HTTP-Level Deduplication with HTML5

Franziska Roesner and Ivayla Dermendjieva
Networks Class Project, Spring 2010

Abstract

In this project, we examine HTTP-level duplication. We first report on our initial measurement study, analyzing the amount and types of duplication in the Internet today. We then discuss several opportunities for deduplication: in particular, we implement two versions of a simple server-client architecture that takes advantage of HTML5 client-side storage for value-based caching and deduplication.

1 Introduction

In our project, we examine HTTP-level duplication in Internet traffic. Our project consists of two components: a measurement component and an implementation component. In the measurement study, we analyze a number of browsing traces (both from user internet browsing as well as from a crawler) to determine the amount of duplication in HTTP traffic on the Internet today. Previous studies of this kind (like [9] and [8]) created or used traces of Internet traffic that are now outdated, and we expect that the nature of traffic has changed somewhat since then.

Through our measurements, we find that there is indeed a significant amount of duplication in HTTP traffic, largely in data from the same source rather than among sources. However, we find that the use of compression, which is now widely supported in browsers, may be the simplest and fastest way to reduce this duplication.

Nevertheless, we include in our project also an implementation component, since we see several reasons why an infrastructure such as the one we prototype might be desirable for web servers. The goal of our described system is to perform HTTP-level deduplication by leveraging the new HTML5 client-side storage feature. In particular, we describe and examine two different possible implementations.

The rest of this paper is structured as follows: In Section 2 we discuss related work on which we build, as well as how our project differs from other approaches; in Section 3 we discuss our measurement study and results; in Section 4 we discuss two possible implementations of our system; in Section 5 we attempt to answer the high-level question “Is this worth it?”; and finally we conclude in Section 6.

2 Related Work

A number of researchers have previously considered deduplication using value-based fingerprinting and caching in a variety of contexts. The basic building block for many of these techniques is Rabin fingerprinting [7], a fingerprinting method that uses random polynomials over a finite field. This method is often chosen for deduplication because it is efficient to compute over a sliding window and because its lack of cryptographic security is irrelevant for deduplication purposes.

Early work by Manber [3] uses Rabin fingerprints to identify similar files in a large file system. Muthitacharoen et al. [5] use a similar mechanism for a low-bandwidth network file system (LBFS), which aims to make remote file access efficient by reducing or eliminating the transmission of duplicated data. In [5], the authors use Rabin fingerprints only to determine chunk boundaries, but then hash the resulting chunks using a SHA-1 hash. We follow this method in this project, though we use MD5 instead of SHA-1.

Spring and Wetherall [9] examine duplication on the level of IP packets, finding repetition using fingerprints. They find a significant amount of duplication in network traffic. In particular, even after Web proxy caching has been applied, they find an additional 39% of web traffic to be redundant. We find similar results in the browsing traces that we analyze in Section 3.

This idea that most traffic not caught by web caches is still likely to contain duplicated content was also pursued in [8], which is the most similar to our work here. Like these authors, we consider duplication at the HTTP level, the motivation being that redundant data transferred over HTTP links is not always caught by web caching due to both resouce modification and aliasing. In that work, the authors use a duplication-finding algorithm similar to [5] and the one we describe in the next section. However, our contribution is different from this work in two main ways: (1) The Internet has changed substantially since the publication of [8], and thus our measurement results update the understanding of the amount of redundancy in HTTP traffic; and (2) Our implementation does not require a proxy and instead takes advantage of HTML5 client-side storage. The advent of this feature is the first time that this type of function can be done transparently in the browser, making it easy for individual web servers to deploy to clients.

Other deduplication methods have also been proposed and/or are in use. Beyond standard name-based caching, Mogul et al. [4] discuss the benefits of delta encoding and data compression for HTTP. Delta encoding implements deduplication by transferring only the difference between the cached entry and the current value, leveraging the fact that resources do not usually change entirely. The result of their study is that the combination of delta encoding and data compression greatly improve response size and delay for much of HTTP traffic.

3 Measurement

Before embarking on any implementation, we recorded a number of browsing traces and analyzed the amount of duplication within them. This allows us to characterize the amount and type of duplication in the Internet today and to see if value-based deduplication would provide a benefit over existing deduplication techniques (name-based caching and compression).

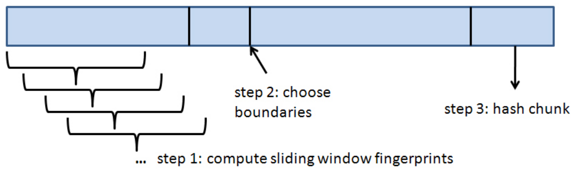
Our measurement study consists of two parts: analysis of a user's browsing trace in the hopes of capturing duplication during normal browsing activity, and analysis of a crawler-based browsing trace in order to capture duplication across a larger number and wider variety of sources. Before discussing the results of these experiments, we explain our measurement infrastructure.

3.1 Measurement Infrastructure

Our measurement infrastructure includes both measurement and analysis tools. To record browsing traces, we used a combination of WireShark for user traces and `wget` for crawler traces. We wrote several scripts to process the trace files (remove headers, combine files, split files by source IP, and split files by source name using reverse DNS lookups).

To analyze the amount of duplication in a file, we modified an existing `rabinpoly` library [2] that computes a sliding window Rabin fingerprint for a file.

Specifically, we augmented the fingerprinting functionality with code that computes fingerprints, randomly assigns chunk boundaries, and then computes an MD5 hash of the resulting chunks. Figure 1 shows this process. The reason for this seemingly complex computation is a problem identified in [5, 9, 8]: choosing chunks based on location causes a pathological case in duplication calculation. Inserting one byte will shift the chunks following it, causing their hashes to be different, and thus finding no duplication where in fact there was a large amount. The key to solving this problem is value-based chunks, and thus we choose chunk boundaries based on fingerprint values. In particular, we use a paramater k to manipulate the probability that a given fingerprint designates a chunk boundary. A fingerprint value determines a chunk boundary if and only if its last k digits are 0. We explore the effect of the choice of k in greater detail in our experiments below. Finally, after choosing chunk boundaries, we hash chunks using an MD5 hashing function, which creates 128 bit hash values. By comparing the hash values of each chunk in a trace, our tool computes an overall duplication percentage (number of duplicated bytes over total number of bytes).


step 2: choose boundaries
step 3: hash chunk
... step 1: compute sliding window fingerprints
Figure 1: The chunk generation process.

3.2 User Browsing Trace

We analyzed a user's browsing trace that took place for several hours on the evening of April 20, 2010, using Firefox with caching enabled and a non-empty cache. This allowed us to consider duplication not captured by browser caching.

A number of variables must be considered, including:

1. **Sliding Window Size.** One variable to consider is the sliding window size for the fingerprint calculation. We determined experimentally (as did [5]) that this variable has little effect of the percentage duplication found. Figure 2 shows the duplication percentage for the user browsing trace with all variables fixed but k (see below) and window size, which ranges from 16 to 128 bytes. While we show only this graph here, we performed the same analysis for all other traces and variable combinations and found the same result. Thus, from this point forward, we use a window size of 64 unless otherwise stated.

2. **Probability of Chunk Boundary.** The variable k corresponds to the probability that a fingerprint value is chosen as a chunk boundary. As discussed

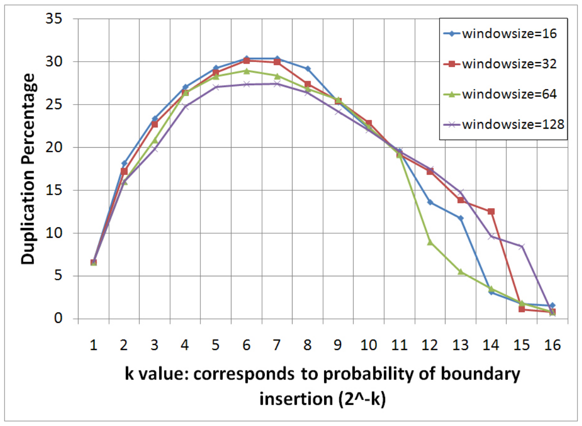


Figure 2: The percentage duplication from a user browsing trace, across varying values of k (1 to 16) and varying sliding window sizes (16 to 128 bytes), with a fixed minimum chunk size of 128.

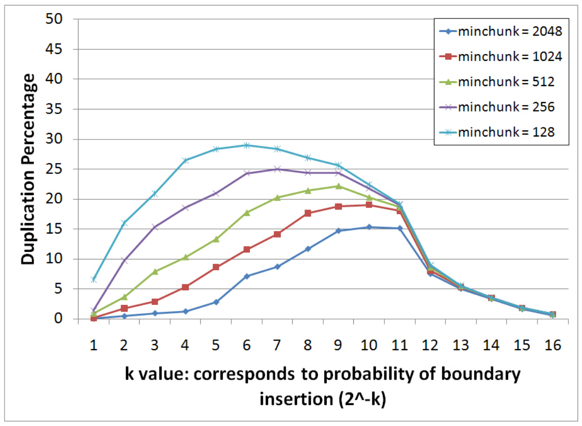


Figure 3: The percentage duplication from a user browsing trace, across varying values of k (1 to 16) and varying minimum chunk sizes (128 to 2048 bytes).

above, we designate a chunk boundary when the last k digits in the fingerprint value are 0. Thus, the probability of choosing a chunk boundary is 2^{-k}, for an expected chunk size of 2^k. As chunk boundaries are more likely, we expect to get smaller chunks, and thus find more duplication (since the comparisons are more granular), and vice versa. We discuss the effect of k further below.

3. **Minimum Chunk Size.** To prevent the bias of trivial duplication (such as on a character level), we enforce a minimum chunk size. We discuss experimentation with this further below.

4. **Maximum Chunk Size.** To avoid pathological cases in which chunks are too large (i.e. an entire file), we also enforce a maximum chunk size. We found this variable to be less important, as no chunks hit the maximum size in any of our experiments unless we made it artifically small. Thus, from this point forward, we use a maximum chunk size of 64 KB (65536 bytes), similar to [5].

3.2.1 Duplication in the entire trace

Figure 3 shows the duplication percentage in the user browsing trace when considering all data in the trace simultaneously. In other words, this data shows the duplication percentage across all sources, i.e. data from one website that is duplicated on another website is included in the percentage.

From this data, we see that for each minimum chunk size (each line in the graph), there is a different optimal k value in terms of maximum amount of duplication found. This optimal k value shifts downward as the minimum chunk size decreases. The reason for this is that, given no minimum chunk size, the smaller the k value, the smaller the average chunk will be (since chunk boundaries are more likely); smaller chunks result in more duplication found. Additional measurement results (not shown) show up to over 99% duplication found for $k = 1$ and no minimum chunk size, simply because the chunks are then as small as individual characters, which are naturally repeated. Imposing a minimum chunk size prevents this less useful duplication, but may thereby create unnatural chunk boundaries for smaller k values. The result of this are the curves seen in Figure 3, where smaller minimum chunk sizes have lesser effect.

3.2.2 Duplication split by source

Figure 4 shows the duplication percentage when the user browsing trace is split by source. In other words, this data includes only duplication that was found on the same website, not across websites. We calculated this information to determine whether or not there is interesting duplication among sources, not merely among HTTP data from the same source.

Unfortunately, the result was somewhat negative. While there is some duplication between sources—the difference between the two graphs is about 5% at the peak—the amount was limited. (This confirms that the results in [9], which found that 78% of duplication was among data from the same server.) An analysis of what this duplication actually contained revealed common Javascript code shared among websites. The code was mainly for tracking purposes: Google Analytics, Dialogix (which tracks brand or company names in social media [1]), etc.

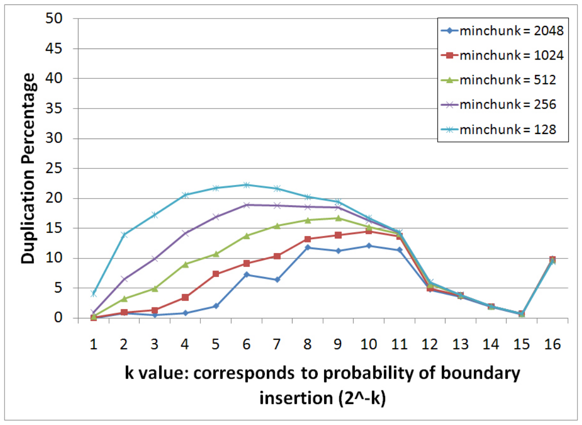


Figure 4: The percentage duplication only in data from common sources in a user browsing trace, across varying values of k (1 to 16) and varying minimum chunk sizes (128 to 2048 bytes).

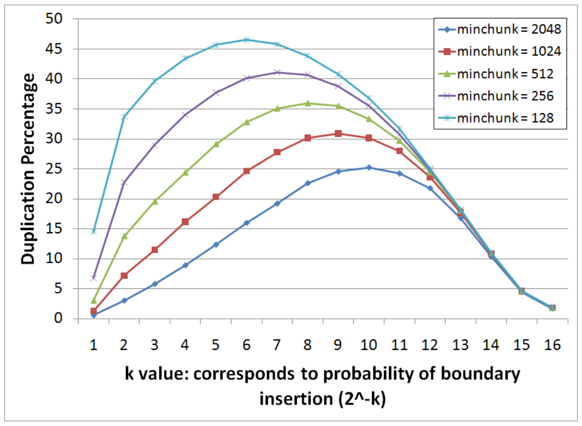


Figure 5: The percentage duplication in HTML from a crawler browsing trace, across varying values of k (1 to 16) and varying minimum chunk sizes (128 to 2048 bytes).

3.3 Crawler Browsing Trace

In this section we analyze a more comprehensive browsing trace that was gathered on May 3, 2010 using `wget`'s webcrawling functionality. The total size of the trace is about 400 MB, and we analyzed separately the duplication in the HTML files and the Javascript files. The results of these measurements are shown in Figures 5 and 6. The curves in these graphs are much smoother than those in the previous section, due to the fact that the traces are much larger, and thus any pathologies with chunk choices are hidden by the sheer amount of data.

Compared to the user browsing trace (Figure 3), the crawler traces contain up to almost 20% more duplication (the graphs are intentionally drawn to the same scale for easy visual comparison). This additional duplication is likely due to the larger amount of available data. As in the user browsing trace results, we see that smaller k values lead to more duplication found, limited by the imposed minimum chunk size.

3.3.1 Space/Performance Tradeoff

Intuitively and given the above results, smaller chunks result in more duplication found. However, smaller chunks require more storage on the client side, since each chunk comes with a 128-bit overhead (the size of the MD5 hash). Figure 7 shows the tradeoff between bytes of duplication saved and bytes of storage needed for the HTML portion of the crawler browsing trace. This graph shows the ratio of bytes of duplicated content found in the trace to bytes of storage required, across the usual suspects in minimum chunk sizes (128 to 2048 bytes). The bytes of storage required is calculated as follows: 16 bytes (128 bits) of hash value per chunk, where the number of chunks is calculated as the total file size minus the number of duplicated bytes, divided by the expected chunk size (2^k). For these (and all other reasonable) minimum chunk sizes, we find the optimal point to be at $k = 14$, or an expected chunk size of 16384 bytes.

For space reasons, we omit similar graphs for the user trace and the Javascript portion of the crawler trace. The optimal points for these traces are different—the optimal point for the user trace is at $k = 15$, and for the Javascript portion is at $k = 16$, indicating larger k values for that trace would have been even better. We expect that the optimal point is different for different traces, but in general will be above $k = 10$ for large enough traces. As in Figure 7, we found the minimum chunk size to have little effect (and thus graphed only a subset of the minimum chunk sizes we actually tested).

3.4 Comparing with gzip

While our deduplication technique is orthogonal to compression, we still find it important to compare the potential savings from value-based deduplication with those of simple compression. As a pessimistic comparison, we simply compressed (using gzip) each trace file and compared the resulting savings in file size with the potential savings from our technique, indicated by duplication percentage in the graphs previously discussed. The results of this comparison can be found in Table 1. The results are disappointing for value-based deduplication, as its potential savings are only a fraction of those that compression might achieve. Even considering that the compression savings estimate is quite pessimistic, we believe

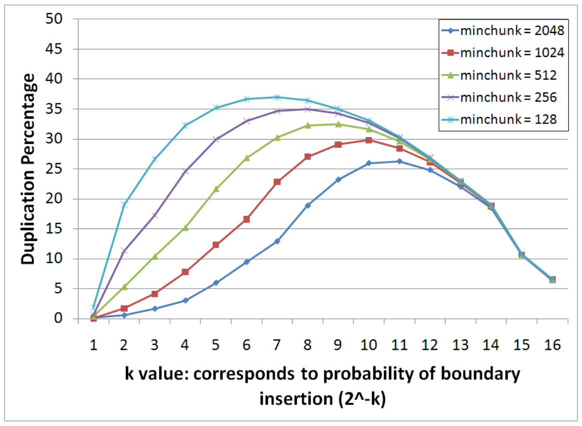


Figure 6: The percentage duplication in javascript from a crawler browsing trace, across varying values of k (1 to 16) and varying minimum chunk sizes (128 to 2048 bytes).

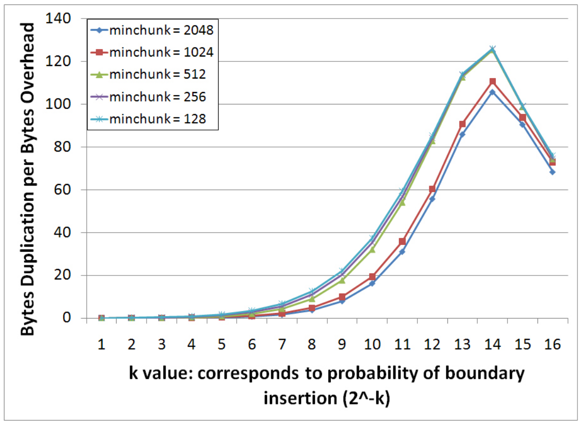


Figure 7: Bytes of duplication per bytes of storage overhead required for the HTML portion of the crawler browsing trace.

Trace	Uncompressed	Compressed	Savings
User	5.463 MB	0.879 MB	83.9%
Crawler (HTML)	339.707 MB	63.879 MB	81.2%
Crawler (JS)	46.04 MB	12.458 MB	72.9%

Table 1: Optimistic savings by compression in browsing trace files.

it is likely still better—and more importantly, easier—to compress web content than to use our deduplication technique. We suspect that many web servers still do not implement compression uniformly because it has not always been standard in browsers, but this is no longer a roadblock for compression today.

4 Implementation

Despite the somewhat negative results of our measurement analysis, we consider how our value-based deduplication mechanism may be deployed by web servers. We see a number of reasons that this may be desirable:

- It gives individual servers fine-grained control of the caching of individual page elements. We describe below a system that would allow a web server to switch to such a framework automatically.
- Leveraging new HTML5 features, this type of value-based caching can be done transparently in the browser, without reliance on any intermediate proxy caches.
- Outside the scope of this project, we envision a client-side storage based system in which different web servers can share data references without sharing data. In other words, Flickr might give Facebook a reference to one of its images already stored on a client's browser, which Facebook can then use to render that image on the client-side, without ever gaining access to the image itself.

For our implementation, we wanted to make use of the new HTML5 client-side storage feature. This idea suggests the following general data flow for a client-server HTTP deduplication system: upon receiving an HTTP request, the server responds with a bare-bones HTML page and a bit of Javascript, which we call `cache.js`. The Javascript code patches up the bare-bones webpage with actual content, using objects already stored in the cache (i.e. in HTML5 browser local storage), or making specific requests to the server if the corresponding cache content is not available. Figure 8 shows this general data flow between client and server. The data stored in the cache corresponds to (hash, value) pairs, where the hash is the MD5 hash of a chunk (as in our measurement study) or some other identifier, and the value is the corresponding chunk data that we attempt to deduplicate.

A major question regarding an implementation of this data flow is how to determine chunks for deduplication. For the purposes of this project, we thus consider two implementations. In one, we use the native structure of HTML to guide the creation of chunks for deduplication; in the other, we create chunks using the more randomized method that we used for measurement, as described previously.

One limitation of using HTML5 local storage is that it does not allow for the sharing of storage elements

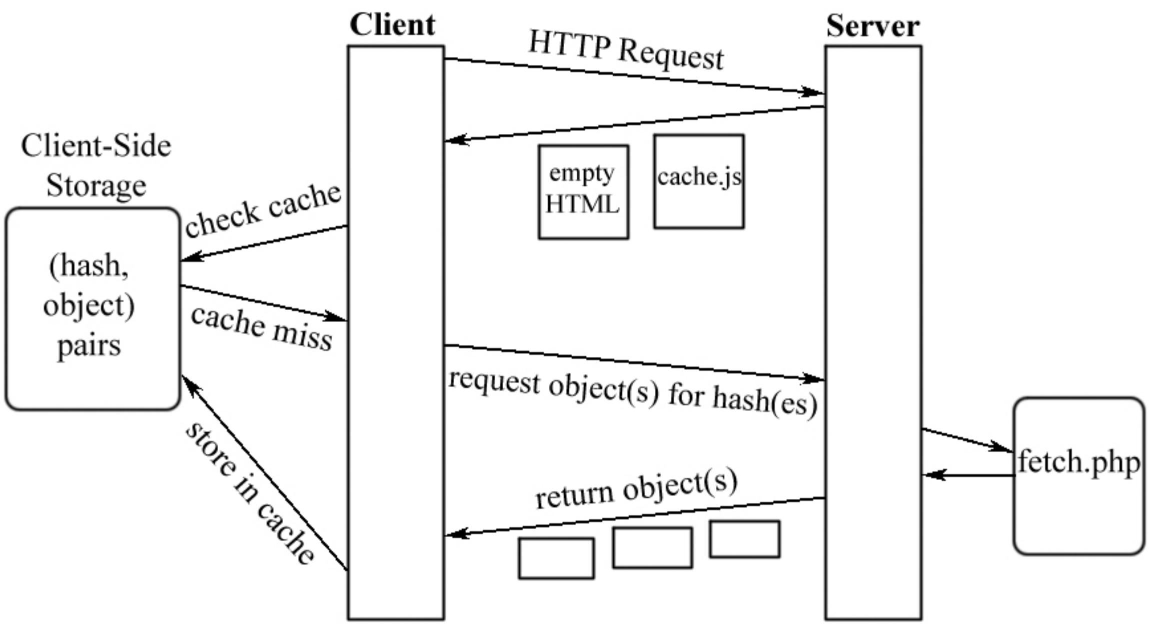


Figure 8: This figure shows the general data flow between client, server, and the client's cache for both implementations. The client constructs the final webpage in the browser using the original empty HTML page sent by the server, the cache.js Javascript file, and the data stored in the cache.

among different domains, for security reasons. This does not allow us to implement something that takes advantage of the shared duplication among different sources—admittedly limited, but potentially quite interesting in terms of the features it might support (such as the sharing of data references but not of data among sites, as mentioned above).

4.1 Implementation 1: HTML Structure

In our first implementation, we tackle the chunk determination problem by leveraging the existing structure of HTML. In other words, we use HTML elements as chunks. We use three such elements, creating chunks from data between `<div>` tags, between `<style>` tags, and between `<image>` tags.

In the first two cases (`div` and `style` elements), we consider the chunk data to be simply the text between the beginning and end tags. This includes any other nested tags. The corresponding chunk hash value is simply the MD5 hash of the chunk text content.

For `image` elements, the process is slightly more complex. Since it is likely that deduplication will be more valuable for images than plain text, we did not want to simply ignore them or use the image source text as the chunk data. Therefore, we chose to take advantage of another new HTML5 feature, the `canvas` element. Since Base64 encoded image data URLs can be extracted from `canvas` objects, we thus transform all `image` elements into `canvas` elements with the corresponding image loaded. We then extract the appropriate image data URL and consider this the chunk value (and its MD5 hash the corresponding hash value). In other words, for images, the image data URL is stored in the cache, and thus when a new request is made for that image, the `cache.js` Javascript will instead pull the image data URL from the cache and load that into the `canvas` element, rather than making another network request for the image source.

4.1.1 Server automation

In order to make this implementation plausibly usable by the general server, we built a system which transforms an existing HTML page (and corresponding resources, like images) into a deduplication-friendly system that follows the data flow shown in Figure 8. Given an existing HTML page, the system creates the following:

- A bare-bones HTML page, in which all `div` elements (at a specified level of depth) are replaced by empty `div` elements, which have an additional `hash` attribute containing the MD5 hash of the corresponding content. Similarly, `style` elements are replaced by empty ones in this fashion, and `image` elements are replaced by empty `canvas` elements with appropriate `hash` attribute values.

- A `fetch.php` script on the server-side, which simply returns the corresponding chunk data for a given hash.
- A `cache.js` script which, on the client-side, checks for objects in the cache, requests them from the server if necessary, and fills out the bare-bones HTML page for a complete page in the client's browser.

The system that we built to generate this framework makes use of Python's built-in HTML parsing class. In particular, we adapted the code in Chapter 8 of [6] to parse an existing HTML file and create the files described above.

A server could thus use this system to automatically generate a deduplication-friendly framework that does deduplication on generic chunks based on HTML's native structure. When the foundational HTML page changes, the server simply reruns the Python generation script as described above. The new bare-bones HTML page will contain different hash values for changed elements, and thus the client will request the new objects instead of finding a match in the cache.

A major limitation of this implementation currently is its extremely (noticeably) slow page load times, which we believe are due mostly to the `image` → `canvas` → data URL overhead. We also envision an optimization (which would apply also to the second implementation) that uses browser cookies to determine the first time a user visits a site, in order to short-circuit the exchange in Figure 8 when the client clearly cannot yet have any of that site's items in its cache.

4.2 Implementation 2: Random Chunking

Our second implementation leverages Rabin fingerprinting to partition HTML, Javascript and any other files sent to the client into chunks which are used to reconstruct the document. The goal of this approach is to leverage duplication across chunks of a document which do not conform to an HTML layout. For a given page the server uses the Rabin fingerprinting technique described above to determine the appropriate chunks and calculates the hash for each chunk. Once the chunks and their corresponding hashes have been computed, a page request results in a three step process, similar to Figure 8.

First, upon a client request, the server responds with all of the hashes needed to reconstruct the page, without any of the chunk data. The client then parses the list of hashes, using each hash to index into its local storage and retrieve the corresponding chunk. If the chunk is not found in local storage, the client saves the hash so that it can later retrieve the chunk from the server. Once the client has checked its local storage for all of the hashes, it requests the chunks for all of the missing hashes in bulk. This is the second step. Once the server responds with all of the missing chunks, the client has all of the information needed to reconstruct the page. The third step is to simply fetch any auxiliary documents once the file has been reconstructed on the client's end.

This scheme is accomplished via a mechanism similar to Figure 8 and the previous implementation: the server sends an empty HTML file along with Javascript files that handle requesting/collecting the chunks and writing the final document. The file `loader.js` is sent to the client along with the empty HTML document. `loader.js` initiates requesting all of the hashes that describe the page from the server, then collects all of the chunks (requesting any missing ones from the server) and finally writes the chunks into the document. The subsequent fetching of auxiliary documents is performed by the browser as normal.

One limitation of this implementation to date is that it does not yet handle the caching of images (as does the previous implementation, though problematically).

5 Discussion

In this section, we discuss first the merits of the techniques we have described in this paper, and then consider a number of security concerns that any commercial implementation of our system would need to address.

5.1 The merits of HTTP-level value-based deduplication

Based on our measurement results in Section 3, we feel there is insuffienct evidence to pursue value-based deduplication of the sort we propose in this paper for the purpose of HTTP traffic reduction. A simpler and more effective method to reduce traffic is gzip compression, which we showed has the potential to provide more savings than value-based caching. We do note that our technique is orthogonal to and composable with data compression, but believe that the additional benefit is not worth the additional implementation, deployment, and client-side storage costs. We also note here that [8] came to a similar conclusion: that there is relatively low opportunity for value-based caching over name-based caching combined with delta encoding and compression.

However, as we discussed in Section 4, we do see a number of reasons that our value-based deduplication mechanism may be valuable to web servers. These reasons include fine-grained control of caching by web servers, easy deployment and transparent execution in the browser using HTML5, and the potential for data reference sharing among servers. We thus view this project in part as a foray into a potential use of the new HTML5 client-side storage feature.

5.2 Security Concerns

We list here a number of security issues that any full-fledged implementation of our system would need to address. Being security students, we cannot help but do so. These issues include but are not limited to:

- **Side-channel attacks:** By sending a client an HTML page containing the hash of a certain object, an attacker can determine whether the client has previously requested this object during normal browsing, based on whether or not the client makes a request in response to the attacker's page, rather than pulling the object from the cache. This could allow an attacker to determine whether a user has visited a certain website or viewed certain content.
- **Client-side DOS:** An attacker might create a website that fills a browser's client-side storage to capacity, reducing performance or breaking certain features on other sites that rely on this storage.
- **Information leakage:** Another concern with HTML5 client-side storage is that it may cause sensitive information to be stored, in plain text, in the user's browser cache, allowing anyone with physical access to extract it. A server could address this by caching encrypted versions of sensitive data, although this incurs additional processing and deployment overhead.

6 Conclusion

In this project, we examined HTTP-level duplication. We first reported on our measurement study and analyzed the amount and types of duplication in the Internet today. We found that value-based deduplication can save up to 50% of traffic for large-scale web traces, though most of this duplication is among traffic from the same source. We further found, in a somewhat negative result, that gzip compression (though orthogonal to our method) would be a simpler and more effective deduplication method.

Nevertheless, we see several reasons that a server might benefit from our system, and we discussed these in Section 4. We thus implemented two versions of a simple server-client architecture that takes advantage of HTML5 client-side storage for value-based caching and deduplication. We conclude that while value-based caching is not likely worth the cost, especially compared to other deduplication mechanisms, our implementation gave us some insight into the potential for HTML5 client-side storage.

References

[1] DIALOGIX. Social media monitoring, 2010. http://www.dialogix.com.au/.

[2] KIM, H.-A. Sliding Window Based Rabin Fingerprint Computation Library (source code), Dec. 2005. http://www.cs.cmu.edu/ hakim/software/.

[3] MANBER, U. Finding similar files in a large file system. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (Berkeley, CA, USA, 1994), USENIX Association, pp. 2–2.

[4] MOGUL, J. C., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 1997), ACM, pp. 181–194.

[5] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM, pp. 174–187.

[6] PILGRIM, M. *Dive Into Python*. APress, 2004.

[7] RABIN, M. O. Fingerprinting by random polynomials. Tech. Rep. TR-15-81, Department of Computer Science, Harvard University, 1981.

[8] RHEA, S. C., LIANG, K., AND BREWER, E. Value-based web caching. In *WWW '03: Proceedings of the 12th international conference on World Wide Web* (New York, NY, USA, 2003), ACM, pp. 619–628.

[9] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. *SIGCOMM Computer Communication Review 30*, 4 (2000), 87–95.