NOVEMBER 1999

WRL
Research Report 99/2

A trace-based analysis of duplicate suppression in HTTP

Jeffrey C. Mogul

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL), located in Palo Alto, California, is part of Compaq's Corporate Research group. WRL was founded by Digital Equipment Corporation in 1982. Our focus is information technology that is relevant to the technical strategy of the Corporation, and that has the potential to open new business opportunities. Research at WRL includes Internet protocol design and implementation, tools for binary optimization, hardware and software mechanisms to support scalable shared memory, graphics VLSI ICs, hand-held computing, and more. Our tradition at WRL is to test our ideas by extensive software or hardware prototyping.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

You can retrieve research reports and technical notes via the World Wide Web at:

http://www.research.digital.com/wrl/home

You can request printed copies of research reports and technical notes, when available, by mailing your order to us at:

Technical Report Distribution
Compaq Western Research Laboratory
250 University Avenue
Palo Alto, California 94301 USA

You can also request reports and notes via electronic mail. For detailed instructions, put the word ''Help'' in the Subject line of your message, and mail it to:

`WRL-Techreports@pa.dec.com`

A trace-based analysis of duplicate suppression in HTTP

Jeffrey C. Mogul

Compaq Computer Corporation Western Research Laboratory
mogul@pa.dec.com

November, 1999

Abstract

Many HTTP resources (pages, graphics, etc.) are exact duplicates of other resources with different URLs. If an HTTP cache contains a duplicate of a requested resource, and could detect this, it could avoid substantial network costs by returning the cached duplicate in place of the requested URL. Previous studies have shown that there is substantial duplication of content in both HTTP and FTP, and several protocols have been proposed to support efficient and safe *duplicate suppression* in HTTP. We use traces covering millions of HTTP requests to quantify the potential benefit of an HTTP duplicate-suppression extension. In particular, we show that the benefits vary depending on content-type, and that a small fraction of Web servers account for most of the duplicated resources.

Copyright © 1999, Compaq Computer Corporation

COMPAQ **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Table of Contents

1. Introduction 1
2. Motivation and context 2
- **2.1. Squeezing more utility out of caching 3**
- **2.2. Automatic duplicate suppression 4**
- **2.3. Questions addressed by this study 4**

3. Prior and related work 5
- **3.1. Proposed duplicate suppression protocol 6**
- **3.2. Duplicate suppression and cookies 8**

4. Trace collection 8
5. Trace analysis 10
- **5.1. Cache simulation 11**

6. Results 12
- **6.1. Overall trace statistics 13**
- **6.2. Frequency of duplication 14**
- **6.3. Contexts where duplication is most likely 17**
- **6.4. Effect of duplicate suppression on latency 22**
- **6.5. Implications for cache design 22**
- **6.6. Protocol and computation overheads 25**

7. Security considerations 27
8. Future work 28
9. Summary and Conclusions 28
Acknowledgments 28
References 28

List of Figures

Figure 6-1: Daily statistics (23-day trace) 13
Figure 6-2: Daily statistics (90-day trace) 14
Figure 6-3: Daily ratios (23-day trace) 15
Figure 6-4: Daily ratios (90-day trace) 16
Figure 6-5: Daily ratios, weighted by response size in bytes (23-day trace) 16
Figure 6-6: Daily ratios, weighted by response size in bytes (90-day trace) 16
Figure 6-7: Static frequency of replication 19
Figure 6-8: Distribution of response lengths 20
Figure 6-9: Duplication frequency by host 21
Figure 6-10: Cache entry age distributions (23-day trace) 23
Figure 6-11: Cache entry age distributions (90-day trace) 24
Figure 6-12: Byte-weighted cache entry age distributions (23-day trace) 24
Figure 6-13: Byte-weighted cache entry age distributions (90-day trace) 24

List of Tables

Table 6-1: Distribution of response status codes 13
Table 6-2: Linear regressions for figures 6-4 and 6-6 17
Table 6-3: Overall duplication statistics, by scheme 17
Table 6-4: Duplication statistics for 10 most frequent Content-types 18
Table 6-5: Duplication statistics, 10 Content-types with highest DR 18
Table 6-6: Duplication statistics, 11 Content-types with highest WDR 19
Table 6-7: Effects of mirroring and aliasing 21
Table 6-8: Projected transfer time savings (as fraction of actual time) 23
Table 6-9: Medians and 90th percentiles for age distributions 25

1. Introduction

HTTP accounts for most of the bytes flowing over the Internet backbone (up to 75%, in one study [45]). This bandwidth demand requires continued investment in link and switch capacity, and leads to congestion, which increases user-perceived latency. At the edges of the Internet, which are often bandwidth-constrained, every extra byte transferred adds incremental delay; this is a particular problem for home users, who do not yet have a cost-effective means to increase bandwidth above 56Kbits/sec.

Many mechanisms have therefore been implemented or proposed to reduce Web-related bandwidth requirements. Above the HTTP level, for example, some designers have learned to use simpler graphical elements, but the desire for richer user experiences usually prevails over such pragmatism. Some new content formats, such as Cascading Style Sheets [26] and HTML macros [10], can increase the coding efficiency without reducing expressiveness, but other emerging formats, such as MPEG-1 Layer 3 (MP3) audio [22], provide new bandwidth challenges. Thus, pressure remains to provide protocol-level mechanisms to conserve bandwidth.

At the HTTP level, such mechanisms include caching, cooperative caching, compression, and support for partial transfers and differential cache updates (also known as delta encoding). These all work on a scope of one URL at a time; for example, a Web cache hit results from a repeated reference to a given URL.

Much of the Web's content is duplicated: that is, the same content appears at more than one URL. Duplication might be intentional (as with a mirror site) or casual (as with a logo or background image). Some duplicates are only approximate; for example, at a mirror site, where the displayed content is identical but some of the link targets have been adjusted [7]. Other duplicates are exact byte-for-byte copies. An earlier study showed that 18% of the data-carrying responses in a trace were exact duplicates of a response for some other URL [9].

If it were possible to automatically eliminate the Internet transfer of all such duplicates by an efficient protocol-level mechanism, this would measurably reduce bandwidth requirements, and would often improve user-perceived latencies. Several proposals have been made for HTTP mechanisms to support duplicate suppression. However, we know of no careful attempt to quantify the benefit of these mechanisms. Common sense also suggests that the benefits differ according to context, yet there have been no prior attempts to identify the most important contexts.

In this paper, we use results from lengthy traces of a moderately large user community to evaluate the performance of a proposed HTTP duplicate-suppression mechanism. We limit our study to the suppression of exact (byte-for-byte) duplicates. Our trace analyses show that, with a sufficiently large proxy cache, duplicate suppression could eliminate 5.4% of the responses (or 10.8% of the cache misses), and 6.2% of the Internet bytes transferred (8.9% of the cache-miss bytes). We also found that some hosts experience duplication far more often than others, and that the benefits vary significantly based on content-type.

2. Motivation and context

There are three basic techniques to avoid sending bytes over a network: content simplification, data compression, and caching.

Content simplification works, to a point. For example, Web designers can use common sense to reduce page complexity, or special software tools to optimize image coding (e.g., [12]). Transcoding proxies can reduce image complexity for transmission over slow links [17]. But some content, such as medical images, broadcast-quality video, and executable software, cannot be simplified without loss of meaning.

Data compression directly targets the transmission of redundant bits within a single transfer. Existing general-purpose compression algorithms provide significant size reductions, typically reducing text file sizes by a factor of three or more. However, the bulk of Web-related bytes transferred come from content-types, such as images, video, and audio, which are already heavily compressed using highly efficient content-specific compression algorithms, and so the net benefit of applying general-purpose compression to all Web traffic would be relatively small [33]. The trend towards increased use of non-text media further reduces the potential for general-purpose compression in HTTP.

With the increased transmission of new data types (such as Java byte codes and other software), we expect to see progress made on new type-specific compression techniques. For example, general-purpose algorithms can be tuned for improved compression of binaries [50], and special-purpose compression algorithms based on parse trees can do even better [13]. Even so, compression has its limits, and data types such as program binaries inevitably gain complexity with time.

Caching, the third technique for bandwidth conservation, has been successfully applied in many contexts, including the Web. Most browsers include a cache, and many sites use Web proxy caches to take advantage of the locality in a larger reference stream. Most studies of Web proxy caches report the hit ratio (HR) per resource and the *weighted hit ratio* (WHR), weighted by the size of the response body; the bandwidth reduction ratio should be similar to the WHR, although the WHR does not account for HTTP and other protocol header overheads[1]. Studies have reported WHRs ranging from 14% to 36% (assuming an infinite cache size) [11, 14]; the variations may be due to differences in user community, geography, or when the traces were obtained. Reports generated from the NLANR caches [36] show actual WHRs vary tremendously over short timescales.

It seems impossible to significantly increase hit ratios and weighted hit ratios above a certain (if fuzzy) threshold, probably because of several intrinsic aspects of Web reference streams:

Uncachable resources: Some responses cannot be cached; for example, stock quotes, query results, or electronic commerce ‘‘shopping baskets.’’ Other responses could be provided from a cache, save for the serving site’s desire to gather demographic information or ad-

[1] A digression on terminology: some people prefer the term ‘‘document hit ratio’’ instead of ‘‘cache hit ratio’’ (although many Web resources are not really documents!). Also, the terms ‘‘byte hit ratio’’ and ‘‘weighted hit ratio’’ are used interchangeably.

vertising revenue. Many potentially-cachable resources actually change fairly rapidly [9], which would lead to cache incoherence if response for these resources were allowed to be cached.

Zipf's law: The Web is so large that many pages will never be referenced more than once in the reference stream seen by any one cache. Studies have shown that page re-reference frequencies (seen from the point of view of a proxy) follow a distribution similar to Zipf's law, in which the relative probability for a reference to the *k*th most popular page is proportional to *1/k* [6, 37]. This implies that in a large universe of pages, most of these pages are extremely unlikely to be referenced twice in the same reference stream. Most cache hits come from a small set of resources, but many references are made to resources outside this set.

Resource size distribution: Why is the weighted hit ratio almost always lower than the simple hit ratio? Williams et al. [48] observed that most references in their traces were for small resources. Breslau et al. [6], working from several trace sets, showed that there is no strong correlation between resource size and access frequency, although the mean size of popular resources is smaller than the mean for unpopular ones. One possible explanation for these observations is that a small set of small resources accounts for most of the cache hits.

2.1. Squeezing more utility out of caching

Given the apparent limits on the performance of simple caches, researchers and vendors have developed several techniques to extend the utility of Web caches. If the basic principle of simple ‘‘re-use’’ caching is to exploit repeated references to entire cached responses, the basic principle of these extended mechanisms is to exploit partial information present in caches.

Such mechanisms include:

Prefetching: If a cache can predict a user's future references, and if there is spare bandwidth, the cache can prefetch the expected resources. When the prediction is correct and timely, this can increase the WHR and reduce user-perceived latency [4, 25, 27, 39], but inevitably increases bandwidth requirements (because of false prefetches). Prefetching thus forces a cache operator to choose between improving latency and minimizing bandwidth requirements.

Partial transfers: HTTP transfers can terminate in mid-stream due to network errors, users clicking the ‘‘stop’’ button, or when users click on a link before the entire page is loaded. In HTTP/1.0, the result of a partial transfer is not worth caching, but in HTTP/1.1, a cache can fill in the missing data using a ‘‘range retrieval request’’ [16]. This creates utility for partial cache entries. (Unfortunately, we know of no published statistics for the prevalence of partial transfers.)

Delta encoding: Douglis et al. [9] analyzed traces to find the rate at which Web resources change. They showed that the distribution of time-since-last-modification generally has a broad peak on the time scale of months, many frequently-referenced resources change much more rapidly: on timescales of hours or days. This means that responses for such resources cannot be cached for very long. However, the changes are often quite small, which motivates the use of *delta encoding* [2, 20, 21, 33], in which the server sends just the difference (or *delta*) between the cached version and the current version of a resource.

2.2. Automatic duplicate suppression

The techniques for extending cache utility, discussed in section 2.1, do not exploit one possible mechanism for exploiting existing cache entries: if two distinct resources would generate exactly identical responses, then a cache entry for one of them could be used to provide a cache hit for a request referencing the other. This technique is called *duplicate suppression*, and in principle could avoid a lot of data transfer and the related latency.

In practice, its utility depends on:

1. A simple and efficient way to detect exact duplication.
2. A simple and efficient HTTP protocol extension to carry the necessary meta-information.
3. A sufficient rate of duplication to justify deploying the protocol extension.

The first requirement is met by the use of a digest (or checksum) algorithm, such as MD5 [42], for which it is difficult to generate identical digest values (‘‘collisions’’) for two different inputs.

Several protocols have been proposed to address the second requirement. The first such design, the ‘‘Distribution and Replication Protocol’’ (DRP) [19], proposed creating a special Universal Resource Name (URN) out of the MD5 or SHA [35] digest for a resource. A later refinement of this proposal retains the traditional HTTP URL mechanism for naming resources, and transmits the digest in a new HTTP header field [32]. We outline the protocol mechanism in section 3.1.

2.3. Questions addressed by this study

The study presented in this paper addresses the third requirement: is the rate of duplication actually sufficient to justify the use of a duplicate-suppression mechanism? In particular, we attempt to answer the following questions, with respect to a representative HTTP reference stream:

- How frequently are duplicates seen?
    - What fraction of responses are duplicates? What fraction of response bytes are in the duplicates? What fraction of transfer time is spent on duplicates?
- In what contexts are duplicates most likely?
    - Does the duplication rate depend on content type? Does the duplication rate depend on response size? Are some servers more likely to generate duplicates? Are some URLs, even if their content changes rapidly, more prone to duplication than others?
- What are the implications for cache design?
    - How large a cache is required? How long should a potential duplicate be retained?
- What are the overheads?
    - How many extra protocol-header bytes are sent? How much time is spent computing checksums?

We also address several more specific issues, including

- The connection between advertising banners and duplication.
- The connection between mirror sites, DNS nicknames, and duplication.

In another publication [30], we report on:

- Whether caching based on HTTP's `Last-Modified` header leads to accidental incoherence.

3. Prior and related work

Even before the Web had become visible within the Internet research community, Danzig et al. [8] had identified the problem of duplication among FTP sites, and suggested the use of server-independent naming to avoid unnecessary transfers of duplicates. They apparently were the first to propose a form of Internet proxy caching. They even used a simple (but not collision-free) signature scheme to analyze their traces of FTP activity, although they apparently failed to realize that a signature might be used as a server-independent name. They also failed to report on the frequency at which duplicate files were transferred using different filenames or hostnames.

Santos and Weatherall propose and analyze a link-level duplicate-suppression mechanism quite similar, in some ways, to the HTTP-level mechanism [44]. In their approach, a ‘‘compressor’’ system is used on the sending side of a link, and a ‘‘decompressor’’ is used on the receiving side. (In practice, these systems might be integrated into the routers, and would be duplicated for a bidirectional link.) The compressor stores each packet's payload in its cache, and computes a payload digest (using, e.g., MD5). If the identical payload was already in the cache, the compressor sends a special ‘‘compressed’’ packet containing just the digest, rather than the original payload. The decompressor therefore receives the first copy of a given payload in uncompressed form, and subsequent copies in compressed form, and uses its own cache to map from digest values back to payloads. This algorithm requires that both compressor and decompressor use identical caching algorithms, and also includes mechanisms for resynchronizing the caches if a packet be lost or one end is restarted. Santos and Weatherall report bandwidth savings of about 20%, with relatively little overhead. They also report an HTTP-specific duplication rate of about 26%, based on trace analysis.

The link-level approach has several advantages over an application-level approach. It applies to all applications using a given link, without any need to modify application code, and runs no risk of affecting end-to-end semantics or cache coherence. However, it also has several disadvantages. Running at link-level requires implementing the algorithm, and the associated cache storage, at every link in the path where bandwidth is limited, and Internet paths tend to have many hops[2]. Core Internet routers probably do not have the capacity to checksum or cache every packet, so the link-level approach probably is not applicable to core links. An end-to-end solution avoids these two problems.

[2] Paxson reports that ‘‘the operational diameter of the Internet has grown beyond 30 hops’’ [40], although we are unaware of any published study of the dynamic mean.

The link-level approach has two other subtle problems specific to HTTP. Although it avoids most bandwidth-related delays when replication is detected, it still requires end-to-end communication between client and server, which can add latency due to round-trip time and server loading. Doing duplicate suppression in an HTTP-level cache close to the client, on the other hand, avoids all of these delays. The final problem is that the link-level approach requires exact duplicate packet payloads, but the support for persistent connections and pipelining in HTTP/1.1 provides a strong incentive for packing of multiple responses into one packet, and without regard for packet boundaries [38]. This may lead to unique packet payloads even with lots of HTTP-level duplication, although it is also possible that HTML-level page compositions will be static enough that substantial packet-level duplication persists.

Broder et al. [7] looked for syntactically similar documents in the Web, rather than byte-for-byte identical ones. They use a careful and quantifiable definition of similarity, although they do not attempt to detect true semantic differences (such as changing ‘‘shall’’ to ‘‘shall not’’ in a lengthy document). Using a static sample of 30 million HTML and text documents taken from the Web, they found 3.6 million ‘‘clusters’’ (12.3 million documents) of similar documents, including 2.1 million clusters (5.3 million documents) where each cluster contained only identical documents (after some canonicalizations). Thus, in their static sample, about 18% of the documents were exact or near-exact duplicates, and about 41% were ‘‘similar’’ by their measure. These results are consistent with the dynamic HTTP-response duplicate rate of 18% found by Douglis et al. [9] and the dynamic HTTP packet-payload duplication rate of 26% reported by Santos and Weatherall, although it is important to remember that each of these studies is looking at distinctly different quantities.

The HTTP-level duplicate-suppression mechanism could, in principle, be used to substitute ‘‘sufficiently similar’’ responses instead of byte-for-byte identical ones, and so the high static rate of similar documents reported by Broder et al., if representative of the dynamic rate, is encouraging. However, we have made no attempt to define ‘‘sufficiently similar’’ and so we cannot evaluate the potential performance of this extension.

The DRP proposal [19] was apparently intended, in large part, to support binary software distribution, and its use of digest values as ‘‘content identifiers’’ allows the use of an appropriate component independent of its (network) source. Miller and Akala describe the use of ‘‘content-derived names,’’ also based on digest values, in a software package management context, but do not discuss the potential for bandwidth savings [29].

3.1. Proposed duplicate suppression protocol

Although our goal is to quantify the potential benefit of HTTP duplicate suppression, not to critique the possible protocol designs, we describe a simplified protocol, in order to make the rest of the paper more concrete. Complete specifications for several complete protocols are available [19, 34].

Although users occasionally load Web pages by typing a URL, in most cases an HTTP transfer is initiated when the browser software follows a link: either explicitly, when the user clicks on an anchor, or implicitly, via an embedded image or a client script. Except in the relatively infrequent case where a link leads to another server, the source of the linkage information is also the source of the linked-to resource.

Therefore, the same server often controls both the link information and its target. This allows the server to provide meta-information about the link target as part of the linkage information. (For example, HTML supports the HEIGHT and WIDTH attributes of an IMG tag, allowing the browser to reserve screen space for an image before actually loading it.) To support duplicate suppression, the server could include in this meta-information an MD5 (or similar) digest of the link target.

Because MD5 provides 128-bit digests, the probability of two randomly chosen, different objects having the same digest is approximately 2^{128}, or about 10^{38}. However, it is unclear if MD5 is truly immune to malicious subversion; see section 7 for more discussion.

As an alternative (or complement) to the possibility of a new HTML attribute, the digest value could be transmitted in a structured type: DRP introduces a new ‘‘index’’ content-type to provide meta-information for a consistent set of link targets [19]; similarly, WEBDAV [18] specifies a similar ‘‘collection’’ resource, whose state consists of a list of member URLs and an extensible set of properties.

Assume therefore that, by some external mechanism, an HTTP client is about to make a request for a URL *U*, and the client already knows the MD5 digest value *D* for the proper response. The client can therefore check its cache not only for an existing entry for *U*; it can also check its cache for an existing entry with a digest value of *D*. Either cache entry should therefore be a satisfactory substitute for getting a response from the actual server. (We defer, for a moment, the problem of cache timeliness and certain other details.)

If the client’s local cache does not contain the target, it might send its request via a proxy cache that does. This request could be of the form ‘‘please send me either a response for URL *U*, or a response with MD5 digest value *D*’’; if the proxy cache has a response cached under either key, it can return the cache hit rather than forwarding the request to the server. Thus, once the client knows the proper MD5 digest value, it can use both its own cache and a proxy’s cache to find a duplicate with the same digest, rather than waiting for a response from the actual server.

HTTP caches can only avoid contacting the server if the cache entry in question hasn’t expired. HTTP servers can attach expiration information to responses (e.g., using the ‘‘Expires’’ header), and so should probably also indicate, along with the MD5 digest of a link’s target, the time at which that digest might no longer be valid. This should make the cached information returned using duplicate suppression no less timely than that provided by standard HTTP caching.

Note that this protocol does not require the origin server to actually transmit a response digest with each response. In principle, these digests could be recomputed at the caches, although in practice it might be more efficient for the server to transmit the value (since the server might already have computed the digest when transmitting the meta-information). The protocol also does not require the transmission of an updated digest value with a status 304 (‘‘Not Modified’’) response, since such a response by definition implies that the cached digest value has not changed.

The complete specification of a duplicate suppression protocol would require attention to a number of other issues, such as whether HTTP header information for a cached response (such as authentication information) can properly be associated with a duplicate-suppression response for a different resource. For the purposes of evaluating potential benefit, we ignore these details, although in section 7 we discuss some aspects of security.

3.2. Duplicate suppression and cookies

Many Web applications rely on a browser mechanism that allows the server to store a small amount of state at the client. This state is known as a *cookie*. In the cookie protocol [24], an HTTP server supplies a ‘‘Set-Cookie’’ header with a response. The client browser then stores the cookie value in a database keyed by the server host name, and returns the cookie on certain subsequent requests to the same host, using a ‘‘Cookie’’ header.

The value supplied in a Set-Cookie response header is normally meant only for a single user, and the response to a request containing a Cookie request header might depend on the user-specific Cookie value. Therefore, caching of responses with cookie-related headers (in either the request or the response) can lead to semantic errors, including the accidental release of personal information. The cookie specification explains how some of these issues can be resolved without disabling caching, but many proxies simply refuse to cache cookied-related responses rather than following complex (and possibly fallible) caching rules.

One study found that cookies are one of the more prevalent causes for uncachable responses: Feldmann et al. [15] report on two traces where 19% and 30% of the responses were uncachable due to cookies. However, another study by Wolman et al. [49] reported only 4.4% of responses as being uncachable due to cookies, and of those, only a tiny fraction would otherwise have been cachable. It therefore remains unclear exactly how significantly the use of cookies decreases cache performance.

Duplicate suppression might provide a way to increase the cachability of cookie-related responses. We can start by observing that if a response is duplicated, then it probably is not client-specific. But the cookie mechanism can give rise to ‘‘duplicate’’ responses (in this case, sharing a URL but in response to different Cookie request headers). Remember that the client sends a Cookie header on subsequent requests to a server that send a Set-Cookie response header. So a client might send a Cookie header for a request on resource `http:/example.com/logo.gif`, even if `logo.gif` itself has no client-specific value.

If the origin server supplies the client with a digest for a particular resource, this implies that server is willing to let the client use any response with that digest in place of the requested resource. So, duplicate suppression bypasses policies that restrict the use of cached responses for requests with Cookier headers. This is another example where the use of digest-based naming helps ensure correct semantics without requiring duplicated transmission of data.

4. Trace collection

We obtained our traces at the Palo Alto, California proxy of Compaq Computer Corporation, one of several firewall proxies serving the company. The proxy is used for access control, not for performance, and so is not set up as a cache.

The proxy runs version 1.1.20 of the Squid proxy software [47]. We modified Squid to compute an MD5 digest for the body of each response, and to log these digest values in one of the log files Squid already keeps (`squid.store.log`). We also augmented this log format to include the connection duration, in milliseconds, as measured by the proxy. These changes were relatively simple, but Squid is a complex program and there may be a few error conditions in which we could log the wrong digest value.

Each `squid.store.log` entry also includes a timestamp (with roughly millisecond resolution), status information, the length of the response body, and the values of selected HTTP response header fields: ‘‘Date’’, ‘‘Last-Modified’’, ‘‘Expires’’, and ‘‘Content-Type’’. Unfortunately, it does not include the value, if any, of the ‘‘If-Modified-Since’’ request header; this makes it impossible to completely model the behavior of a caching proxy.

The `squid.store.log` also does not include any kind of client identification. We therefore cannot use these logs to study the potential for duplicate suppression in browser caches (which would also require us to accurately model the contents of these caches, a near-impossible task when we do not see the local references from these clients.) Browser caches are usually much smaller than proxy caches, so one might expect the potential for browser-local duplicate suppression to be relatively small, but we will avoid further speculation on this point.

The log entries contain complete URLs with server host names (such as `http://www.compaq.com/`), rather than server IP addresses. A given name might resolve to several IP addresses, allowing transparent server replication; we assume that any site with this feature is internally consistent.

In some cases, several names resolve to a single IP address; for example, `www.compaq.com` and `compaq.com` might actually be identical. This can give rise to apparent duplicates, since two URLs leading to the same server file will appear to be different. From the point of view of an HTTP cache, they are different URLs: HTTP/1.1 specifies the use of the name in the Host header, rather than the server's IP address, to determine server identity. Therefore, we assume that no proper HTTP cache would satisfy a request for `compaq.com/home.html` with a cache entry for `www.compaq.com/home.html`.

The average log entry consumes about 169 bytes, or about 44 bytes after compression with gzip. It is thus feasible to store many millions of log entries on a moderately large disk.

We collected a continuous trace covering 23 days from 17 October 1998 to 11 November 1998, inclusive. The trace includes 29,390,845 log entries, and occupies 1.2 GBytes in compressed form. The busiest day during the trace accounts for 1,936,315 log entries.

For certain of our analyses, we also used a longer trace, covering 90 days from 1 January 1999 through 31 March 1999. This trace includes 125,259,641 log entries, and occupies 5.1 GBytes in compressed form. The busiest day during the trace accounts for 2,085,909 log entries.

5. Trace analysis

We wrote a program to analyze a trace. It starts by parsing the extended `squid.store.log` format, skipping over unusable log entries. These include malformed entries, aborted transfers, all HTTP methods other than ‘‘GET’’, and all HTTP response status codes other than 200 (the normal success code, where the response carries the entire body of the resource value).

The program also skips a relatively small set of responses which match both the URL and MD5 digest of a previous response, but which have different content-type or content-length values (according to the log entries). Since it should not be possible to generate the same MD5 digest if the length varies, we believe that these ‘‘impossible’’ values represent failed transfers that are not indicated as such in the log. A review of the Squid sources reveals several possible places where a transfer could be aborted without being logged as such (this is our fault, not a bug in Squid). Some of the content-type mismatches may reflect a changed content-type assignment at the server, perhaps because of a misconfiguration, but we also try to skip these to avoid confusing the results.

From the 29,390,845 log entries in our 23-day trace, this winnowing process yielded 18,802,027 entries (about 64%) usable for our analysis. For the 90-day trace, winnowing yielded 79,441,708 usable entries (about 63%).

The analysis program uses each entry to create one or more nodes in an *ad hoc* database. For example, a record for each unique MD5 digest D_i is stored in a hash table. This record serves as the head of a list of nodes each representing a tuple (D_i, URL_j), where URL_j is a distinct URL with at least one response matching digest D_i. Each such node contains back-pointers to a nodes describing URL_j in more detail, and itself serves as the head of a list of nodes with per-entry information (such as elapsed time). If the list associated with D_i contains at least two elements, this implies that the trace contains a duplicated response with that digest value.

Once the database has been created, the linkages between the various nodes allow the analysis program to traverse the database in various orders, collecting statistical information about the responses described in the trace. For example, the program can iterate over a list of all known servers, and thereby discover which servers are most susceptible to duplication.

The analysis program keeps its database in main memory. A database with tens of millions of entries requires a significant amount of RAM to avoid excessive paging; this program cannot feasibly analyze arbitrarily long traces. While the memory requirements vary somewhat with the analysis requested, because of the creation of auxiliary data structures, we found that analysis of our 23-day trace required about 2080 MBytes of program memory. This is not simply a problem for simulation, since a practical implementation of duplicate suppression would require a cache to keep the necessary index structures in main memory. Fortunately, this probably represents a generous upper bound on the incremental memory use of duplicate suppression, since roughly half of that 2080 MBytes is used for storing information that is only useful for simulation (such as event timestamps), and another quarter is already needed for caching (such as URLs). In practice, support for duplicate suppression would probably require about 24 to 32 additional main-memory bytes per cache entry, for MD5 digests and several pointers.

Analysis of the 90-day trace required over 7400 MBytes of program memory. Because we had no systems available with more than 8 GB of RAM, this set a practical upper limit on the feasible trace length.

We note one important shortcoming of our analysis. In actual use, the duplicate suppression mechanism described in section 3.1 requires that client has received meta-information (e.g., a digest value) from the server *S* for a URL *U* before it can generate a request for *U* that might be duplicate-suppressed by a proxy cache. That is, the client must have made at least one prior reference to a resource on *S*[3] before the reference to *U*. Since our logs do not contain client identities, we cannot simulate this step, and so we assume that the client might learn the meta-information by magic, if necessary. This assumption is acceptable, because our intent is to find the upper bound on the performance of duplicate suppression.

5.1. Cache simulation

Although we obtained our trace at a non-caching proxy, our analysis program attempts to simulate the behavior of a caching proxy, since duplicate suppression implies the use of a cache. We must also avoid crediting the duplicate suppression algorithm with ‘‘hits’’ that would have been provided anyway by a normal cache. However, we use an atypical approach to HTTP cache simulation.

A typical HTTP cache uses two mechanisms to increase the likelihood that it will provide accurate responses:

1. **Expiration times**: If the server provides an ‘‘Expires’’ header, the cache assumes that the response can be used until that deadline, without refreshing it from the server. Otherwise, some caches estimate an expiration time, based on the ‘‘Last-Modified’’ time and other parameters.
2. **Revalidation**: If a cache entry has expired, the cache checks with the server to see if the entry is still valid. It does this check by sending a GET request with an ‘‘If-Modified-Since’’ header including the ‘‘Last-Modified’’ date from the cached response; if the resource is unmodified, the server responds with a status code of 304 (‘‘Not modified’’).

At a proxy cache, which acts as both client and server, the picture is more complicated. For example, a proxy that receives a GET request with an ‘‘If-Modified-Since’’ header might find in its cache a matching and unexpired entry (and should return a 304 response), or a more recent unexpired entry (and should return a 200 response), or it might have to forward the request towards the ‘‘origin server’’ (the master server for the resource).

The HTTP caching mechanisms do not guarantee that caches provide accurate (that is, cache-coherent) responses, for several reasons: The expiration time might be too optimistic, or the Last-Modified timestamp might be wrong (perhaps due to clock skew), or the source might be modified twice during one second (a condition not detectable with the one-second resolution of the ‘‘Last-Modified’’ header). On the other hand, the use of timestamps to check validity eliminates certain possibilities for cache hits, especially for dynamic resources. HTTP/1.1 intro-

[3] Or to another server *S’* that speaks for *S*.

duces a new ‘‘entity tag’’ mechanism to avoid some of these problems [16], but Squid 1.1.20 does not understand entity tags, and so these do not appear in our logs.

Instead of trying to emulate an HTTP cache (partly because our traces lack ‘‘If-Modified-Since’’ header values), we simulate a ‘‘perfect coherency’’ cache, where a request seen for a cached URL will always miss if the origin server would return a different value, and will always hit if the origin server would generate the exact same value. We use the MD5 digest values from our logs, and the fact that our logs (unlike those of a true cache) contain an origin-server response for every request. This allows us to test, on every request, whether the origin server’s response is identical to what would have been cached (we assume no MD5 collisions).

While a perfect coherency cache does not generate exactly the same set of hits as a normal HTTP cache, we believe that it generally will see a slightly higher hit rate (because of the potential for hits after an entry has expired, and because of the potential for hits on dynamic resources). This gives us a more conservative estimate of the value of duplicate suppression, by increasing the apparent performance of the baseline (no duplicate suppression) case.

If one assumes an infinite cache, the perfect coherency simulation is also trivial to implement in our analysis program: if any two responses for URL_j have the same digest value D_i, then the second is a cache hit. If a response for URL_j arrives with a unique digest value, then it represents a cache miss. We decided not to simulate finite caches for other reasons (chiefly, the need to choose and implement a realistic replacement policy), so we avoid the complexity of simulating a finite perfect-coherency cache.

We did, however, simulate two different kinds of infinite cache. In the simpler version, no cache entry is ever deleted. In the more realistic version, used for most of our results, a cache entry is deleted if a newer entry is created for the given URL.

Because our trace was made at a non-caching proxy, we must consider how the use of an infinite cache affects the processing of client requests carrying ‘‘If-Modified-Since’’ headers. If a traced request received a 304 (i.e., not modified) response, then a real caching proxy should also return 304: either because it has no cache entry and forwarded the request, or because it has a coherent entry with the proper Last-Modified date (it should not have an entry with a newer Last-Modified date). If a traced ‘‘If-Modified-Since’’ request received a 200 response, then a cache would also return a 200 response, except in cases of coherency failure (not, alas, impossible).

Our logs contain enough information to discover if a traditional HTTP cache would in fact have yielded a non-coherent response, either because the Last-Modified date matches but the response isn’t identical, or because the expiration deadline is too optimistic. We report elsewhere on the results of this analysis [30].

6. Results

In this section, we present the results of our trace analyses.

6.1. Overall trace statistics

Our 23-day trace included 29,390,845 individual log entries. Of these, 19,119,669 (65%) had a response status of 200 (i.e., carried a full response body). Table 6-1 shows the distribution of the most common status codes; we ignore all but the status-200 responses in our analysis.

Code	Meaning	Count	Fraction
200	OK	19119669	65.05%
204	No Content	833008	2.83%
206	Partial Content	42035	0.14%
301	Moved Permanently	432914	1.47%
302	Found [Redirect]	1503323	5.11%
304	Not Modified	6538793	22.25%
400	Bad Request	413291	1.41%
404	Not Found	409474	1.39%
500	Internal Server Error	54123	0.18%

Table 6-1: Distribution of response status codes

Our analysis program ignored an additional 317,648 log entries (1.1% of the total) because of various consistency checks. This included just 4,104 entries skipped because two responses with the same MD5 digest and URL had different lengths or content-types. This left 18,802,021 ‘‘useful’’ responses, containing just over 200 GBytes of data (not counting HTTP headers), and representing about 32.4 million seconds of total service time. (Of course, most of these transfers took place in parallel; the total clock time covered by the trace is under 2 million seconds.) The mean service time was about 1.5 seconds, and the mean response body size was about 9.5 Kbytes.

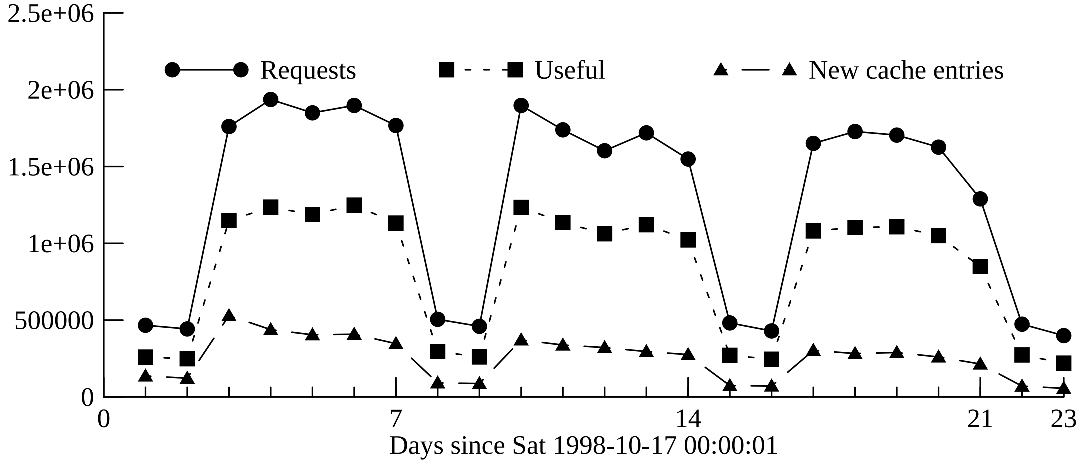

Figure 6-1: Daily statistics (23-day trace)

The request load in these traces varies considerably from day to day, with weekend use about 20%-25% of the mid-week load. Figure 6-1 shows the total request rate, sampled at 24-hour intervals. It also shows the variation in the rate of ‘‘useful’’ entries (with respect to our analysis), and the rate at which new entries would be added to a infinite perfect-coherency cache.

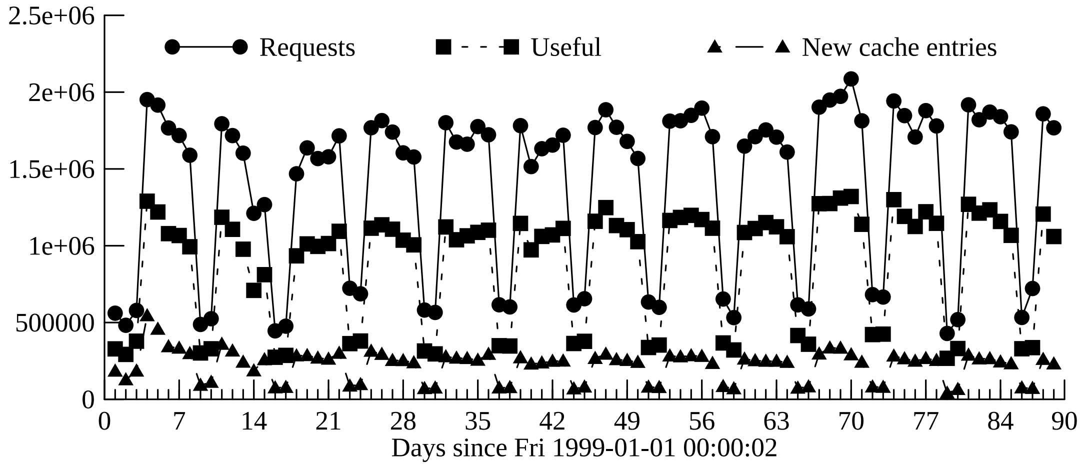

Figure 6-2: Daily statistics (90-day trace)

This rate shows a general decrease over time, but may be at or near an asymptote by the end of our trace, implying that even an infinite cache would have reached a limiting miss-rate after several weeks of operation. In fact, a similar graph for the 90-day trace (figure 6-2) shows no obvious long-term trend in the arrival of new cache entries.

6.2. Frequency of duplication

Over the entire 23-day trace, our infinite cache stored 95 GBytes in 5,788,702 entries, assuming that it kept just one entry per URL. (An infinite cache that stored all past entries for every URL would have required 138 GBytes in 9,172,918 entries.)

Our infinite perfect-coherency cache simulation encountered 9,333,219 cache misses. Therefore, there were 9,468,802 cache hits, for a hit ratio of 50.4%. This ratio is higher than has usually been reported for real caches, presumably because our perfect-coherency cache would store some responses normally not marked as cachable. (If one calculates the hit ratio using the total number of requests, rather than the total number of body-carrying responses, as the denominator, the result is 32.2%. Note that calculation accounts for all of the status 304 ‘‘Not Modified’’ responses seen in the trace.)

For bandwidth reduction, the simple cache hit ratio matters less than the byte-weighted hit ratio, which was 30.4% over the entire trace. In other words, the infinite perfect-coherency cache would eliminate 30.4% of the server response bytes, compared to a non-caching proxy. (In this case, the status 304 ‘‘Not Modified’’ responses are negligible, since they never carry response bodies, and their response headers are typically small.)

Had the duplicate-suppression mechanism been applied to every eligible request, it would have avoided 1,006,324 of the retrievals in our trace, or 5.4% in addition to the cache-hit ratio of 50.4%. Duplicate-suppression ‘‘hits’’ always take the place of cache misses, rather than cache hits, so their benefit always adds to the benefit of simple caching.

When weighted by the number of bytes transferred, best-case duplicate suppression could eliminate 6.2% of the server response bytes, in addition to the bandwidth savings provided by simple caching.

Of the 5,788,702 URLs in the trace, 1,503,822 (25.9%) were the subject of at least one cache hit. 943,807 (16.3%) were duplicated under a different URL.

Since duplicate suppression requires the use of a cache of prior response, all of our analyses assume that the cache is also used in the traditional manner. However, in theory one could use the cache contents only for duplicate-suppression, and otherwise have no cache hits. Applying this scenario to our trace would increase the number of requests where duplicate suppression (rather than simple caching) would apply, to 16.9% of the status-200 responses (rather than 5.4%). The byte-weighted ratio increases to 13.5% (rather than 6.2%). In other words, almost 17% of the response bodies in our trace are identical to a prior response for some different URL (but possibly also identical to a prior response for the same URL).

Figures 6-3 and 6-4 show the unweighted cache-hit ratio and duplicate-suppression ratios, sampled at 24-hour intervals. They also show the unweighted ‘‘total’’ ratio (including both cache hits and duplicate-suppression ‘‘hits’’). Figures 6-5 and 6-6 show the corresponding byte-weighted ratios. Note that the unweighted and weighted figures use slightly different vertical scales, for clarity.

The weighted cache-hit ratio is much lower than the unweighted cache-hit ratio, but the weighted duplicate-suppression ratio is actually higher than the unweighted duplicate-suppression ratio. Therefore, the net bandwidth improvement due to duplicate suppression is more significant than one would expect from the unweighted ratios.

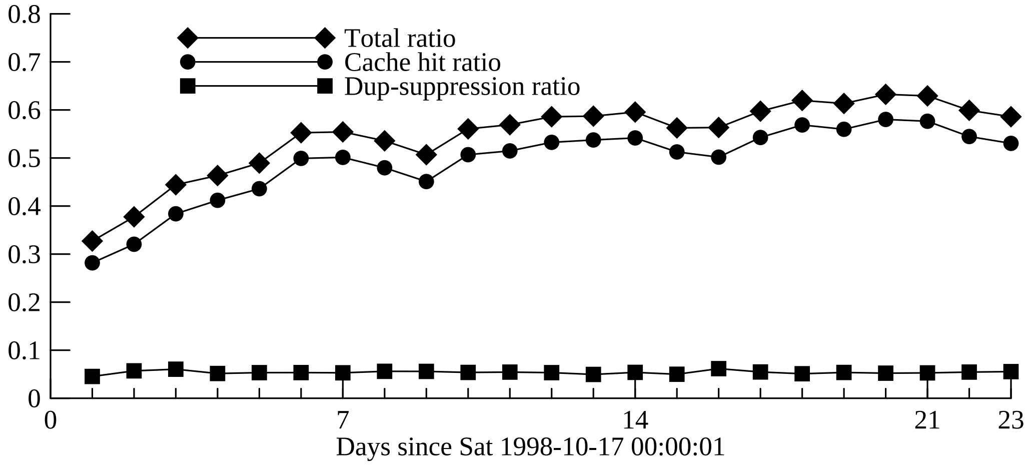

Figure 6-3: Daily ratios (23-day trace)

The cache hit and duplicate suppression ratios in all of figures 6-3 through 6-6 show an initial increase from a cold-cache start, but then the curves level off. The graphs for the 90-day trace seem to show a slightly increasing trend even many after weeks, and linear regressions confirm that there is a small, nearly negligible, positive slope. However, these regressions have relatively poor correlation coefficients. Table 6-2 shows the results of the regressions, both over the entire 90-day trace and over the final 45 days (i.e., treating the first 45 days as a cache warmup period).

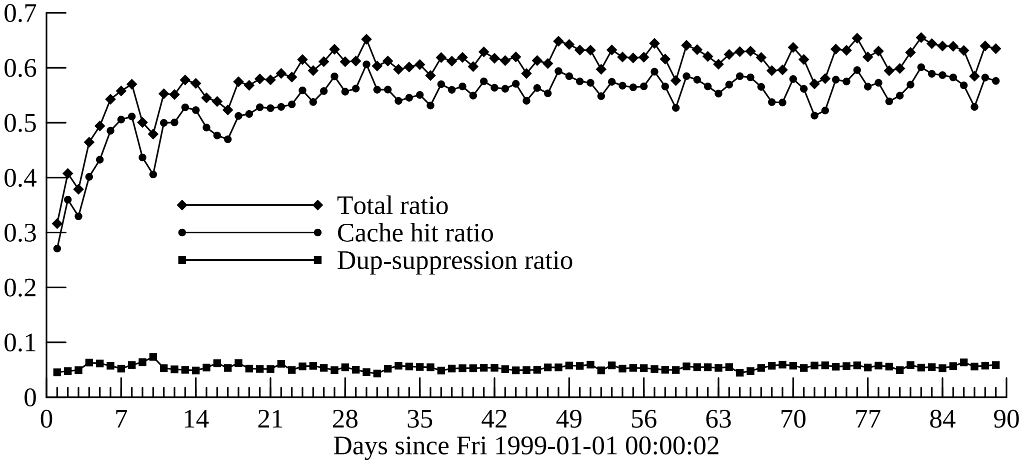

Figure 6-4: Daily ratios (90-day trace)

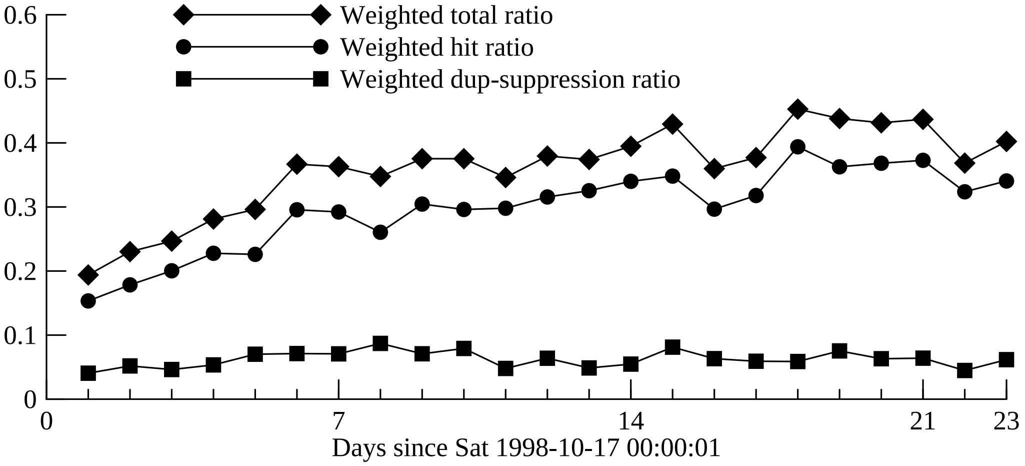

Figure 6-5: Daily ratios, weighted by response size in bytes (23-day trace)

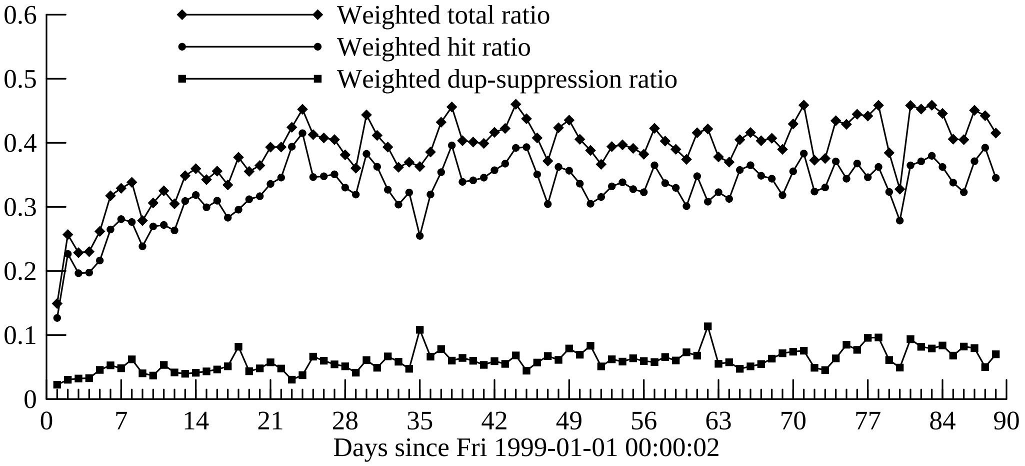

Figure 6-6: Daily ratios, weighted by response size in bytes (90-day trace)

Dependent variable	**Interval**	**Slope (% per day)**	**Correlation coefficient**
cache-hit ratio	all 90 days	0.144	0.642
cache-hit ratio	final 45 days	0.007	0.044
duplication ratio	all 90 days	0.001	0.076
duplication ratio	final 45 days	0.010	0.359
weighted cache-hit ratio	all 90 days	0.105	0.559
weighted cache-hit ratio	final 45 days	0.047	0.238
weighted duplication ratio	all 90 days	0.039	0.591
weighted duplication ratio	final 45 days	0.033	0.286

Table 6-2: Linear regressions for figures 6-4 and 6-6

6.3. Contexts where duplication is most likely

The unweighted duplication ratio never gets much above 6%, on a daily basis, which implies that it might not be worth the extra protocol overhead. This statistic, however, is calculated over all responses in our trace, but one might expect that some sets of resources are far more likely than others to be subject to duplication. If so, one could limit the protocol overhead of duplicate suppression to these contexts, and concentrate the effort where the benefits justify the costs.

Scheme	**Total refs**	**Cache hits**	**Duplicates**	**Total Mbytes**	**Cache hit ratio**	**Weighted HR**	**Dup ratio**	**Weighted DR**
All	18802021	9468802	1006324	205310	0.50	0.30	0.054	0.062
http:	18779187	9464698	1005334	174833	0.50	0.30	0.054	0.055
ftp:	22834	4104	990	30476	0.18	0.35	0.043	0.106

Table 6-3: Overall duplication statistics, by scheme

Most of the traced references are to ‘‘http:’’ URLs, but a small fraction are to ‘‘ftp:’’ URLs. In table 6-3, we show results broken down by URL scheme. The FTP URLs show a slightly lower duplication ratio (DR), but a higher weighted DR. FTP transfers average two orders of magnitude longer than HTTP transfers, and the difference in weighted DRs may reflect frequent duplication of some especially long FTP transfers (such as software downloads). Although FTP transfers are now nearly insignificant to the overall network load, duplicate suppression might be quite beneficial for these downloads.

We then broke down the results by content-type. Table 6-4 lists the ten most frequently-referenced content-types. Table 6-5 lists the ten content-types with the highest duplication ratios, excluding any content-type with fewer than 1000 total references. The ‘‘image/gif’’ type is the only one that shows up in both tables, making it the most obvious choice for general-purpose servers. (The rare ‘‘image/jpg’’ seems to be a non-standard name for ‘‘image/jpeg’’.)

Content-type	Total refs	Cache hits	Dups	Total Mbytes	Cache hit ratio	WHR	Dup ratio	WDR
image/gif	9687564	6907047	691492	35800	0.71	0.57	0.071	0.096
text/html	4897223	1154558	166949	51235	0.24	0.17	0.034	0.013
image/jpeg	2498656	919973	112300	30022	0.37	0.26	0.045	0.040
application/octet-stream	1066245	69426	14577	49112	0.07	0.36	0.014	0.118
text/plain	330130	257981	8494	4019	0.78	0.56	0.026	0.029
application/x-javascript	152154	78252	6367	150	0.51	0.70	0.042	0.027
audio/x-pn-realaudio	31285	15893	115	1181	0.51	0.16	0.004	0.003
unknown	18928	8255	166	104	0.44	0.17	0.009	0.000
text/css	15396	13735	447	36	0.89	0.92	0.029	0.000
application/zip	10913	3054	523	6621	0.28	0.14	0.048	0.046

WHR = byte-weighted cache hit ratio; WDR = byte-weight duplication ratio

Table 6-4: Duplication statistics for 10 most frequent Content-types

However, several audio- and video-related content-types show significant duplication ratios, and sites with such content might also benefit from automatic duplicate suppression.

Content-type	Total refs	Cache hits	Dups	Total Mbytes	Cache hit ratio	WHR	Dup ratio	WDR
audio/x-midi	3216	1420	488	62	0.44	0.31	0.152	0.129
audio/midi	5219	1822	783	129	0.35	0.28	0.150	0.155
image/x-xbitmap	4569	1766	536	4	0.39	0.25	0.117	0.000
image/jpg	1053	616	120	7	0.58	0.43	0.114	0.000
application/java-vm	3936	2634	434	17	0.67	0.65	0.110	0.118
audio/basic	3853	1578	312	2719	0.41	0.05	0.081	0.005
video/mpeg	1458	299	105	3051	0.21	0.14	0.072	0.048
image/gif	9687564	6907047	691492	35800	0.71	0.57	0.071	0.096
application/x-director	1405	651	85	194	0.46	0.51	0.060	0.046
video/x-msvideo	1050	173	61	1921	0.16	0.14	0.058	0.066

WHR = byte-weighted cache hit ratio; WDR = byte-weight duplication ratio

Table 6-5: Duplication statistics, 10 Content-types with highest DR

Table 6-6 list the eleven content-types with the highest weighted duplication ratios, again excluding any content-type with fewer than 1000 total references (we show eleven types to include the popular ‘‘image/jpeg’’). Here, several types commonly used for software distribution show

Content-type	**Total refs**	**Cache hits**	**Dups**	**Total Mbytes**	**Cache hit ratio**	**WHR**	**Dup ratio**	**WDR**
audio/midi	5219	1822	783	129	0.35	0.28	0.150	0.155
audio/x-midi	3216	1420	488	62	0.44	0.31	0.152	0.129
application/octet-stream	1066245	69426	14577	49112	0.07	0.36	0.014	0.118
application/java-vm	3936	2634	434	17	0.67	0.65	0.110	0.118
app’n/x-zip-compressed	2501	344	78	4835	0.14	0.14	0.031	0.100
image/gif	9687564	6907047	691492	35800	0.71	0.57	0.071	0.096
video/x-msvideo	1050	173	61	1921	0.16	0.14	0.058	0.066
video/mpeg	1458	299	105	3051	0.21	0.14	0.072	0.048
application/zip	10913	3054	523	6621	0.28	0.14	0.048	0.046
application/x-director	1405	651	85	194	0.46	0.51	0.060	0.046
image/jpeg	2498656	919973	112300	30022	0.37	0.26	0.045	0.040

WHR = byte-weighted cache hit ratio; WDR = byte-weight duplication ratio

Table 6-6: Duplication statistics, 11 Content-types with highest WDR

up, especially ‘‘application/octet-stream’’, which is also in the top four by reference frequency. This seems to confirm the intuition behind the DRP proposal, that software components are often duplicated.

We counted the number of duplicates we saw for each URL, over the entire trace. Almost 84% of the URLs were never involved in duplication; 16% were duplicated exactly once, and 0.15% were duplicated exactly twice. In other words, very few URLs are duplicated more than once. However, figure 6-7, which plots the cumulative fraction of the total number of distinct duplicate responses as a function of the number of duplicates per URL, shows that some highly-duplicated URLs account for most of the duplication. In fact, half of all duplicate responses come from URLs that give rise to at least 406 different duplicate responses.

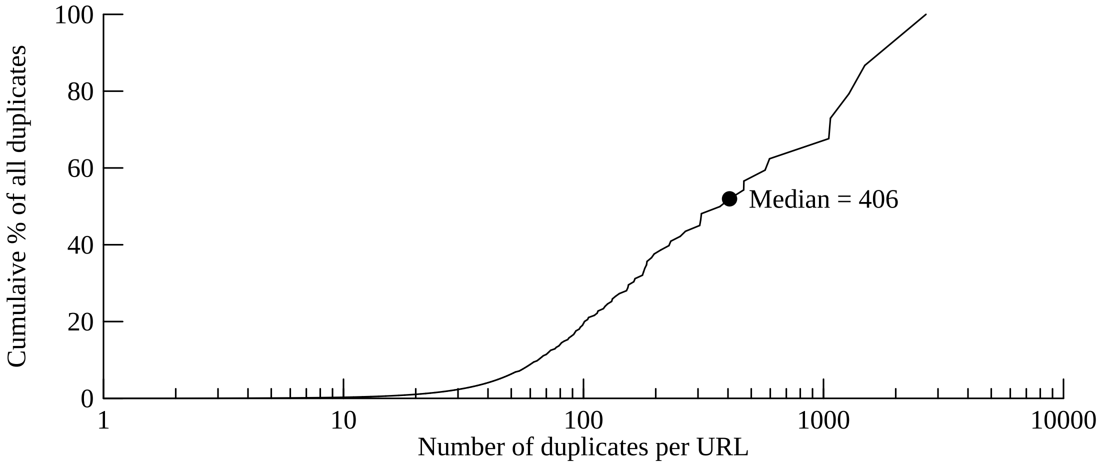

Figure 6-7: Static frequency of replication

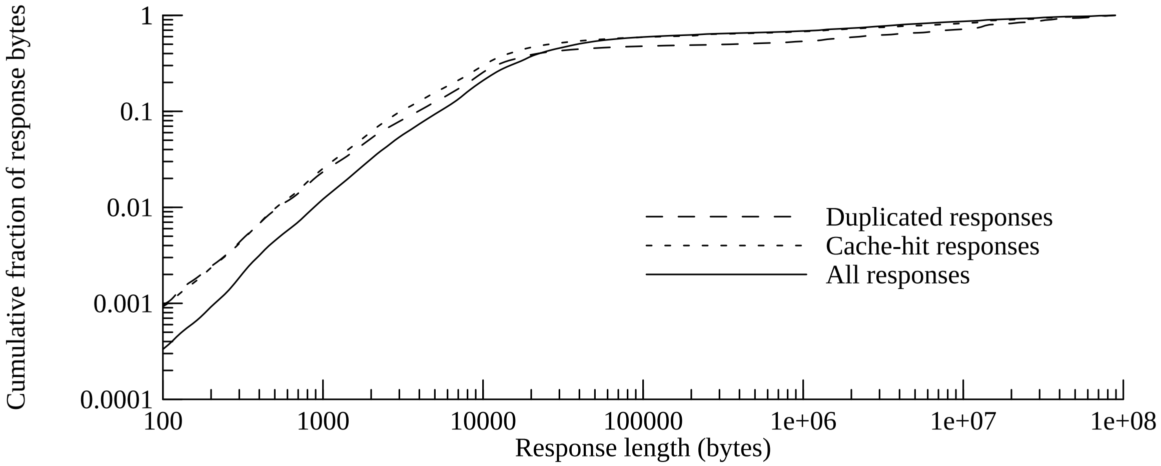

Figure 6-8: Distribution of response lengths

We also looked at the effect of response length on cache-hit and duplication patterns. Figure 6-8 shows the byte-weighted cumulative distributions of response lengths for all responses, for cache-hit responses, and for cache-miss duplicate responses. Generally, cache-hit and duplicates responses are smaller than for the entire trace set. This would seem to contradict our results showing that the byte-weighted duplication ratio is higher than the unweighted DR. However, figure 6-8 reveals that a somewhat larger fraction of duplicated response bytes are from especially long responses. This is especially significant because a large fraction (31.3%) of the response bytes in the trace are carried in responses longer than 1 MByte, even though these account for only 0.075% of the responses. While only 0.058% of the duplicated responses are over 1 MByte, this accounts for 46.4% of the duplicated bytes.

Taking the results in figures 6-7 and 6-8 together, we see that if one limited duplicate suppression only to a small set of frequently-duplicated resources, or a small set of very lengthy resources, one would still see most of the available bandwidth savings. We have not done an analysis to determine if these two sets overlap or are complementary.

6.3.1. Duplication or mirroring?

One might guess that a lot of the apparent ‘‘duplication’’ in our traces comes from intentional mirror sites. For example, a site might be mirrored in geographically separate locations, or might be split across numerous servers for scalability. Use of DNS nicknames (e.g., ‘‘www.cnn.com’’ for ‘‘cnn.com’’) can lead to apparent mirroring, even though the two names resolve to the same server. Because traditional HTTP caching uses the URL as the lookup key, mirrored responses would not result in cache hits, but do result in duplication hits, so the use of mirroring does not, in itself, argue against duplicate suppression. It might even suggest contexts in which duplicate suppression is particularly useful.

We re-analyzed our traces using two transformations on the URLs. In the first, we treated two URLs as identical if one could be transformed into the other by removing a host name prefix (‘‘www[^-\.]*\.’’, to use UNIX-style regular expression syntax); e.g., we treated ‘‘www.cnn.com’’ and ‘‘cnn.com’’ as identical. In the second, if a set of responses with a given MD5 digest all had the same URL except for the hostname part, we treated these as mirrored responses for one resource, rather than duplicate responses for multiple resources. For example, in this transformation we would treat ‘‘example.com/logo.gif’’ and ‘‘example.org/logo.gif’’ as mirrors if the responses had identical digests. (HTTP terms ‘‘/logo.gif’’ the *abs_path* of these URLs.)

	Prefix stripped	Mirrors treated as dups	Dup ratio	Weighted DR
1	No	Yes	0.054	0.062
2	No	No	0.010	0.012
3	Yes	Yes	0.049	0.059
4	Yes	No	0.007	0.009

Table 6-7: Effects of mirroring and aliasing

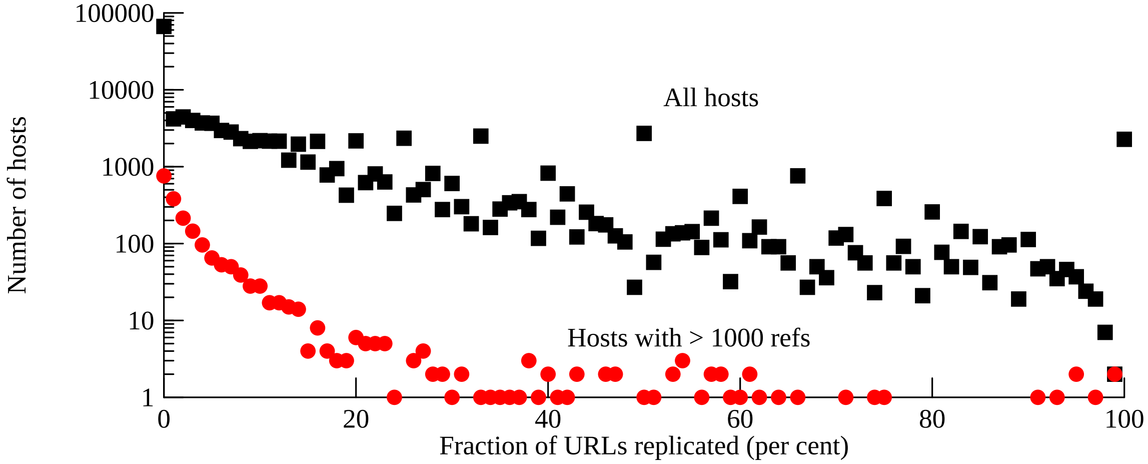

Figure 6-9: Duplication frequency by host

Table 6-7 shows the effects of these two transformations, individually and together, on the duplication ratio and weighted DR. Clearly, a substantial fraction (81%) of the ‘‘duplicates’’ probably are mirrored, at least on a per-URL basis. On the other hand, only 9% of the ‘‘duplicates’’ disappear when we ignored a leading ‘‘www*’’ prefix, indicating that most of the duplicates are not simply the result of DNS nicknames.

6.3.2. Duplication frequency by host

We computed, for each server hostname seen in the traces, the frequency with which its URLs experience duplication. Figure 6-9 plots the results; the upper curve shows all 138,595 hosts in the trace, while the lower one shows just those hosts with at least 1000 references (2028 hosts, or 1.5% of the total). Note that the vertical axis is a log scale; the vast majority of hosts experienced no duplication. In the upper curve, there are spikes at 100%, 50%, 33%, etc., corresponding to hosts seen just one, two, three, etc. times during the trace; while most of these seldom-seen hosts experience no duplication, any duplication at all for a host seen, say, twice means that it has a ‘‘duplication frequency’’ of at least 50%.

The lower curve, for frequently-referenced hosts, avoids these artifacts. It shows that while a small number of such hosts experience frequent duplication, for most hosts the duplication rate is either zero, or on the order of a few per cent.

6.3.3. Additional results

We looked at the number of different URLs associated with each MD5 digest (i.e., each unique response body value). For each of 619 digest values, we found responses with at least 100 URLs; the most frequently duplicated response was seen with 8191 different URLs. In only three of those 619 sets of URLs did we see the same abs_path for every member of the set; most other frequently-duplicated responses are seen under a variety of abs_paths, suggesting that they are not the result of mirroring.

Inspection of these URLs reveals three major categories for frequently-duplicated responses: advertising banners, simple graphic elements (such as dots, icons, etc.), and ‘‘affiliation logos’’ (such as a Netscape logo). Indeed, ad banners accounted for most of the frequently-duplicated responses, which suggests that there might be some resistance to the use of automatic duplicate suppression: ad placement services depend on seeing as many references as possible, and do not appreciate HTTP caching.

6.4. Effect of duplicate suppression on latency

Our traces included total durations for each request. We can project the latency saved by caching or duplicate suppression, if one assumes a constant transfer time for a given response-length from a given URL. Table 6-8 shows the projected fraction of time saved, for caching and for duplicate suppression; we include breakdowns by scheme and for the ten most frequent content-types.

The results in table 6-8, when compared to those in tables 6-3 and 6-4, suggest that the latency savings are roughly proportional to the weighted DR, but do vary in some specific cases.

6.5. Implications for cache design

Since infinite caches are still hard to obtain, we tried to discover how long a cache would have to retain an entry before it paid off, either for duplicate suppression or simple caching. Figure 6-10 shows the distribution of cache-entry ages for the 23-day trace, measured at the instant of either a cache hit or a duplicate-suppression using a given entry. The distributions show peaks at various multiples of one day, suggesting that periodic access patterns result in both cache hits and duplication hits. (We cannot explain the dip at about 9 hours.)

Figure 6-3 showed that the cache-hit ratio increased for at least several weeks, while the duplication ratio leveled off after just a few days. Figure 6-10 shows that cache entries useful for duplicate suppression tend to be younger than those involved in cache hits. For duplicate suppression, the median is about 900 sec., and the 90th percentile is about 25 hours. For caching, the median is about 2000 sec., and the 90th percentile is about 2.3 days. So, in general, retaining a cache entry pays off more quickly (if at all) for duplicate suppression than for caching, and so we suspect that with the right replacement policy, a moderately large cache would deliver most of the available benefits.

Figure 6-10 shows a steep drop in the age distributions past about one week. This is probably an artifact of the 23-day trace length, since by construction it excludes most cache-to-use intervals longer than a fraction of the trace length. The distribution for the 90-day trace, in figure

	Total refs	**Cache hits**	**Duplicates**	**Total transfer time (secs)**	**Time saved by caching**	**Time saved by duplicate suppression**
All	18802021	9468802	1006324	32398818	0.36	0.059
By scheme						
http:	18779187	9464698	1005334	28643899	0.37	0.058
ftp:	22834	4104	990	3754919	0.31	0.067
Top 10 content-types						
image/gif	9687564	6907047	691492	9018443	0.61	0.093
text/html	4897223	1154558	166949	8695820	0.22	0.021
image/jpeg	2498656	919973	112300	4597560	0.27	0.042
application/octet-stream	1066245	69426	14577	4847410	0.34	0.095
text/plain	330130	257981	8494	560865	0.50	0.040
application/x-javascript	152154	78252	6367	60659	0.55	0.050
audio/x-pn-realaudio	31285	15893	115	162854	0.39	0.003
unknown	18928	8255	166	37651	0.20	0.006
text/css	15396	13735	447	8891	0.88	0.026
application/zip	10913	3054	523	1181521	0.26	0.061

Table 6-8: Projected transfer time savings (as fraction of actual time)

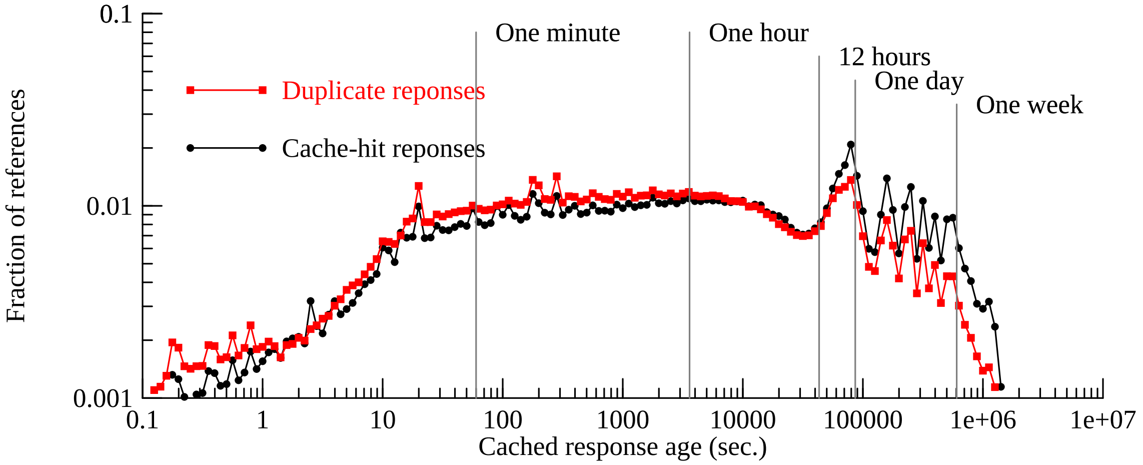

Figure 6-10: Cache entry age distributions (23-day trace)

6-11, is similar, except that it shows a much more gradual decline at ages above one week. Even so, only 6.7% of the cache hits, and 3.9% of the duplicate suppressions, come from responses that have been in the cache for more than a week.

We also looked at the byte-weighted age distributions for both traces. Figures 6-12 and 6-13 show that the byte-weighted distributions, compared to the unweighted age distributions in figures 6-10 and 6-11, are biased towards higher ages. That is, if bandwidth conservation is

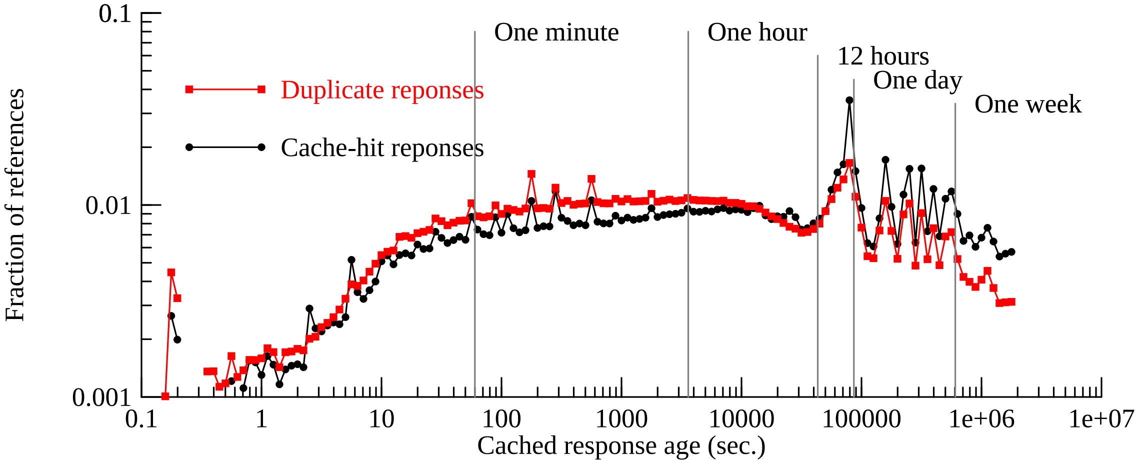

Figure 6-11: Cache entry age distributions (90-day trace)

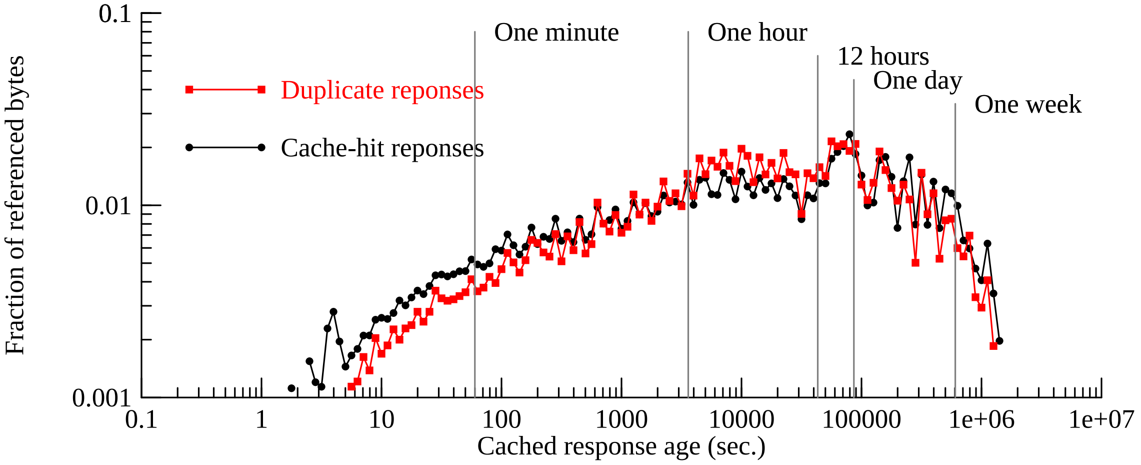

Figure 6-12: Byte-weighted cache entry age distributions (23-day trace)

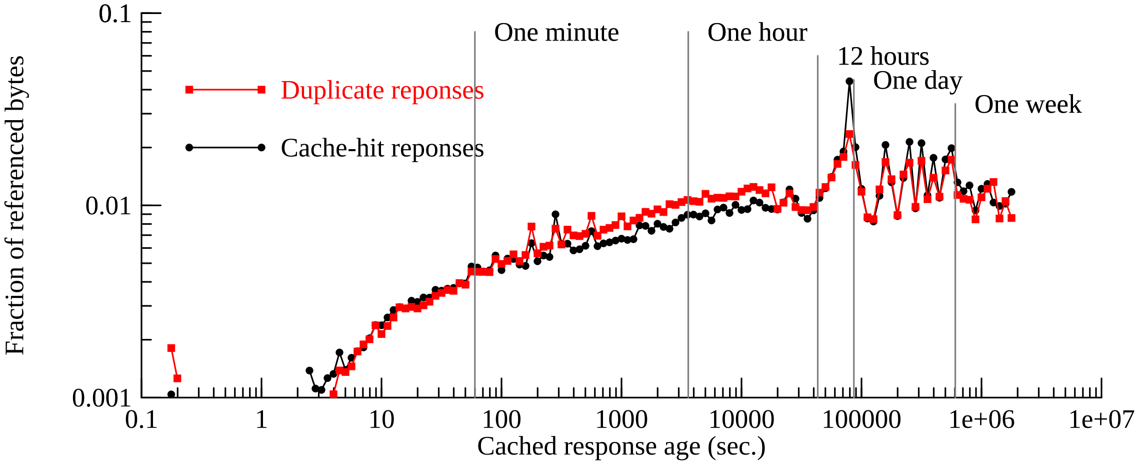

Figure 6-13: Byte-weighted cache entry age distributions (90-day trace)

important, it might be necessary to cache responses for longer than would be necessary simply to avoid HTTP requests.

Table 6-9 summarizes the median and 90th percentile points for the various age distributions. One would expect the values for the 90-day trace to be more reliable, since the 23-day trace suffers more from cold-cache and measurement-window effects. The table shows that duplicate suppression generally pays off with younger cache entries than does traditional caching, if either pays off at all.

Distribution	Trace length	Median	90th percentile
Cache-hit responses	23 days	1995 sec.	2.3 days
Cache-hit bytes	23 days	10000 sec.	3.7 days
Dup responses	23 days	891 sec.	24.8 hours
Dup bytes	23 days	11220 sec.	2.6 days
Cache-hit responses	90 days	4467 sec.	4.6 days
Cache-hit bytes	90 days	28184 sec.	7.3 days
Dup responses	90 days	1413 sec.	2.3 days
Dup bytes	90 days	14125 sec.	7.3 days

Table 6-9: Medians and 90th percentiles for age distributions

As noted in section 5.1, our infinite cache simulation could either retain just the most recent entry for each URL, or all past entries. One might expect that the latter would give more opportunities for duplicate suppression, by providing a large pool of possible duplicates. However, it also artificially increases the cache hit ratio, because it decreases the chances of a coherency miss. (This might be viewed as a simulation bug, since existing HTTP caches cannot discover if an older response is now coherent again, but the proposed delta-encoding extension [32] does provide a means for doing so.) With fewer cache hits, there are more chances for duplicate suppression to pay off. In any event, the difference in hit ratios is insignificant, but the retain-all-entries cache consumes 45% more storage than a ‘‘realistic’’ infinite cache. This suggests that the retain-all-entries policy is neither necessary for good duplicate suppression, nor desirable for a practical (finite-cache) implementation.

6.6. Protocol and computation overheads

The benefits of duplicate suppression are offset by several overhead costs. We evaluated those costs to determine if they overwhelm the benefits.

6.6.1. CPU costs

The computation of a reasonably secure message digest requires significant CPU time compared to many other typical per-message operations. For duplicate suppression, servers could attach an digests to every response they send (which might have other benefits), or caches could compute digests when storing a cache entry. The latter shifts load from servers to caches, and slightly reduces network traffic. More significant, it moves the digest computation off of the critical path for latency (it can be done during idle CPU time), so it should not directly offset the benefits of duplicate suppression.

Even so, digest computation adds load, but apparently not enough to worry about. One study [5] showed that a 90 MHz Pentium can compute MD5 digests at 136 Mbits/sec., or SHA-1 digests at 55 Mbits/sec. We measured a different MD5 implementation [46] on a 333 MHz Pentium-II, which yielded 335 Mbits/sec for 5000-byte messages. On a 500 MHz AlphaServer ES40 (21264 CPU), that implementation reached 386 Mbits/sec for 5000-byte messages. We would expect digest performance to continue to improve with processor clock rates.

We also used DCPI [1] to profile the CPU time used by our modified Squid proxy; the MD5 computation consumed about 2.4% of the total non-idle CPU time (including user-mode and kernel-mode time), implying that digest computation costs are negligible. Squid is not normally considered a fast proxy, but its primary inefficiency lies in the way it misuses the disks [28], which does not significantly affect CPU time (and in any case, our installation does not use a disk cache). Also, we used a modified version of Squid that avoid one particular CPU scaling error [31] and a modified kernel that avoid several CPU scaling problems [3].

Although the digest computation can be done off the cache's critical path, a cache participating in duplicate suppression must still maintain and use an additional lookup table. Lookups using a digest instead of a URL would not occur on every request, but the digest-based lookup table would have to be updated whenever a new response-body is received (or perhaps only when a response carrying a Digest header is received). This update, while not directly on a critical path, might represent a modest overhead during periods of heavy load. Note, however, that because duplicate suppression is a performance optimization, an overloaded cache could simply drop these updates, without harming correctness.

6.6.2. Message-length costs

Proposed duplicate suppression protocols add message-length overhead in two places: the meta-information headers passed from server to client, and the meta-information passed from client to proxy cache. Both include a message digest value (possibly naming the specific algorithm), and some syntactical overhead, on the order of 10 bytes or less per digest. If we assume the use of MD5, the 128-bit digests are encoded in HTTP headers as 24-byte strings; a 160-bit digest, such as SHA-1, would yield 28-byte encodings. So, the total encoded overhead would be under 40 bytes.

Note that the overhead need not include any contribution from a response message digest transmitted by the server with the relevant response itself. This is because (as discussed in section 3.1) the caches can recompute the digest locally. However, server-transmitted response digests have benefits independent of the duplicate suppression protocol: they are useful for end-to-end checking of other HTTP mechanisms, such as Range retrievals and delta encoding, so one might want to transmit them anyway. Therefore, we do not count these digests as part of the overhead of duplicate suppression.

Estimating the ratio of overhead to benefit requires some guesswork, for three reasons:

1. A server might supply meta-information for many resources never actually referenced in a trace, leaving us no way to quantify the upper bound.
2. A server might not send duplicate-suppression headers unless the information had good chance of paying off.

3. Our traces do not include size information for HTTP headers.

In the most optimistic scenario, overhead would only be incurred for those references where duplicate suppression definitely pays off, for a lower bound of about 77 MBytes of meta-information, or just 0.6% of the simulated savings from duplicate suppression. In a more pessimistic scenario, every client request would be 40 bytes longer, and the server-supplied meta-information might be about the same order of magnitude, adding 1434 Mbytes of total meta-information, or about 11.2% of the simulated savings.

The worst-case scenario, in which a server would transmit meta-information for far too many resources, could clearly swamp any savings available from duplicate suppression. That does not make the mechanism useless; it does mean that server implementors must be prudent about how it is used.

We expect that if duplicate-suppression is limited to those cases where it has at least a small chance of paying off, then the message-length overheads will remain significantly smaller than the bandwidth savings. Note, however, that the savings occur only on the server-to-proxy path, while some of the overheads accrue on the client-to-proxy path, which complicates the tradeoff.

7. Security considerations

The duplicate suppression mechanism analyzed in this paper could be spoofed. An attacker would have to arrange for a proxy cache to contain an entry with the same digest value as a targetted response, but with a different response body value. In other words, the attacker needs to find a digest collision.

Security is not the main topic of this paper, but because countermeasures to the creation of digest collisions might change the size of, or cost of computing, a digest value, we briefly address this issue.

There is no known way to create MD5 collisions, short of brute-force search. However, there is some suspicion that MD5 is not collision-proof. To quote from Robshaw’s 1996 analysis [43],

- ‘‘collisions for the compression function of MD5 have been demonstrated, though [not] for the full MD5’’
- ‘‘more, possibly very complex, analytical work is required in designing an attack for MD5’’
- ‘‘collisions for [MD5] have not yet been discovered but this advance should be expected’’

Even if an attacker could find an MD5 collision, a practical attack would have to generate a false response body that has at least a plausible resemblance to the targetted response. If spoofing results in gibberish, it enables a denial-of-service attack but not a way to spread misinformation.

If MD5 turns out to be insufficient, Robshaw suggests that several alternatives exist. For example, the use of HMAC [23] could be used without a secret key to strengthen MD5. This would generate the same number of digest bits (i.e., would add no additional network overhead), but would approximately double the computation costs. SHA-1 [35] or RIPEMD-160 [41], with their longer digest lengths (160 bits instead of 128 bits) also should be stronger than MD5, but would increase network overhead and roughly triple the computation cost. RIPEMD-128 (same digest length as MD5, but slower) might also be acceptable.

8. Future work

With slight additional effort, one could analyze these traces to answer several additional questions. For example, what fraction of duplicates are local to one server (i.e., have different URLs but the same hostname)? One could also model the effect of limited cache sizes and realistic replacement algorithms, to see if this changes the utility of duplicate suppression.

A trace-based simulation study of just one site might not give a representative picture of the benefits of duplicate suppression. This study should be replicated at other sites, and especially with other user communities.

It would also be helpful to study the performance of deployed systems using the DRP protocol [19], which uses signatures to name resources.

9. Summary and Conclusions

Our trace analysis suggests that HTTP duplicate suppression would yield modest, but not negligible, savings in Internet bandwidth consumption and latency. Since these benefits are strictly in addition to those provided by other HTTP caching techniques, duplicate suppression may prove useful in increasing the efficiency of HTTP caches. Duplicate suppression seems to reach its full potential using less cache space than regular HTTP caching, and so the relative benefits may in fact be greater for realistic cache sizes than for our simulated infinite cache.

Duplicate suppression imposes computational and message-length overheads, but these appear to be minimal compared to the benefits.

Duplicate suppression certainly works better in some contexts than in others. It pays off best for multimedia, program binaries, and images, and worst for text (including HTML). Any trend towards greater use of Web-based multimedia and software distribution, especially the use of reusable software components, should increase the utility of duplicate suppression. However, its utility in reducing the bandwidth required for GIF image transmission may require resolution of the deep conflict between caching and ad-impression counting.

Acknowledgments

This report could not have been prepared without the help of several people, especially Andrei Broder and Mark Manasse, who provided explanations of several cryptographic issues. Peter Druschel helped run one of the MD5 benchmarks.

References

[1] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems* 14(4):357-390, November, 1997.

[2] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proc. 1997 USENIX Technical Conference*, pages 289-303. Anaheim, CA, January, 1997.

[3] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. 1998 USENIX Annual Technical Conf.*, pages 1-12. USENIX, New Orleans, LA, June, 1998.

[4] Azer Bestavros and Carlos Cunha. *A Prefetching Protocol Using Client Speculation for the WWW*. Technical Report TR-95-011, Boston University, CS Dept, Boston, MA, April, 1995.

[5] A. Bosselaers, R. Govaerts and J. Vandewalle. Fast hashing on the Pentium. In N. Koblitz (editor), *Advances in Cryptology, Proc. Crypto '96, LNCS 1109*, pages 298-312. Springer-Verlag, 1996. Updated at Eurocrypt '97 as ''Even faster hashing on the Pentium,'' ftp://ftp.esat.kuleuven.ac.be/pub/COSIC/bosselae/pentiumplus.ps.gz.

[6] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. Infocom '99*. New York, NY, March, 1999.

[7] Andrei Broder, Steve Glassman, Mark Manasse, and Geoffrey Zweig. Syntactic clustering of the Web. In *Proc. 6th International World Wide Web Conf.*, pages 391-404. Santa Clara, CA, April, 1997.

[8] Peter B. Danzig, Richard S. Hall, Michael F. Schwartz. A Case for Caching File Objects Inside Internetworks. In *Proc. SIGCOMM '93 Symposium on Communications Architectures and Protocols*, pages 239-248. San Francisco, CA, September, 1993.

[9] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of Change and Other Metrics: a Live Study of the World Wide Web. In *Proc. Symposium on Internet Technologies and Systems*, pages 147-158. USENIX, Monterey, CA, December, 1997.

[10] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proc. Symposium on Internet Technologies and Systems*, pages 83-94. USENIX, Monterey, CA, December, 1997.

[11] Bradley M. Duska, David Marwood, and Michael J. Feeley. The Measured Access Characteristics of World-Wide-Web Proxy Caches. In *Proc. Symposium on Internet Technologies and Systems*, pages 23-35. USENIX, Monterey, CA, December, 1997.

[12] Equilibrium. DeBabelizer Product Information page. http://www.equilibrium.com/debab/.

[13] Jens Ernst, Christopher W. Fraser, William Evans, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proc. ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation*, pages 358-365. Las Vegas, NV, June, 1997.

[14] Li Fan, Pei Cao, Jussara Almeida and Andrei Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proc. SIGCOMM '98*, pages 254-265. Vancouver, September, 1998.

[15] Anja Feldmann, Ramon Caceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. In *Proc. INFOCOMM '99*, pages 107-116. New York, NY, March, 1999.

[16] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2616, HTTP Working Group, June, 1999.

[17] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to Network and Client Variation via On-Demand Dynamic Transcoding. In *Proc. ASPLOS VII*, pages 160-170. Cambridge, MA, October, 1996.

[18] Y. Goland, E. Whitehead, Jr., A. Faizi, S. Carter, D. Jensen. *HTTP Extensions for Distributed Authoring -- WEBDAV*. RFC 2518, IETF, Feb., 1999.

[19] Arthur van Hoff, John Giannandrea, Mark Hapner, Steve Carter, and Milo Medin. *The HTTP Distribution and Replication Protocol*. Technical Report NOTE-DRP, World Wide Web Consortium, August, 1997. http://www.w3.org/TR/NOTE-drp-19970825.html.

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

[21] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An Empirical Study of Delta Algorithms. In *IEEE Soft. Config. and Maint. Workshop*. 1996.

[22] ISO/IEC. *11172-3, Information technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s -- Part 3: Audio* 1993.

[23] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104, Network Working Group, February, 1997.

[24] D. Kristol, L. Montulli. *HTTP State Management Mechanism*. RFC 2109, HTTP Working Group, February, 1997.

[25] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. Symposium on Internet Technologies and Systems*, pages 13-22. USENIX, Monterey, CA, December, 1997.

[26] Hakon Wium Lie and Bert Bos. *Cascading Style Sheets, level 1*. Recommendation REC-CSS1, World Wide Web Consortium, December, 1996. http://www.w3.org/pub/WWW/TR/REC-CSS1.

[27] Tong Sau Loon and Vaduvur Bharghavan. Alleviating the Latency and Bandwidth Problems in WWW Browsing. In *Proc. Symposium on Internet Technologies and Systems*, pages 219-230. USENIX, Monterey, CA, December, 1997.

[28] Evangelos P. Markatos, Manolis G.H. Katevenis, Dionisis Pnevmatikatos, and Michail Flouris. Secondary Storage Management for Web Proxies. In *Proc. 2nd USENIX Symp. on Internet Technologies and Systems*, pages 93-104. Boulder, CO, October, 1999.

[29] Ethan L. Miller, Kennedy Akala, and Jeffrey K. Hollinsworth. Using Content-Derived Names for Package Management in Tcl. In *Proc. 6th Annual Tcl/Tk Workshop*, pages 171-179. San Diego, CA, September, 1998.

[30] Jeffrey C. Mogul. *Errors in timestamp-based HTTP header values*. Research Report 99/3, Compaq Computer Corporation Western Research Laboratory, December, 1999. URL http://www.research.digital.com/wrl/techreports/abstracts/99.3.html.

[31] Jeffrey C. Mogul. Speedier Squid: A Case Study of an Internet Server Performance. *;login:* 24(1):50-58, February, 1999.

[32] Jeffrey C. Mogul, Balachander Krishnamurthy, Fred Douglis, Anja Feldmann, Yaron Goland, and Arthur van Hoff. *Delta encoding in HTTP*. Internet-Draft draft-mogul-http-delta-02, IETF, October, 1999. This is a work in progress.

[33] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proc. SIGCOMM '97*, pages 181-194. Cannes, France, September, 1997.

[34] Jeffrey C. Mogul and Arthur Van Hoff. *Duplicate Suppression in HTTP*. Internet-Draft draft-mogul-http-dupsup-01, IETF, February, 1999. This is a work in progress.

[35] National Institute of Standards and Technology. *Secure Hash Standard*. Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce, April, 1995. http://csrc.nist.gov/fips/fip180-1.txt.

[36] National Laboratory for Applied Network Research (NLANR). NLANR Hierarchical Caching System Usage Statistics. http://www.ircache.net/Cache/Statistics/.

[37] National Laboratory for Applied Network Research (NLANR). Cache Popularity Index. http://www.ircache.net/Cache/Statistics/Popularity-Index/.

[38] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *Proc. SIGCOMM '97*. Cannes, France, September, 1997.

[39] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *Computer Communication Review* 26(3):22-36, 1996.

[40] Vern Paxson. End-to-End Routing Behavior in the Internet. In *Proc. SIGCOMM '96*, pages 25-38. Stanford, CA, August, 1996.

[41] Bart Preneel, Antoon Bosselaers, and Hans Dobbertin. The cryptographic hash function RIPEMD-160. *CryptoBytes* 3(2):9-14, Autumn, 1997.

[42] R. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321, Network Working Group, April, 1992.

[43] M. J. B. Robshaw. On Recent Results for MD2, MD4 and MD5. *RSA Laboratories' Bulletin* 4(12):1-6, November 12, 1996.

[44] Jonathan Santos and David Wetherall. Increasing Effective Link Bandwidth by Suppressing Replicated Data. In *Proc. USENIX 1998 Annual Technical Conference*, pages 213-224. New Orleans, LA, June, 1998.

[45] K. Thompson, G. J. Miller, and R. Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network* 11(6):10-23, November/December, 1997.

[46] Joseph D. Touch. Performance Analysis of MD5. In *Proc. SIGCOMM '95*, pages 77-86. Cambridge, MA, August, 1995.

[47] Duane Wessels. Squid Internet Object Cache. http://squid.nlanr.net/Squid/.

[48] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox . Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. SIGCOMM '96*, pages 293-305. Stanford, CA, August, 1996.

[49] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. Organization-Based Analysis of Web-Object Sharing and Caching. In *Proc. 2nd USENIX Symp. on Internet Technologies and Systems*, pages 25-36. Boulder, CO, October, 1999.

[50] Tong Lai Yu. Data Compression for PC Software Distribution. *Software - Practice and Experience* 26(11):1181-1195, 1996.