Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP

Jeffrey C. Mogul, Yee Man Chan, Terence Kelly
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2004-29
February 18, 2004*

E-mail: JeffMogul@acm.org, ymc@shgc.stanford.edu, terence.p.kelly@hp.com

HTTP, aliasing, duplicate transfer detection

Organizations use Web caches to avoid transferring the same data twice over the same path. Numerous studies have shown that forward proxy caches, in practice, incur miss rates of at least 50%. Traditional Web caches rely on the reuse of responses for given URLs. Previous analyses of real-world traces have revealed a complex relationship between URLs and reply payloads, and have shown that this complexity frequently causes redundant transfers to caches. For example, redundant transfers may result if a payload is *aliased* (accessed via different URLs), or if a resource *rotates* (alternates between different values), or if HTTP's cache revalidation mechanisms are not fully exploited. We implement and evaluate a technique known in the literature as *Duplicate Transfer Detection* (DTD), with which a Web cache can use digests to detect and potentially eliminate all redundant payload transfers. We show how HTTP can support DTD with few or no protocol changes, and how a DTD-enabled proxy cache can interoperate with unmodified existing origin servers and browsers, thereby permitting incremental deployment. We present both simulated and experimental results that quantify the benefits of DTD.

* Internal Accession Date Only
To be published in and presented at the First Symposium on Network Systems Design and Implementation, 29-31 March 2004, San Francisco, CA
Approved for External Publication

Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP

Jeffrey C. Mogul
HP Labs
Palo Alto, CA 94304
JeffMogul@acm.org

Yee Man Chan
Stanford Human Genome Center
Palo Alto, CA 94304
ymc@shgc.stanford.edu

Terence Kelly
HP Labs
Palo Alto, CA 94304
terence.p.kelly@hp.com

Abstract

Organizations use Web caches to avoid transferring the same data twice over the same path. Numerous studies have shown that forward proxy caches, in practice, incur miss rates of at least 50%. Traditional Web caches rely on the reuse of responses for given URLs. Previous analyses of real-world traces have revealed a complex relationship between URLs and reply payloads, and have shown that this complexity frequently causes redundant transfers to caches. For example, redundant transfers may result if a payload is *aliased* (accessed via different URLs), or if a resource *rotates* (alternates between different values), or if HTTP's cache revalidation mechanisms are not fully exploited. We implement and evaluate a technique known in the literature as *Duplicate Transfer Detection* (DTD), with which a Web cache can use digests to detect and potentially eliminate all redundant payload transfers. We show how HTTP can support DTD with few or no protocol changes, and how a DTD-enabled proxy cache can interoperate with unmodified existing origin servers and browsers, thereby permitting incremental deployment. We present both simulated and experimental results that quantify the benefits of DTD.

1 Introduction

Web caches are widely used to save bandwidth and improve latency. However, numerous studies have shown that, in practice, forward proxy caches (i.e., shared Web caches used near clients) incur miss rates of 51-70%, and byte-weighted miss rates of 64-86% [27,40]. Even warm caches with infinite storage cannot eliminate all misses.

In this paper, we are specifically concerned with redundant payload transfers, i.e., cases where a payload is transmitted to a recipient that has previously received it. In a traditional Web cache, each cache entry is indexed by a given URL. If a subsequent request arrives for that URL, and the cache cannot satisfy the request (it "misses"), it forwards the request to the origin server, which normally generates a reply containing a payload (Section 4.2 gives a careful definition for "payload"). If that exact payload has previously been received by the cache, we define this as a redundant payload transfer.

Others have identified the problem of redundant payload transfers on the World Wide Web, quantified its prevalence, and explored a range of possible solutions [2, 16, 28]. According to one measurement, over 20% of payload transfers between origin servers and proxies are redundant [16].

We do not know all causes of redundant transfers. Many result from three common phenomena: *aliasing*, in which the same content is referenced under two different URLs; *rotation*, in which the same content is referenced twice under a single URL, but an intervening reference to that URL resolves to different content; and absent or faulty metadata that causes avoidable revalidation failures.

We previously proposed a technique called *Duplicate Transfer Detection* (DTD) [16] that allows any Web cache to potentially eliminate all redundant payload transfers, regardless of cause. DTD uses message digests to detect redundant transfers before they occur. In its use of digests to detect duplication, DTD is similar to approaches developed for other contexts, e.g., router-to-router packet transfers [32] and file systems for low-bandwidth environments [23]. Unlike an alternative proposal for eliminating redundant HTTP transfers [28], DTD does not require soft state that scales with the number of clients and the size of responses.

In [16] we did not propose a concrete protocol design or describe an implementation of DTD, nor did we measure its impact on client latency. In this paper, we show how one can use standard HTTP, with few or no explicit protocol changes, to support DTD without relying on any additional semantics, naming mechanisms, validation mechanisms, or cooperation with or between origin servers. This allows a DTD-enabled cache to interoperate with *unmodified* existing origin servers and browsers, thereby permitting smooth, incremental deployment. We describe how to implement DTD in a Web cache, and report on experiments showing that it can accomplish its goal of completely eliminating redundant transfers. We quantify the benefits of DTD using both experimental measurements of our implementation, and simulation results.

The main contributions of this paper are a well-defined protocol specification for DTD, the design of a real implementation of DTD, and performance evaluations of DTD.

2 Why eliminate redundant transfers?

Our DTD proposal does not reduce the number of times an HTTP cache must contact an origin server; it only reduces the number of response bodies that must be transferred. What makes this worthwhile?

Eliminating redundant transfers can improve at least four metrics:

Bandwidth: Web caches are often deployed to reduce bandwidth requirements (over half of large companies surveyed in 1997 cited bandwidth savings as their motivation for Web caching [11]). Redundant transfers consume bandwidth and increase peak bandwidth requirements.

Latency: Eliminating a redundant transfer can save latency in two ways: *directly*, by making the result available sooner (i.e., without having to wait for the redundant transfer to finish), and *indirectly*, by reducing channel utilization and thereby reducing queueing delays for subsequent responses.

Per-byte charges: Network tariffs are often flat-rate, but not always. In particular, wireless-data tariffs range from a few dollars to tens of dollars per Mbyte [3]. Redundant transfers on such networks directly waste money.

Energy: Studies have shown that energy consumption for wireless (and hence portable) networking is at least somewhat dependent on the amount of data transferred [10]. Eliminating some redundant transfers might therefore improve battery life.

In our previous study, using two large real-world traces, we showed that roughly 20% of payload transfers between origin servers and proxy caches are redundant [16]. Therefore, a solution to the redundant-transfer problem could yield significant savings on some or all of the metrics listed above. In this paper, we concentrate on quantifying these improvements.

3 Related work

The first published suggestion to eliminate redundant HTTP payload transfers using message digests, and a trace-based evaluation of its impact on Web cache hit rates, appeared in [15]. A recent unpublished undergraduate dissertation [4] develops a similar idea for GPRS Web access.

Santos & Wetherall [32] and Spring & Wetherall [33] describe protocol-independent network-level analogues of DTD that employ *packet* digests to save bandwidth. Muthitacharoen *et al.* designed a network file system for low-bandwidth environments that performs similar operations on chunks of files [23].

Web caches can use payload digests to avoid wasting *storage* as well as bandwidth. We have implemented this natural counterpart of DTD (see Section 8) but we are not the first. Bahn *et al.* report that by using digests to avoid storing redundant copies of payloads a Web cache can reduce its storage footprint by 15% and increase its hit rates [1]. Inktomi Corporation has patented such a scheme [18].

A variety of “duplicate suppression” schemes have been proposed for the Web. These differ from DTD chiefly in that 1) they are typically end-to-end mechanisms requiring the participation of orgin servers, whereas DTD can be used hop-by-hop at any level of a cache hierarchy, 2) they avoid the extra round trip that some variants of DTD suffer upon a miss, and 3) they can reduce but not eliminate redundant transfers. Mogul [19] reviews several duplicate suppression schemes (e.g., the Distribution and Replication Protocol (DRP) of van Hoff *et al.* [38]) and reported that they improve hit rates by modest margins, at best.

Previous studies have shown that redundant payload transfers on the Web are caused by complexities in the relationship between URLs and reply payloads (e.g., aliasing and rotation) [16], and by deficiencies in cache management algorithms and server-supplied metadata [41, 42].

Rhea *et al.* describe a sophisticated generalization of DTD called “Value-Based Web Caching” (VBWC) [28]. Whereas DTD operates on entire payloads, VBWC detects and eliminates redundant transfers at finer granularity by employing fingerprints calculated on variable-sized blocks. Block boundaries are computed as in Spring & Wetherall's approach [33]. In VBWC, editing a file affects only payload blocks in the immediate neighborhood of the change, ensuring that minor changes don't eliminate bandwidth savings. Rhea *et al.* implemented VBWC and evaluated it by polling seventeen popular Web sites; their evaluation also includes comparisons with delta encoding. They did not evaluate VBWC based on an actual client or proxy reference stream.

DTD sometimes entails an additional round trip between client and server, but requires no additional server state. By contrast, VBWC proxies must explicitly track client cache state in order to avoid the extra RTT except in rare circumstances. This is soft state, but it scales with both the number of clients and the size of responses, which makes VBWC less easily deployable than DTD. VBWC is also harder to evaluate using anonymized traces, because existing traces that include only

MD5 digests of response bodies cannot be used to compute partial-payload fingerprints.

VBWC was designed to be run between an ISP's proxy and the end clients. While DTD can be used server-to-client or server-to-proxy, it can also be used proxy-to-client or proxy-to-proxy. In the latter cases, DTD imposes a store-and-forward cost (for computing the digest at the first proxy) on the entire payload, while VBWC's store-and-forward costs are per-block and thus potentially smaller. We do not yet know how significant these overheads are.

4 Duplicate Transfer Detection

Motivated by the wish to eliminate redundant HTTP transfers, we proposed "Duplicate Transfer Detection" (DTD). This solution applies equally to *all* redundant payload transfers, regardless of cause. Here we provide an overview of DTD (derived from [16]), and discuss several general design issues. In Section 5, we will present a more detailed protocol design, showing how DTD can be defined as a simple, compatible extension to HTTP/1.1 [9].

4.1 Overview of DTD

First, consider the behavior of a traditional HTTP cache, which we refer to as a "URL-indexed" cache, confronted with a request for URL U. If the cache finds that it does not currently hold an entry for that URL, this is a cache miss, and the cache issues or forwards a request for the URL towards the origin server, which would normally send a response containing payload P. (If the cache does hold an expired entry for the URL, it may send a "conditional" request, and if the server's view of the resource has not changed, it may return a "Not Modified" response without a payload.)

Now suppose that an idealized, infinite cache retains in storage every payload it has ever received, whether or not these payloads would be considered valid cache entries. A finite, URL-indexed cache differs from this idealization because it implements both an update policy (it only stores the most recent payload received for any given URL), and a replacement policy (it only stores a finite set of entries).

The concept behind Duplicate Transfer Detection is quite simple: If our idealized cache can determine, before receiving the server's response, whether it had ever previously received P, then we can avoid transferring that payload. Such a cache would suffer only compulsory misses and would never experience redundant transfers. A finite-cache realization of DTD would, of course, also suffer capacity misses.

How does the cache know whether it has received a payload P before the server sends the entire response? In DTD, the server (origin server or intermediate proxy cache) initially replies with a digest D of the payload, and the cache checks to see if any of its entries has a matching digest value. If so, the cache can signal the server not to send the payload (although the server must still send the HTTP message headers, which might be different). Thus, while DTD does not avoid the request and response message headers for a cache miss, it can avoid the transfer of any payload it has received previously. We say a "DTD hit" occurs when DTD prevents a payload transfer that would have occurred in a conventional URL-indexed cache.

An idealized DTD cache stores *all* payloads that it has received, and is able to look up a cached payload either by URL or by payload digest. In particular, it does not delete a payload P from storage simply because it has received a different payload P' for the same URL U. A realistic DTD cache, with finite capacity, may eventually delete payloads from its storage, based on some replacement policy.

4.2 What is a "payload"?

We have described DTD as operating on "payloads." In order to precisely specify DTD, we must also precisely specify the term "payload." That is, over what set of bytes is a digest calculated?

HTTP servers (the term "server" includes both origin servers and proxies) can send response messages containing either the full current value of a resource, a partial response containing one or more sub-ranges of the full value, or more complex partial responses (such as with delta encoding [21] or rsync [37]). HTTP responses can also be encoded using various compression formats, or with "chunked" encoding.

Whatever the format of the response, the ultimate client almost always wants to obtain a full current value of the referenced resource.[1] One of us introduced the term "instance" to mean "The entity that would be returned in a status-200 response to a GET request, at the current time, for [...] the specified resource," in an IETF standards-track document specifying how to extend HTTP/1.1 to support "instance digests" [20]. An instance consists of an "instance body" and some "instance headers."

Our DTD design equates "payloads" and "instance bodies." That is, servers provide instance digests, and a cache entry is indexed by the digest of the instance body it stores.

One could imagine an alternative in which DTD's digests are computed on HTTP message bodies, which might be partial responses. However, this seems less likely to eliminate redundant transfers; two partial responses for the same instance might not span the same range.

The "payloads are instance bodies" model works nicely with partial responses. For example, if a client requests bytes 0-10000 of URL X, and the server responds with a digest of the entire instance body, a DTD client checks its cache for a matching instance digest. If such an entry is found, the transfer can be avoided; the client can easily extract the required byte-range from its cache entry, rather than relying on the server's extraction.

Nothing in the DTD design prevents a cache from computing digests on non-instance data (such as partial responses, encoded responses, etc.) and matching incoming instance digests against cached non-instance data. Our intuition, however, is that such matches will occur too rarely to justify the additional overhead.

4.3 Deployment of DTD

DTD is best thought of as a hop-by-hop optimization of HTTP caching,[2] which can be implemented between any HTTP server and client (either one of which could be a proxy; DTD can be implemented between any data sender and receiver). In particular, DTD can be deployed unilaterally by an organization that controls both browser and proxy caches, e.g., AOL or MSN. It can also be deployed incrementally by any implementor of clients, servers, or proxies, because it is always optional for either end of a transfer. In the experiments described in Section 9 we demonstrate that DTD can be enabled purely through proxy modifications, if the origin server supports digest generation.

DTD's main requirement for server implementors is to compute and send instance digests. The algorithm used to compute the digest value *D* must not use too much server CPU time, and the digest representation must not consume too many bytes, or else the cost of speculatively sending digests will exceed the benefits of the DTD hits. Also, the digest must essentially never yield collisions, or else the client could end up with the wrong payload. A cryptographic hash algorithm such as MD5 [29] might have the right properties. We will assume the use of MD5 for this paper; Section 10.1 covers some issues in the choice of digest algorithm.

Note that DTD does not inherently require the client to compute any digests, if all servers send digests. However, to check against transmission errors or servers sending bogus digests, clients should probably compute digests anyway (see Section 10).

5 Protocol design issues

Our previous paper [16] briefly covered protocol design issues for DTD. In this section, we expand that discussion, including mechanisms for suppressing data transfer and specific HTTP mechanisms to support DTD.

5.1 Options for suppressing data transfer

One key aspect of DTD is the mechanism by which the client avoids receiving a payload, if the digest *D* matches an existing cache entry. This could be accomplished by deferring the transfer until the digest can be checked, or by aborting the transfer in progress if the digest matches some cache entry.

In the first category of approaches, the server sends the response headers but defers sending the payload until the client sends an explicit "proceed" message. In the other category, the server sends the payload immediately after the headers, but stops if the client sends an "abort" message. The "proceed" model imposes an extra round-trip time (RTT) on every cache miss, but never sends any redundant payload bytes. The "abort" model imposes no additional delays, but the abort message may fail to reach the server in time to save any bandwidth. Thus, the choice between alternatives requires consideration not only of implementation issues, but also of the magnitude of the RTT, and whether one is more concerned with optimizing bandwidth utilization or latency.

Each of these basic models allows several alternatives. These include:

Pure-proceed: Upon receiving the client's request, the HTTP server replies only with the HTTP headers (including digest *D*). The client sends a "proceed" message if *D* is not found in its cache, and the server sends the HTTP body (payload). Otherwise, no further messages are sent.

Proceed/don't bother: In the pure-proceed alternative, the server might need to buffer responses indefinitely, waiting for a possible "proceed" message. The "proceed/don't bother" alternative addresses this concern by allowing the client to send a "don't bother" message, if digest *D does* match a cache entry; the message allows the server to free the buffer more quickly.

Auto-proceed for short responses: The proceed model risks exchanging an extra set of headers and delaying an extra RTT. For short payloads, the transfer time saved by a DTD hit might not be worth this overhead. The server could optimize the short-payload case by sending the payload immediately for payload sizes below a threshold.

Abort: The server sends the payload immediately after the HTTP headers (as in normal HTTP operation). The client sends a special HTTP "abort" message if digest *D* matches a cache entry, telling the server to terminate the transfer as soon as possible.

Note that in the proceed model, not every payload need be delayed. Web pages often include multiple images; for example, we previously found 8.5 image references per HTML reference in an uncached reference stream, and 1.9 images per HTML reference in a client-cached stream [16]. A client that pipelines [26] its requests for images can also pipeline its "proceed" messages. Thus, the extra RTT delay can be amortized over all of the images on a Web page, rather than being paid once per image.

In this paper, we examine only the pure-proceed model, for reasons of space and simplicity.

5.2 Extending HTTP to support DTD

The changes required to extend HTTP/1.1 [9] to support DTD depend on which transfer-suppression approach is chosen. The "pure proceed" approach to DTD can be implemented without any changes to HTTP/1.1 beyond existing IETF standards-track proposals.

The client first uses mechanisms specified in the Proposed Standard for instance digests [20] to obtain current instance headers, including an instance digest. It obtains these via a HEAD request, which prevents the server from sending an instance body [9, Section 9.4]. If the client finds no cache entry with a matching instance digest, or if a non-DTD server fails to return a digest, the client simply issues a GET request to obtain the full instance body.

This protocol design, while simple, has several drawbacks:

- **It potentially adds one extra RTT per miss:** The client sends both a HEAD and a GET request on a DTD miss, so this could add an extra RTT of latency per request. In practice, most HTTP requests are for images embedded in HTML pages, which allows an HTTP/1.1 client to pipeline some or all of a page's image requests in one transmission (and the server can likewise batch the HEAD and GET responses). So for typical compound Web pages the pure-proceed approach adds *at most* two additional mandatory RTTs: one for the HTML container and one more for *all* of the embedded images.
- **It adds an extra set of request and response headers per miss:** This cuts into the bandwidth savings offered by DTD. Therefore, DTD is not worth doing if the mean savings (in response-body bytes) is smaller than the sum of the mean request and response header lengths (see Section 6.1).
- **It depends on request idempotency:** If the (HEAD, GET) sequence had different side effects than a single GET request on the same URI, this would give DTD incorrect semantics. The HTTP/1.1 specification recommends that "the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval," [9, Section 9.1.1], but some sites might ignore this recommendation. If so, DTD clients might need to apply some heuristics, such as not issuing the extra HEAD request on URLs containing "?", or (perhaps) using DTD *only* for embedded images.
- **The server might never send a digest:** HTTP servers are not required to send instance digests, and there is no (current) mechanism to discover if a server would ever send one. The client could thus incur all of the costs listed above, with respect to a given server, without ever gaining a benefit. Clients might need to cease using this DTD approach with any server that fails to send a digest after some threshold number of requests.

Figure 1 shows an example of the HTTP messages between a client and server for a DTD miss. For a DTD hit, the second pair of messages would simply be omitted. The `Want-Digest` and `Digest` headers are described in RFC 3230 [20]; all other headers are standard in HTTP/1.1 [9].

Using `Want-Digest` and `Digest` is the "right" implementation of DTD, because it works even for partial-content responses, is extensible to digest algorithms other than MD5, and avoids unnecessary digest computations at the origin server. But since RFC 3230 is not widely implemented, we tested DTD using the `Content-MD5` support available in major Web servers (e.g., Apache and IIS). This is sub-optimal because it does not allow the server to avoid computing MD5s when the client has no use for them.

The pure-proceed approach is equally usable hop-by-hop or end-to-end, because any intermediate proxy can generate or check digests. (A proxy-to-proxy implementation must use `Digest` because HTTP/1.1 [9, section 14.15] specifically prohibits proxies from adding `Content-MD5`.) Note that proxy-to-client or proxy-to-proxy DTD could impose an extra store-and-forward delay, while the first proxy computes the digest header. (Some existing proxies might already buffer short responses, in any case.)

6 Trace-based performance analysis

Section 9 presents measured performance of an actual DTD implementation. However, those measurements are driven from a synthetic reference stream, which cannot prove how frequent redundant transfers are in real-world workloads. Here we analyze two real-world traces to show how many redundant transfers, and how many bytes, could be eliminated by DTD.

Relatively few existing client and proxy HTTP traces include the response body digests we needed for our analysis. For example, the trace used by Douglis *et al.* [5] may have been lost in a disk crash; other such traces are unavailable due to proprietary considerations. We re-analyzed the anonymized client and proxy traces from our prior study [16]. These were collected, respectively, at WebTV Networks in September 2000 and at Compaq Corporation in early 1999. The WebTV trace was made with client caches disabled; both traces were made without proxy caching. Both traces include an MD5 digest for each payload transferred. The WebTV trace in-

First client request:

```
HEAD /images/logo.gif HTTP/1.1
Host: example.com
Want-Digest: MD5
```

First server response:

```
HTTP/1.1 200 OK
Date: Tue, 30 Jul 2002 18:30:05 GMT
Digest: md5=HUXZLQLMuI/KZ5KDcJPcOA==
Cache-control: max-age=3600
ETag: "xyzzy"
```

Second client request:

```
GET /images/logo.gif HTTP/1.1
Host: example.com
```

Second server response:

```
HTTP/1.1 200 OK
Date: Tue, 30 Jul 2002 18:30:06 GMT
Digest: md5=HUXZLQLMuI/KZ5KDcJPcOA==
Cache-control: max-age=3600
ETag: "xyzzy"
```

(message body omitted)

Figure 1: Example of HTTP messages (pure-proceed approach).

cludes 326 million references from 37 thousand clients to 33 million URLs on 253 thousand servers over sixteen days; the Compaq trace includes 79 million references from 22 thousand clients to 20 million URLs on 454 thousand servers over 90 days. Many further details of these traces are described in [16] and are omitted here for space reasons.

Given a request for URL X that results in reply instance body B, the following properties may or may not hold:

i) there exists some URL Y such that $Y \neq X$ and B was the most-recent instance body for Y.
ii) there exists some URL Z such that $Z \neq X$ and B was a past instance body for Z, but not the most recent.
iii) B was a past instance body for X, but not the most recent.
iv) B was the most recent instance body for X.

Properties (iii) and (iv) are mutually exclusive, but any other combination is possible, so a total of twelve possibilities exist: a given transaction might have none of these properties (if it has never been seen before), or several at once (e.g., both most recent for X and most recent for $Y \neq X$).

We analyzed both the WebTV and Compaq traces according to this categorization. The results are in Tables 1 and 2 respectively. The cold-start results cover the entire traces. Consistent with our earlier methodology [16], for the warm-start results we (only somewhat arbitrarily) warm the simulated cache with the first 186 million references (for WebTV) or 50 million references (for Compaq).

In the WebTV warm-start results, 10% of the transfers involve payloads never before seen in the trace ("new payloads"); these will miss in any kind of cache. Another 87% have property (iv), for which a traditional, infinite cache with perfect revalidation would avoid a payload transfer. (This "hit rate" seems high, but remember that the WebTV trace was made with client caches disabled.) The remainder, about 3%, are transfers that DTD would avoid. In other words, a traditional URL-indexed cache would see a miss rate of at least 13%, compared to a DTD-cache miss rate of 10%; DTD would eliminate 23% of a conventional cache's misses.

In the Compaq warm-start results, 37% are new payloads, and 55% have property (iv). The remainder, about 8%, are transfers that DTD would avoid. A traditional cache would see a miss rate of 45%, versus a DTD-cache miss rate of 37%; DTD would eliminate roughly 18% of a conventional cache's misses for this trace.

If we restrict the DTD implementation to save at most one entry per URL (i.e., to store no more entries than a traditional cache), then the DTD cache will require transfers for properties (ii) and (iii), but will still avoid transfers for property (i). In this situation, DTD would avoid 2.6% of the transfers in the WebTV trace, and 5.8% of the transfers in the Compaq trace, assuming a warm cache. (These values are the sums of the *Warm-start Transfers* column for rows where property (i) holds and property (iv) does not.)

Weighting the results by bytes transferred better describes bandwidth savings, of course. Looking just at the warm-cache data, new (i.e., mandatory-transfer) payloads account for 30% of the WebTV bytes, and 57% of the Compaq bytes. Variations of property (iv), hits for a perfect traditional cache, account for 64% of the WebTV bytes, and 34% of the Compaq bytes. The transfers that DTD would avoid account for 5% of the WebTV bytes, and 9% of the Compaq bytes.

In other words, a traditional URL-indexed cache would see a byte-weighted miss rate of at least 36% for the WebTV trace, compared to a DTD-cache miss rate of 30% (66% vs. 57% for the Compaq trace). In terms of the *reduction* in the number of bytes sent from the origin server, DTD would save (relative to a URL-indexed cache) 15% for the WebTV trace, and 14% for the Compaq trace.

6.1 Overheads from the proceed model

Because the proceed model for DTD causes an extra pair of request and response headers when the digest does not match, to evaluate the overall byte-transfer savings for this model we must compare the bytes saved by DTD (for properties (ii) and (iii)) with the number of

property				Cold-start		Cold-start		Warm-start		Warm-start		
iv	iii	ii	i	Transfers	%	MBytes	%	Transfers	%	MBytes	%	Current reply payload was...
0	0	0	0	36,573,310	11.22	609,935	32.40	13,915,207	9.94	245,010	30.40	never returned before
0	0	0	1	6,047,586	1.85	39,205	2.08	2,332,816	1.67	15,735	1.95	most-recent for other URL
0	0	1	0	94,375	0.03	1,937	0.10	43,313	0.03	1,066	0.13	returned for other URL, not most recent
0	0	1	1	2,070,537	0.64	8,820	0.47	908,075	0.65	3,715	0.46	
0	1	0	0	1,048,493	0.32	35,074	1.86	465,865	0.33	16,906	2.10	returned for current URL, not most recent
0	1	0	1	129,349	0.04	3,089	0.16	62,776	0.04	1,551	0.19	
0	1	1	0	150,533	0.05	2,189	0.12	67,477	0.05	1,093	0.14	
0	1	1	1	681,840	0.21	3,350	0.18	309,030	0.22	1,655	0.21	
1	0	0	0	131,262,060	40.26	662,120	35.17	52,607,080	37.56	272,289	33.79	most recent for current URL
1	0	0	1	138,927,549	42.61	490,892	26.08	64,263,811	45.88	231,911	28.78	
1	0	1	0	290,628	0.09	2,202	0.12	168,472	0.12	1,143	0.14	
1	0	1	1	8,784,417	2.69	23,740	1.26	4,916,756	3.51	13,857	1.72	
				326,060,677		1,882,552		140,060,678		805,928		Totals

Table 1: WebTV trace categorization.

property				Cold-start		Cold-start		Warm-start		Warm-start		
iv	iii	ii	i	Transfers	%	MBytes	%	Transfers	%	MBytes	%	Current reply payload was...
0	0	0	0	30,591,044	38.77	512,562	59.53	10,575,651	36.58	182,372	56.56	never returned before
0	0	0	1	3,504,391	4.44	49,967	5.80	1,291,369	4.47	21,793	6.76	most-recent for other URL
0	0	1	0	148,533	0.19	1,357	0.16	49,339	0.17	490	0.15	returned for other URL, not most recent
0	0	1	1	604,076	0.77	2,721	0.32	229,799	0.79	1,146	0.36	
0	1	0	0	1,554,331	1.97	10,521	1.22	612,795	2.12	3,741	1.16	returned for current URL, not most recent
0	1	0	1	130,356	0.17	1,010	0.12	48,965	0.17	430	0.13	
0	1	1	0	164,992	0.21	1,091	0.13	62,984	0.22	432	0.13	
0	1	1	1	264,100	0.33	1,812	0.21	112,359	0.39	789	0.24	
1	0	0	0	20,492,740	25.97	166,360	19.32	7,230,114	25.01	59,824	18.55	most recent for current URL
1	0	0	1	19,126,071	24.24	106,183	12.33	7,587,555	26.24	47,811	14.83	
1	0	1	0	165,425	0.21	943	0.11	65,695	0.23	397	0.12	
1	0	1	1	2,167,290	2.75	6,442	0.75	1,046,725	3.62	3,198	0.99	
				78,913,349		860,970		28,913,350		322,421		Totals

Table 2: Compaq trace categorization.

extra header bytes spent on the new-payload transfers. We can ignore property (iv) by assuming that these references could be cache hits. DTD (warm-start) saves a mean of 3036 bytes of payload transfer for each new-payload reference in the WebTV trace (warm-start), and 2857 bytes for each new-payload reference in the Compaq trace. These savings are much larger than the mean request+response header sizes reported in previous studies (e.g., [6, 13]) [3] so the proceed model does not waste too much of the potential savings.

DTD requires digests in response headers (for MD5, 24 bytes plus about 10 bytes of syntax overhead), which further reduces savings. However, digests are useful for integrity checks, and so might be sent even without DTD.

6.2 If-None-Match with multiple entity tags

HTTP/1.1 supports the use of *entity tags* to validate cache entries: a server may provide an instance-specific entity tag in the `ETag` response header, and a client may send this entity tag back to the server in an `If-None-Match` request header to check if its cache entry is still valid. `If-None-Match` may carry multiple entity tags, in which case the server can return "304 Not Modified" (along with the current entity tag) if any of those tags is current.

This feature would allow a non-DTD cache to avoid transfers when property (iii) holds. Referring to the warm-start columns in Tables 1 and 2, we see that this could avoid at most 0.6% of the transfers and 2.6% of the bytes for the WebTV trace, and 2.9% of the transfers and 1.7% of the bytes for the Compaq trace.

However, these are upper bounds, since this simple analysis assumes that every response carries an entity tag, and the servers always use exactly one entity tag per distinct instance body. Neither is true in practice; only 66% of the responses in the WebTV trace carried entity tags, and we know that some servers can assign different entity tags to identical instance bodies. In summary, DTD avoids transferring significantly more bytes than could be avoided using multiple entity tags in `If-None-Match`.

6.3 Multiple cache entries per URL?

The full benefit of DTD accrues when the cache stores more than one payload per URL. The most natural clean-slate DTD cache design treats *payloads* rather than URLs as the basic storage type. URLs are merely one way to index into this underlying store; payload digests are another. The cache may therefore store multiple payloads for a given URL, and also payloads that are not the most-recent response for *any* URL (as in the case of rotated resources). These properties, while desirable, might be difficult to retrofit onto some legacy cache implementations; how much do they help? It helps for references that have either property (ii) or (iii) while having neither property (i) nor (iv). These represent just 0.4% of the warm-start transfers in the WebTV trace, but 2.5% of the warm-start transfers in the Compaq trace, so it probably is useful to store multiple payloads per URL.

7 Model-based latency analysis

The analysis in Section 6 concentrates on the number of bytes that could be saved using DTD, which may be of economic interest to network operators. End users, however, care more about latency. Predicting the latency effects of change to Web protocols can be difficult, since so many variables can affect overall latency.

We have developed a simple model for understanding when pure-proceed DTD might improve latency over a traditional Web cache. This model ignores issues such as response pipelining, network congestion, TCP algorithms such as slow-start, and correlations of the hit ratio and duplication ratio with other parameters, but it can help guide intuition.

Given these parameters:

$$\begin{aligned} RTT &= \text{round trip time, cache to server} \\ BW &= \text{effective link bandwidth, bits/sec} \\ L_{\text{resp}} &= \text{response length, bits} \\ HR_{\text{Conv}} &= \text{conventional-cache hit ratio} \\ HR_{\text{DTDonly}} &= \text{DTD-only hit ratio} \\ T_{\text{lookup}} &= \text{Cache-lookup latency} \end{aligned}$$

then we can derive these latencies (if we over-simplify by assuming that HTTP headers are negligible in length):

$$\begin{aligned} T_{\text{ConvHit}} &= T_{\text{lookup}} \\ T_{\text{ConvMiss}} &= T_{\text{lookup}} + RTT + L_{\text{resp}}/BW \\ T_{\text{DTDonlyHit}} &= T_{\text{lookup}} + RTT + T_{\text{ConvHit}} \\ T_{\text{DTDMiss}} &= T_{\text{lookup}} + RTT + T_{\text{ConvMiss}} \end{aligned}$$

The extra RTT in $T_{\text{DTDonlyHit}}$ and T_{DTDMiss} comes from the HEAD operation that a DTD cache performs after the conventional lookup misses. The extra T_{lookup} in $T_{\text{DTDonlyHit}}$ and T_{DTDMiss} comes from the need to do lookups both on the URL and the digest in those cases.

We simplify by assuming that $T_{\text{lookup}} = 0$, a reasonable approximation for a well-implemented cache.

We can then express the expected latencies for conventional and DTD caches:

$$\begin{aligned} E_{\text{Conv}} &= HR_{\text{Conv}} \times T_{\text{ConvHit}} \\ &\quad + (1 - HR_{\text{Conv}})T_{\text{ConvMiss}} \end{aligned}$$

Scenario	RTT	Bandwidth	Break-even response size (bytes) WebTV	Compaq
Cellphone	100ms	10Kb/s	415	549
Modem	100ms	56Kb/s	2325	3075
DSL	30ms	384Kb/s	4783	6325
WAN	42ms	6000Kb/s	104629	138367

Table 3: Examples of model output.

$$\begin{aligned} E_{\text{DTD}} &= HR_{\text{Conv}} \times T_{\text{ConvHit}} \\ &\quad + HR_{\text{DTDonly}} \times T_{\text{DTDonlyHit}} \\ &\quad + (1 - (HR_{\text{Conv}} + HR_{\text{DTDonly}}))T_{\text{DTDMiss}} \end{aligned}$$

DTD improves the expected latency if $E_{\text{DTD}} < E_{\text{Conv}}$, which (by algebra) is true if

$$BW < \frac{L_{\text{resp}} \times HR_{\text{DTDonly}}}{RTT(1 - (HR_{\text{Conv}} + HR_{\text{DTDonly}}))} \quad (1)$$

DTD is thus more likely to pay off as the effective link bandwidth and/or RTT decrease, and as the transfer length and hit ratios increase.

We evaluated Equation 1 using warm-cache hit-ratio values taken from the WebTV and Compaq trace analyses in Tables 1 and 2 and various combinations of RTT and bandwidth. Table 3 shows the results for several scenarios: "cellphone," "modem," "DSL," and "WAN," corresponding respectively to the results shown later in Figures 4(a), 4(b), 4(c), and 6. The break-even response sizes shown in the table imply that DTD would improve latency on cellphone and modem links, and perhaps on DSL links, given the typical mean response sizes summarized in Table 4 of [16]. DTD would hurt latency on high-speed WAN links except if its use were restricted to relatively large responses.

8 Implementation design and experience

Most of the new code required for DTD, using the proceed model, is located in cache implementations. (We also needed server support for digests; we relied on existing support for `Content-MD5`, which is only partially appropriate; see Section 5.2.) Both clients (browsers) and proxies have caches; for our experiments, we limited ourselves modifying a proxy cache server. By running a "private" proxy cache co-located with a browser, we can emulate most of the benefits of integrating DTD into a browser cache. (It should be simpler to add DTD to a browser cache than it was to add it to a proxy cache.)

We chose to implement the pure-proceed approach to DTD as modifications to the Squid proxy server [34] (version 2.4.STABLE7). Our code is available from `http://devel.squid-cache.org/dtd/`. The major changes we made are:

- Creating a "payload" datatype separate from a cache entry. This inverts the existing data-structure dependence between a payload and a URL.

- Indexing into the payload database by digest as well as by URL.
- Generating a preliminary HEAD request to obtain the server's digest.
- Checking the returned digest for DTD-related HEAD requests, and generating a GET request if the digest is not found in the cache, or if no digest is returned.

Our modified Squid uses "Duplicate Storage Avoidance" (DSA). Each distinct payload (i.e., with a given digest) is stored only once; if the payload is current for several URLs, the URL-indexed entries incorporate the payload by reference (see `http://devel.squid-cache.org/dsa/`).

The DTD and DSA changes together involve about 3420 lines of mostly simple but tedious "diffs" to Squid; much of the new code represents modified versions of existing Squid code. About one third of the new lines are pre-processor directives (e.g., "#ifdef").

A cache that supports partial content (HTTP status-206 responses) must be careful not to associate an entire-instance digest with a stored partial-instance body, or else DTD could unwittingly supply incomplete bodies. Our implementation does not yet support partial content.

In hindsight, the choice to modify Squid may have been a mistake. The existing Squid code is extremely complex and hard to understand, and we found many bugs in our own code that resulted from our failure to maintain poorly documented invariants expected by the rest of Squid. We know some bugs remain.

9 Experimental results

The analysis in Section 6, based on traces of real users, predicts the bandwidth savings from DTD, but cannot tell us how DTD affects latency. To help answer this question, we ran experiments using our modified version of Squid.

9.1 Experimental design

We tested our DTD implementation in two different environments The first was an "Emulated-WAN" environment, in which the two systems (server, proxy+client) were physically close, and connected by a 10 Mbit/sec LAN. We then emulated a variety of WAN environments using the Dummynet [30] feature of FreeBSD, which allowed us to choose a variety of latency and bandwidths between the server and proxy, enabling us to measure how DTD performance varies with network characteristics. The second was a "Real-WAN" environment, using a server at Worcester Polytechnic Institute (WPI) in Massachusetts, while the DTD-capable proxy and the client ran on a system at the University of Michigan.

In our tests, we ran the proxy (modified or unmodified) on the same system as the client, to simulate the use of a client cache with or without support for DTD. All systems were otherwise unloaded, except for the real-WAN origin server.

All of the hosts ran Linux, except for the emulated-WAN server which ran FreeBSD. The server at WPI uses Apache/1.3.12, while the emulated-WAN server uses Apache/2.0.47. For the emulated-WAN experiments, the proxy/client was a 550 MHz Pentium III and the server was a 466 MHz AlphaServer DS10L. For the real-WAN experiments, the proxy/client was a 4-CPU 450 MHz Pentium II and the server was a 600 MHz Pentium III.

We measured a mean RTT of 42 msec for the real-WAN path, and approximate effective bandwidths from 6.1 to 7.7 Mbits/sec. In the emulated-WAN tests, we used Dummynet to impose symmetric RTTs of 0, 30, and 100 msec, and bandwidth limits of 10K, 56K, 384K, 1.5M, and 10M bits/sec.

We ran trials with file (body) sizes (not including HTTP headers) of 2^i bytes, for $i = 10, 11, \ldots, 20$; i.e., between 1KB and 1MB.[4] Each file byte was derived from a pseudo-random number generator, thus making it difficult for any network element (such as a modem) to compress the files and change their effective transfer sizes.

For each combination of network characteristics and body size, we ran experiments using three different proxy configurations: no proxy, unmodified Squid, and our DTD-capable modified Squid proxy. With unmodified Squid, we ran trials where the references were arranged to be compulsory cache misses, and trials where the references were guaranteed to be cache hits. With our DTD-capable Squid, we ran compulsory-miss, guaranteed-hit, and DTD-only-hit trials; the last category were references where we arranged that the cache contained an entry with a matching digest value, but not a matching URL. We arranged compulsory cache misses by restarting the proxy software with a cold cache as necessary; we arranged guaranteed hits by careful choice of the reference sequence, and by ensuring that the working set was much smaller than the cache size.

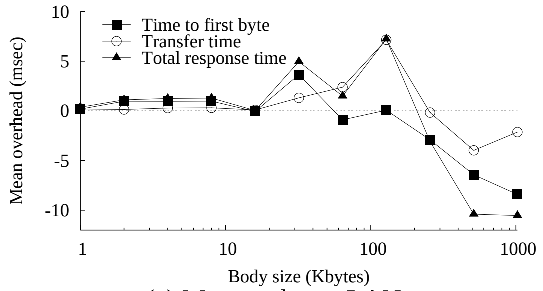
In each set of experiments, we measured end-to-end response time using httperf [22]. This program reports the latency between issuing a request and receiving the first byte of the response headers (time-to-first-byte, or TTFB), as well as the latency between receiving the first byte of the response headers and the last byte of the response body (transfer duration, or TD). For the 1KB body size, the headers and body might fit into one packet, in which case TD would be negligible. The total response latency is thus TTFB+TD. In each trial, we used httperf to fetch "bunches" of 10 distinct files with the same length.

For a given network configuration, we measure latencies for one bunch for each combination of body size and proxy configuration, then repeat that set of measurements

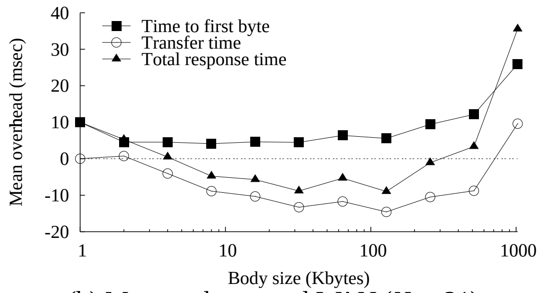
N times. Results in this paper show the mean for $N = 9$ unless otherwise noted.

9.2 Measured overheads

The use of a proxy server introduces overheads that would not be present if our DTD implementation were integrated into a client cache. Also, Squid is known to add significant latency due to fundamental design choices [17]. We can estimate the overheads imposed by our implementation strategy of using Squid rather than an integrated client cache; we do this by comparing the no-proxy latencies with the latencies for cache-miss retrievals via unmodified Squid.



(a) Measured over LAN



(b) Measured over real WAN ($N = 21$)

Figure 2: Overhead imposed by unmodified Squid

Figure 2 shows the overheads imposed by unmodified Squid connected to the server over both a full-speed LAN and over the WAN path described above. In the LAN case, Squid adds almost no latency larger than the trial-to-trial measurement errors (which cause some of the negative "overheads" in Figure 2(a); these errors are below 2% of the total latencies). Overheads from our WAN tests (Figure 2(b)) are harder to interpret, although using unmodified Squid seems to consistently *improve* the transfer times for most body sizes. This effect also holds when we run experiments using an emulated WAN with similar delay and bandwidth. We cannot offer a plausible explanation, but because most of the results in this paper compare performance for our modified Squid against the unmodified version, rather than against the no-proxy case, we leave this mystery to others.

On our LAN, the TTFB latency difference between a Squid miss and a no-proxy operation, for most body sizes, is about 1 msec. This places an upper bound on the cache-lookup latency, because Squid imposes other overheads beyond this lookup, and so confirms our assumption in Section 7 that the lookup latency is negligible.

DTD requires the origin server to send the digest of the payload (body). In our experiments, we use the MD5 digest algorithm, whose computation imposes some cost [36]. In principle, servers could cache MD5 computations for frequently-accessed content. Also, Moore's Law suggests that MD5 computation will decline in cost relative to speed-of-light latencies. However, current servers (such as Apache) do not cache MD5 values, so the use of DTD adds this computational overhead. We quantified the cost by comparing the latencies for no-proxy retrievals with and without MD5 computation enabled at the Apache server.

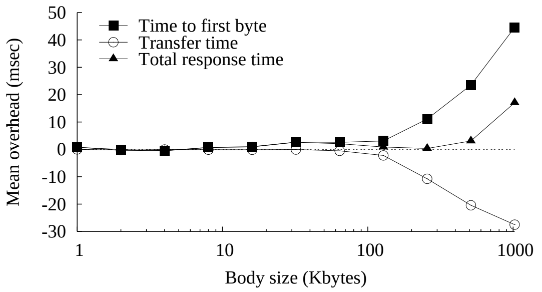


Figure 3: Overhead for MD5 computation

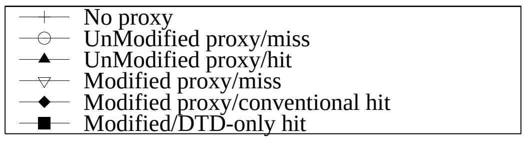
Figure 3 shows the overheads that MD5 imposes for a LAN-based, proxyless configuration. For bodies smaller than 128 KBytes, the overheads are negligible (under 3 msec). For larger bodies, MD5 computation adds measurable overhead, but still less than a tenth of the absolute response time (e.g., 1218 msec for 1024-byte bodies). The increase in response time is *smaller* than the increase in TTFB for these larger sizes, probably because the MD5 pass effectively prefetches the file into the server's file buffer; this prefetching (as Figure 3 implies) makes the TCP transfer slightly more efficient.

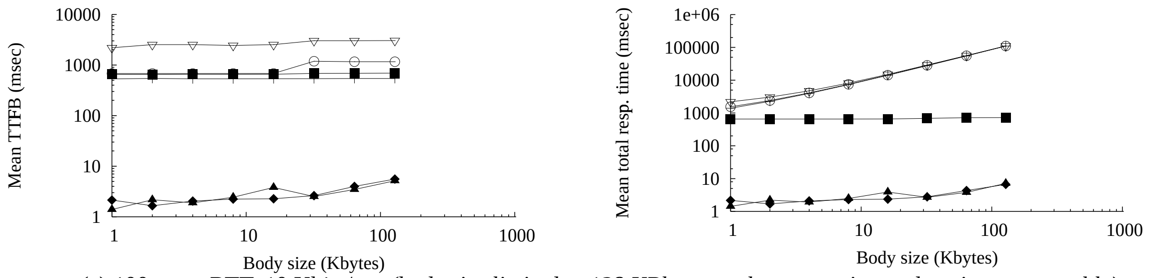
9.3 Emulated-WAN experiments

Figure 4 shows time-to-first-byte and total response time results, in the left and right columns respectively, for selected emulated-WAN experiments. For reasons of space, we only show results for: ($RTT = 100msec$, $10Kbits/sec$), a plausible cellphone link; ($RTT = 100msec$, $56Kbits/sec$), a typical dialup modem; ($RTT = 30msec$, $384Kbits/sec$), a DSL connection to a regional server; and ($RTT = 100msec$, $10Mbits/sec$), a bad case for DTD because the RTT and bandwidth are both high.

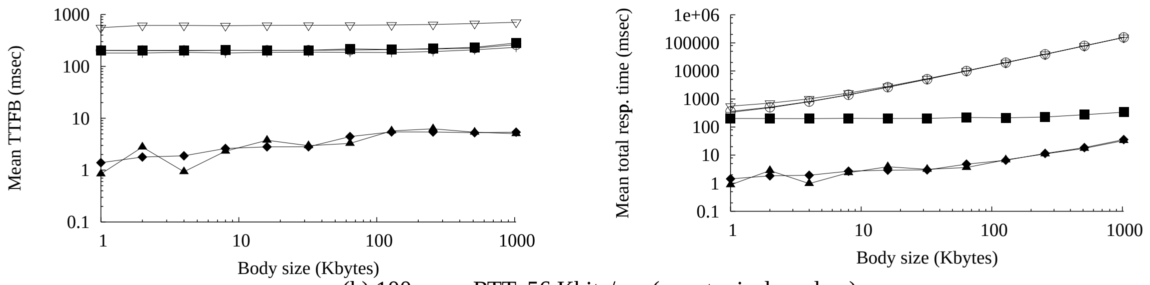
For all combinations of network parameters that we tested, the TTFB latency for a DTD-only hit is slightly above one RTT (approximately the TTFB of a cache miss), as we would expect from the cost of the HEAD

■ Key for all graphs in this figure

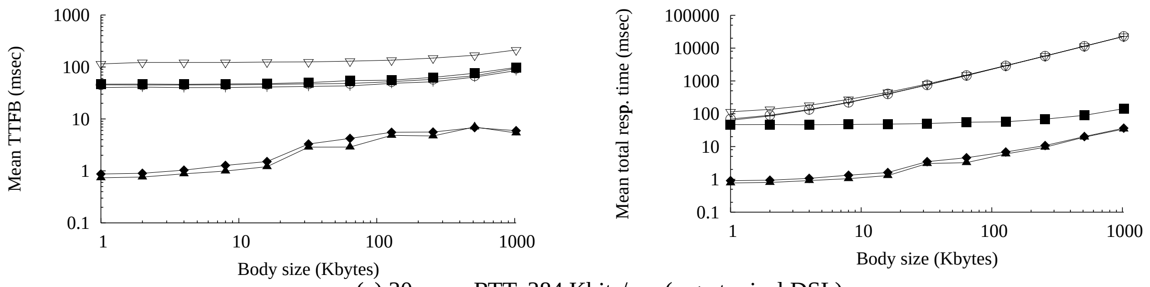




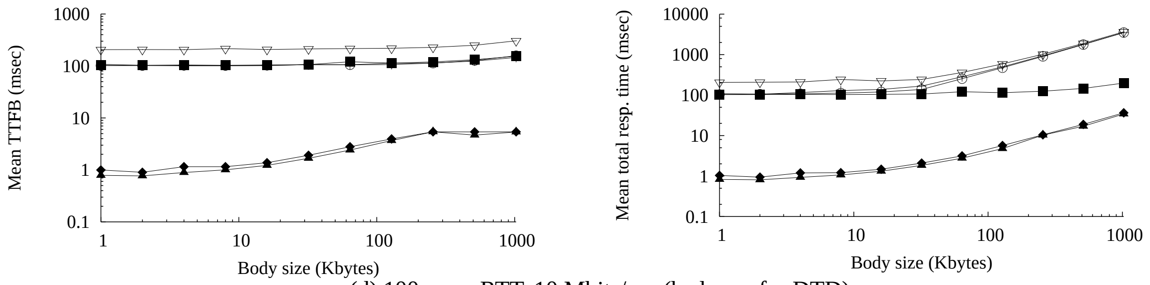
(a) 100 msec RTT, 10 Kbits/sec *(body size limited to 128 KBbytes, to keep experiment durations reasonable)*



(b) 100 msec RTT, 56 Kbits/sec (e.g., typical modem)



(c) 30 msec RTT, 384 Kbits/sec (e.g., typical DSL)



(d) 100 msec RTT, 10 Mbits/sec (bad case for DTD)

Figure 4: Emulated-WAN results

operation. The total response latency for a DTD-only hit is also approximately one RTT, because no body is transferred from origin server to cache. (The cache is co-located with the client, so there is almost no transfer cost between those agents.)

The total latency for compulsory miss by a DTD-capable cache will be one RTT higher than that of a traditional cache. This is clearly visible in the left column of figures (the log scale makes it less visible in the right column, where results are sometimes dominated by bandwidth-induced delays). This is a penalty that a DTD cache must make up by its improved latency on DTD-only hits, with respect to the conventional misses that they displace.

A DTD-only hit should never have a higher total latency than a conventional miss by a non-DTD cache, but it can be much lower if the conventional miss incurs a large transfer cost. For example, in Figure 4(a–c), at a body size of just 8 KBytes, the total latency is significantly lower for a DTD-only hit than for a conventional miss. In Figure 4(d), however, DTD shows no latency benefit except for very large body sizes, because the high bandwidth minimizes transfer cost, while the high RTT dominates total latency.

Note that while Figure 4 shows that DTD-only hits can be much faster than the conventional misses they replace, without knowing the various hit ratios (see Section 7) one cannot infer whether DTD provides a net benefit.

9.4 Real-WAN experiments

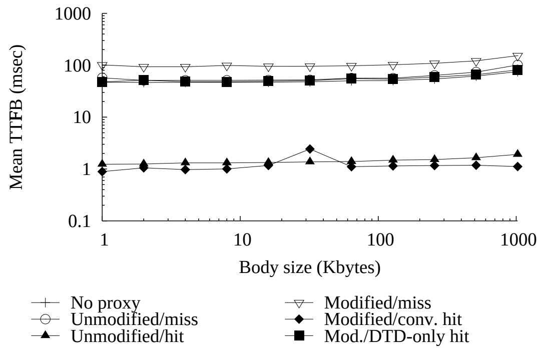


Figure 5: Real WAN – Time-to-first-byte results

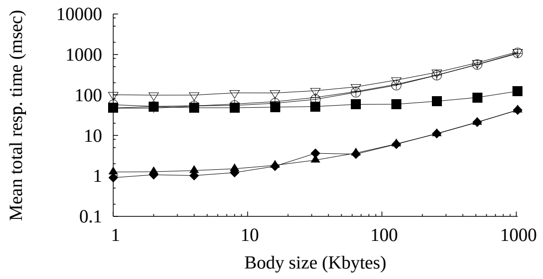


Figure 6: Real WAN – total response time results

Figure 5 and 6 show, respectively, the time-to-first-byte and total response time results for our real-WAN experiments. (In this experiment, $N = 21$.) These results agree quite closely with our emulated-WAN results (not shown in Figure 4) for similar RTT and bandwidth.

9.5 Implications of results

Our experimental results generally confirm the analytic model in Section 7, although our experiments do not attempt to model miss ratios.

We can evaluate whether DTD is beneficial at a particular point in the parameter space. For this example, we assume the miss ratios reported from the WebTV trace in Section 6, 10% for a DTD client cache vs. 13% for a conventional client cache, and assume that these ratios are independent of response size. Using the results in Figure 4(b), a modem user with $(RTT = 100msec, 56Kbits/sec)$ who retrieves a number of 8 KByte files would have a mean expected latency improvement using DTD of about 15 msec (compared to an overall expected mean, without DTD, of 185 msec). The same user retrieving a number of 32 KByte files would see a mean improvement of 126 msec (vs. an overall non-DTD mean of 661 msec).

A user on a slower network, with $(RTT = 100msec, 10Kbits/sec)$, would see even larger improvements from DTD. However, a user of our relatively good WAN connection would see a net latency loss from DTD for body sizes below a break-even point of about 64 KBytes. Since most Web responses are smaller than that, on good WAN links one might only want to use DTD for special tasks such as downloading software (the original motivation for DRP [38]).

10 Security considerations

Measures that improve the performance of computing systems often create subtle security vulnerabilities, and caching is a prime example. Timing attacks on processor memory hierarchies have been known for decades, e.g., the famous TENEX password attack [35, pp. 183–4]. Recently Felten *et al.* have described variants applicable to Web caching [8]. DTD adds at least two additional security problems.

First, if an attacker can generate payload digest collisions, then she can cause a DTD proxy to deliver incorrect payloads. The details are omitted here but are available in [14]. The attack is straightforward and can be prevented through the use of secure message digest functions (see Section 10.1).

A more subtle problem involves information leakage; interestingly, the attack does *not* rely on timing information of any kind.[5] A server can exploit DTD to learn the contents of a client's cache:

1. User Bob's browser and the `nosy.com` server employ DTD.
2. Bob issues a request for uninteresting URL `http://nosy.com/humdrum.html`.
3. `nosy.com` replies with digest(`naughty.gif`), even though it never receives or serves requests for this interesting payload.
4. Bob's browser fails to retrieve the full payload, thereby revealing that Bob already has it.

Sophisticated implementations of this attack might employ JavaScript within HTML pages to systematically search a client's cache for interesting payloads, analogous to the timing attacks described by Felten *et al.* [8]. Attacks of this form can be *detected* easily, by simply retrieving a full payload and verifying the digest previously obtained from the server. Furthermore such attacks can be *avoided* if the client simply refrains from employing DTD when communicating with untrusted sites. Another possible countermeasure is to employ DTD only *within* sites; in the example above, Bob's browser would always fetch payloads except when it found a match supplied by the same server. This ensures that DTD reveals nothing about Bob's surfing that the server doesn't already know. However this approach may severely limit the benefits of DTD, because most aliasing occurs *across* sites rather than within sites [16].

10.1 Choice of digest algorithm

DTD would be unreliable if the digest function were prone to accidental collisions under normal usage. MD5 might not be sufficient for widespread deployment; if not, one could achieve an arbitrarily low rate of accidental collisions by increasing the hash size, at the cost of slightly higher overheads. (Henson [12] discusses some risks associated with digest-based protocols; we disagree with some of the conclusions in that paper.)

DTD would be vulnerable to attack if it were computationally feasible to generate digest collisions deliberately. Our work has assumed the use of MD5 [29], but MD5's collision-resistance has been questioned [31]. Other algorithms, such as SHA1 [24], might be more appropriate.

11 Future work

We see many possible extensions of this work. We would like to explore and evaluate the protocol alternatives in Section 5, and perhaps to unify DTD with similar techniques such as rsync [37]. We would also like to see the trace-based analysis of Section 6 applied to a broader set of traces. One could also improve on our synthetic benchmarks by using miss-ratio and response-length distributions taken from traces.

Neither our model nor the original Squid code base supports pipelining, which is known to benefit HTTP performance in general [25], and ought to improve the tradeoff in favor of DTD; evaluation of a pipelined DTD cache would require shifting to a new code base.

Because a DTD cache, unlike a traditional cache, might store multiple entries per URL, cache replacement policies designed for traditional caches might interact poorly with DTD. We suspect that the most natural replacement policy for DTD is to redefine an existing policy with respect to unique instances rather than to URLs. While we have not yet evaluated such policies, we believe that a DTD cache with such a policy will not suffer a higher miss rate than a conventional URL-indexed cache with the analogous policy.

12 Summary and conclusions

This paper has described how Duplicate Transfer Detection can be implemented in HTTP without explicit protocol changes, and briefly sketched several alternative designs. We showed, using two real-world traces, how DTD could reduce miss rates and bandwidth requirements—14% to 15% of the bytes transferred in our traces. We provided a simple model to show when use of DTD should reduce expected latency relative to a conventional cache. We described a simple implementation of DTD for Squid. Using tests of real and emulated WANs, we showed measurements that clarify the conditions under which DTD reduces overall latency. For realistic hit ratios and response sizes, DTD does provide a net latency benefit for some common network environments.

Acknowledgments

We thank Microsoft for allowing us to study their data, especially Stuart Ozer, Arnold de Leon, and Jay Logue (and others listed in [15]) for gathering the trace, and David Surovell and Jake Brutlag for describing how WebTV client caches work. HP Labs gave us access to the Compaq trace and generous equipment support. Glenn Cooper (HP Labs) and Jeff Kopmanis (U. Michigan AI Lab) provided computer support.

We would especially like to thank Alex Rousskov, for spending late nights adding aliased-content support to Polygraph, and for answering many novice questions. Duane Wessels helped us with Squid. Mikhail Mikhailov lent us a server that made our WAN tests possible, configured it to order, and answered numerous questions. Anja Feldmann, James Hall, and Kevin Jeffay each provided information on request sizes. Fred Douglis, the reviewers from both USITS and NSDI, and our shepherd Geoff Voelker provided helpful comments.

References

[1] H. Bahn, H. Lee, S. H. Noh, S. L. Min, and K. Koh. Replica-aware caching for Web proxies. *Computer Communications*, 25(3):183–188, Feb. 2002.

[2] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the Web. In *Proc. 6th WWW Conf.*, Apr. 1997.

[3] cellular-news. GPRS architecture tariffs. `http://www.cellular-news.com/gprs/tariffs.php`, 2002. Tariff data seems to have disappeared from this page since 2002.

[4] A. Clark. Optimising the Web for a GPRS link. Undergraduate dissertation, Univ. of Cambridge, 2002.

[5] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: A live study of the World Wide Web. In *Proc. 1st USITS*, pages 147–158, Dec. 1997.

[6] A. Feldmann, J. Rexford, and R. Caceres. Efficient

policies for carrying Web traffic over flow-switched networks. *IEEE/ACM Trans. Networking*, 6(6):673–685, Dec. 1998.

[7] E. W. Felten, Sept. 2003. Personal communication.

[8] E. W. Felten and M. A. Schneider. Timing attacks on Web privacy. In *Proc. of 7th ACM Conference on Computer and Communications Security*, Nov. 2000.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol—HTTP/1.1, June 1999.

[10] J. Flinn, E. de Lara, M. Satyanarayanan, D. S. Wallach, and W. Zwaenepoel. Reducing the energy usage of office applications. In *Proc. IFIP/ACM Int'l Conf. on Distributed Systems Platforms (Middleware 2001)*, pages 252–272, Heidelberg, Germany, Nov. 2001.

[11] B. Hannigan, C. D. Howe, S. Chan, and T. Buss. Why caching matters. Technical report, Forrester Research, Inc., Oct. 1997.

[12] V. Henson. An analysis of compare-by-hash. In *Proc. HotOS IX*, Lihue, HI, May 2003.

[13] F. Hernandez-Campos, K. Jeffay, and F. Smith. Tracking the evolution of Web traffic: 1995-2003. In *Proc. MASCOTS*, Orlando, FL, Oct. 2003.

[14] T. Kelly. *Optimization in Web Caching*. PhD thesis, University of Michigan, July 2002.

[15] T. Kelly. Thin-client Web access patterns: Measurements from a cache-busting proxy. *Computer Communications*, 25(4):357–366, Mar. 2002.

[16] T. Kelly and J. Mogul. Aliasing on the World Wide Web: Prevalence and performance implications. In *Proc. 11th Intl. World Wide Web Conf.*, pages 281–292, Honolulu, HI, May 2002.

[17] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance issues of enterprise level Web proxies. In *Proc. SIGMETRICS*, pages 13–23, Seattle, WA, June 1997.

[18] P. Mattis, J. Plevyak, M. Haines, A. Beguelin, B. Totty, and D. Gourley. U.S. Patent #6,292,880: “Alias-free content-indexed object cache”, Sept. 2001.

[19] J. C. Mogul. Squeezing more bits out of HTTP caches. *IEEE Network*, 14(3):6–14, May/June 2000.

[20] J. C. Mogul and A. V. Hoff. Instance digests in HTTP. RFC 3230, IETF, Jan. 2002.

[21] J. C. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. Delta encoding in HTTP. RFC 3229, IETF, Jan. 2002.

[22] D. Mosberger and T. Jin. `httperf`: A tool for measuring Web server performance. In *Proc. First Workshop on Internet Server Performance*, pages 59–67, Madison, WI, June 1998.

[23] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proc. 18th SOSP*, pages 174–187, Oct. 2001.

[24] National Institute of Standards and Technology. Secure hash standard. FIPS Pub. 180-1, U.S. Dept. of Commerce, Apr. 1995. `http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.txt`.

[25] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proc. ACM SIGCOMM*, pages 155–166, Sept. 1997.

[26] V. N. Padmanabhan and J. C. Mogul. Improving HTTP latency. In *Proc. 2nd WWW Conf.*, pages 995–1005, Chicago, IL, Oct. 1994.

[27] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison Wesley, Dec. 2001.

[28] S. C. Rhea, K. Liang, and E. Brewer. Value-based Web caching. In *Proc. WWW 2003*, pages 619–628, Budapest, May 2003.

[29] R. L. Rivest. RFC 1321: The MD5 message-digest algorithm, Apr. 1992.

[30] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31–41, 1997.

[31] M. J. B. Robshaw. On recent results for MD2, MD4 and MD5. *RSA Labs Bulletin*, 4(12):1–6, Nov. 1996.

[32] J. Santos and D. Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *Proc. USENIX Annual Technical Conf.*, June 1998.

[33] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. ACM SIGCOMM*, pages 87–95, Aug. 2000.

[34] Squid Team. Squid Web proxy cache, Aug. 2002. `http://www.squid-cache.org/`.

[35] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992. ISBN 0-13-588187-0.

[36] J. Touch. Performance analysis of MD5. In *Proc. SIGCOMM*, pages 77–86, Cambridge, MA, Aug. 1995.

[37] A. Tridgell and P. Mackerras. The `rsync` algorithm. Technical Report TR-CS-96-05, Dept. of Computer Science, Australian National University, June 1996.

[38] A. van Hoff, J. Giannandrea, M. Hapner, S. Carter, and M. Medin. The HTTP distribution and replication protocol. Technical Report NOTE-DRP, World Wide Web Consortium, Aug. 1997.

[39] D. Wallach, Sept. 2003. Personal communication.

[40] D. Wessels. *Web Caching*. O'Reilly, June 2001.

[41] C. E. Wills and M. Mikhailov. Examining the cacheability of user-requested Web resources. In *Proc. 4th Web Caching Workshop*, Apr. 1999.

[42] C. E. Wills and M. Mikhailov. Towards a better understanding of Web resources and server responses for improved caching. In *Proc. 8th WWW Conf.*, May 1999.

Notes

[1]In some rare cases, the client may only want to render a well-defined sub-part, such as a chapter of a PDF file.

[2]The “proceed” model, described in Section 5.1, also supports end-to-end use.

[3]We based this conclusion on other traces, because the Compaq trace does not include this data, and the header-size data in the WebTV trace appears to be unreliable, possibly the result of incorrect logic for recording header lengths in the trace-gathering process.

[4]Most Web responses are at the low end of this range; we previously summarized results from several traces showing mean sizes between 6,054B and 21,568B, and medians between 1,821 and 4,346B [16].

[5]We thank Flavia Peligrinelli Ribeiro for pointing out this attack. As far as we can determine [7, 39], this form of attack has not previously been reported in the literature.