodings known to the client, the server may choose to use a delta-encoded response. The server knows exactly which previous instance to base the delta on, since the client’s entity-tag specifies this unambiguously. A delta-encoded response is marked to indicate which encoding is used, and to prevent improper caching by shared proxies.

The use of deltas would slightly increase HTTP header sizes. Conditional request headers would be about 14 bytes longer, or 5% of the observed mean request size. (The client might choose to omit the ‘‘Delta’’ header on requests for images, thus avoiding this overhead.) The headers for delta-encoded responses would be slightly longer than for normal responses, but the increase would be much less than the decrease in response body size.

8. Future work

We have not been able to explore all aspects of delta encoding in this study. Here we discuss several issues that could be addressed using a trace-based analysis. Of course, the most important proof of the delta-encoding design would be to implement it and measure its utility in practice, but because many variations of the basic design are feasible, we may need additional trace-based studies to establish the most effective protocol design. (The previous studies [2, 8] did implementations, but using a double-proxy-based approach that adds store-and-forward delays.)

We also note that all of our analyses would benefit from a more accurate model for the transfer time, perhaps including a model of network congestion.

8.1. Delta algorithms for images

A significant fraction of the responses in our traces (or logged but not traced by the proxy), and an even larger fraction of the response body bytes, were of content-type ‘‘image/*’’ (i.e., GIF, JPEG, or other image formats). Delta-eligible image responses are relatively rare, but if these could be converted to small deltas, that would still save bandwidth. While *vdelta* appears capable of extracting deltas from some

	Size (bytes)			LAN transfer (secs)			Modem transfer (secs)			
File	**HTML**	**gzip**	**vdelta**	**HTML**	**gzip**	**vdelta**	**HTML**	**gzip**	**vdelta**	saved w/vdelta
A	17545	6177	7997	0.17 (0.05)	0.12 (0.04)	0.10 (0.00)	6.6 (0.52)	4.7 (0.58)	5.8 (0.54)	0.8 sec (13%)
B	6017	2033	2650	0.10 (0.00)	0.10 (0.00)	0.10 (0.00)	2.2 (0.20)	1.8 (0.30)	2.0 (0.42)	0.2 sec (12%)
C	374144	39200	35212	1.22 (0.04)	0.22 (0.04)	0.20 (0.00)	66.4 (0.17)	25.1 (3.07)	21.8 (2.95)	44.6 sec (67%)
D	97125	10223	8933	0.38 (0.04)	0.12 (0.04)	0.12 (0.04)	17.7 (0.12)	6.9 (1.04)	6.1 (0.99)	11.6 sec (66%)

Times are the mean of at least 7 trials; standard deviations shown in parentheses

Table 6-3: Effect of modem-based compression on transfer time

File	**gzip compr.**	**gunzip decomp.**	**gzip total**	**vdelta compr.**	**vdelta decomp.**	**vdelta total**
A	0.151	0.065	**0.216**	0.121	0.042	**0.163**
B	0.061	0.031	**0.092**	0.044	0.020	**0.064**
C	2.028	0.319	**2.347**	1.438	0.020	**1.458**
D	0.481	0.100	**0.581**	0.290	0.020	**0.310**

Table 6-4: Compression and decompression times (seconds) for files in tables 6-2 and 6-3

pairs of these image files, it performs much worse than it does on text files. We also have evidence that *vdelta* does poorly on images generated by cameras, such as the popular ‘‘Web-Cam’’ sites, many of which are updated at frequent intervals. MPEG compression of video streams relies on efficient deltas between frames, so we have some hopes for a practical image-delta algorithm.

8.2. Effect of cache size on effectiveness of deltas

Our trace analyses assumed that a client (or proxy) could use any previously traced response as the base instance for a delta. Although in many cases the two responses involved appear close together in the trace, in some cases the interval might be quite large. This implies that, in order to obtain the full benefits of delta encoding shown in our analyses, the client or proxy might have to retain many GBytes of cached responses. If so, this would clearly be infeasible for most clients.

It would be fairly simple to analyze the traces using a maximum time-window (e.g., 1 hour or 24 hours) rather than looking all the way back to the beginning of the trace when searching for a base instance. By plotting the average improvement as a function of the time-window length, one could see how this parameter affects performance. It might be somewhat harder to model the effect of a limited cache size.

8.3. Deltas between non-contiguous responses

Our analyses of delta-eligible responses looked only at the most recent status-200 response preceding the one for which a delta was computed. This policy simplifies the analysis, and would also simplify both the client and server implementations, since it limits the number of previous instances that must be stored at each end.

It is possible, however, that reductions in the delta sizes might be possible by computing deltas between the current instance and several previous instances, and then sending the shortest. The complexity and space and time overheads of this policy are significant, but the policy would not be hard to support in the protocol design. We could modify our trace analysis tools to evaluate the best-case savings of such policies.

8.4. Avoiding the cost of creating deltas

The response-time benefits of delta encoding are tempered by the costs of creating and applying deltas. However, as shown in section 6, the cost of creating a delta is usually much larger than the cost of applying it.

Fortunately, it may be possible to avoid or hide the cost of creating deltas, in many cases. Whenever a server receives several requests that would be answered with the same delta-encoded responses, it could avoid the computation cost of delta-creation by simply caching the delta. We could estimate the savings from this technique by counting the number of status-304 (Not Modified) and unchanged responses for a given URL, following a delta-eligible response for that URL in the trace. (The estimate would be conservative, unless the trace included the server’s entire reference stream.)

Even when a delta is used only once, it may be possible for the server to hide the cost of creating it by precomputing and caching the delta when the resource is actually changed, rather than waiting for a request to arrive. While this might substantially increase the CPU and disk load at the server (because it would probably result in the creation of many deltas that will never be used), it should reduce the latency seen by the client, especially when the original files are large. Many studies have shown that Web server loads are periodic and bursty at many time scales (e.g., [1]). If the server sometimes has background cycles to spare, why not spend them to precompute some deltas?

8.5. Decision procedures for using deltas or compression

While our results show that deltas and compression improve overall performance, for any given request the server’s decision to use delta encoding, compression, or simply to send the unmodified resource value may not be a trivial one. It would not make much sense for the server to spend a lot more time deciding which approach to use than it would take to transfer the unmodified value. The decision might depend on the size and type of the file, the network bandwidth to the client, perhaps the presence of a compressing modem on that path (see section 6.1), and perhaps the past history of the resource. We believe that a simple decision algorithm would be useful, but we do not yet know how it should work.

9. Summary and conclusions

Previous studies have described how delta encoding and compression could be useful. In this study, we quantified the utility based on traces of actual Web users. We found that, using the best known delta algorithm, for the proxy trace 83% of the delta-eligible response-body bytes and 31% of all response-body bytes could have been saved; at least 39% of the transfer time for delta-eligible responses and 12% of the total transfer time could have been avoided. For the packet-level trace, we showed even more savings for delta-eligible responses (85% of response-body bytes), although the overall improvement (9% of response-body bytes) was much less impressive. We confirmed that data compression can significantly reduce bytes transferred and transfer time, for some content-types. We showed that the added overheads for encoding and decoding are reasonable, and support for deltas would add minimal complexity to the HTTP protocol. We conclude that delta encoding should be used when possible, and compression should be used otherwise.

The goal for a well-designed distributed system should be to take maximal advantage of caches, and to transmit the minimum number of bits required by information theory, given acceptable processing costs. delta encoding and compression together will help meet these goals.

Acknowledgements

We would like to thank Kiem-Phong Vo for discussions relating to *vdelta*, Glenn Fowler for discussions regarding *diff -e* update problems, Guy Jacobson for discussions regarding *vdelta* compression, and Manolis Tsangaris and Steven Bellovin for discussions relating to modem compression. We would also like to thank Kathy Richardson and Jason Wold, for help in obtaining traces; Gaurav Banga and Deborah Wallach, for proofreading; and the SIGCOMM reviewers, for valuable suggestions.

References

[1] Martin F. Arlitt and Carey L. Williamson. *Web Server Workload Characterization: The Search for Invariants (Extended Version)*. DISCUS Working Paper 96-3, Dept. of Computer Science, University of Saskatchewan, March, 1996. ftp://ftp.cs.usask.ca/pub/discus/paper.96-3.ps.Z.

[2] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proc. 1997 USENIX Technical Conf.*, pp. 289-303. Anaheim, CA, January, 1997.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol -- HTTP/1.0*. RFC 1945, May, 1996.

[4] F. Douglis. On the Role of Compression in Distributed Systems. In *Proc. Fifth ACM SIGOPS European Workshop*. Mont St.-Michel, France, September, 1992. Also appears in *Op. Sys. Review* 27(2):88-93, April, 1993.

[5] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of Change and Other Metrics: A Live Study of the World Wide Web. 1997. Submitted for publication.

[6] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2068, January, 1997.

[7] A. Fox, S. Gribble, E. Brewer, and E. Amir. Adapting to Network and Client Variation via On-Demand Dynamic Transcoding. In *Proc. ASPLOS VII*, pp. 160-170. Cambridge, MA, October, 1996.

[8] B. Housel and D. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. In *Proc. 2nd Annual Intl. Conf. on Mobile Computing and Networking*, pp. 108-116. Rye, New York, November, 1996.

[9] J. Hunt, K. P. Vo, and W. Tichy. An Empirical Study of Delta Algorithms. In *IEEE Soft. Config. and Maint. Workshop*. Berlin, March, 1996.

[10] V. Jacobson. *Compressing TCP/IP Headers for Low-Speed Serial Links*. RFC 1144, February, 1990.

[11] G. Fowler, D. Korn, S. North, H. Rao, and K. P. Vo. Libraries and File System Architecture. In B. Krishnamurthy (editor), *Practical Reusable UNIX Software*, chapter 2. John Wiley & Sons, New York, 1995.

[12] D. Lelewer and D. Hirschberg. Data Compression. *ACM Computing Surveys* 19(3):261-296, 1987.

[13] S. McCanne and V. Jacobson. An Efficient, Extensible, and Portable Network Monitor. Work in progress.

[14] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. *Potential benefits of delta encoding and data compression for HTTP*. Research Report 97/4, Digital Equip. Corp. Western Research Lab., July, 1997.

[15] W. Tichy. RCS - A System For Version Control. *Software - Practice and Exp.* 15(7):637-654, July, 1985.

[16] S. Williams. Personal communication. June, 1996. http://ei.cs.vt.edu/~williams/DIFF/prelim.html.

[17] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. SIGCOMM '96*, pp. 293-305. Stanford, CA, August, 1996.