Towards full on-line deduplication of the Web

Ricardo Filipe and João Barreto

Inesc-ID/Technical University Lisbon
[ricardo.filipe,joao.barreto]@ist.utl.pt

Abstract. The Internet is widely used nowadays but still has important limitations. While average Web page size keeps increasing, the bandwidth available to each user is not growing at the same rate. Users want to do several things at the same time on the Web and they don't want to wait much time to do it.
We notice that most Web pages have substantial redundancy with previous versions of themselves and other pages. With this in mind, we developed a novel deduplication system for HTTP traffic.
Our proposed system works from end client to server and only for text resources transmitted through the HTTP protocol. Our system transfers less bytes than a plain HTTP transfer, achieving gains of 82% when downloading pages from popular sites such as `cnn.com`. It completes a transfer request 4% faster than plain HTTP. It is also not as time consuming as delta-encoding between the requested resource and an older version of it. Our approach is the first to be implemented and analysed on a real Internet environment.

1 Introduction

The Web has had a major growth in recent years. The number of users keeps growing, the average Web page size and number of objects per page have steadily increased [1], the data available through the Internet is doubling every year [2].

This enormous growth entails many challenges. While available bandwidth has steadily grown in some network environments, it remains a scarce resource in others, such as Bluetooth networks. Even in high bandwidth scenarios, each transferred byte frequently has a cost, for instance in terms of battery consumption or fees paid to the client's ISP, which clients naturally wish to minimize.

Fortunately, the substantial redundancy that exists across the contents downloaded from the Web tells us that most of the transmitted payload could be, in fact, avoided. Most individual files exhibit intra-file redundancy, which one can eliminate with compression schemes like Gzip. Furthermore, Web browsers frequently request resources (e.g. previously downloaded HTML pages or images) that were already downloaded by the same client (or nearby clients) and have not changed meanwhile, a well studied problem that Web caches effectively tackle.

However, there is evidence that much redundancy in the Web originates from situations that neither per-file data compression nor Web caching can exploit. Such redundancy is caused by two phenomenons. A first one occurs when the

INForum 2010 - II Simpósio de Informática, Luís S. Barbosa, Miguel P. Correia (eds), 9-10 Setembro, 2010, pp. 291–302

same resource is referenced by different URLs, which occurs mostly in image resources and is commonly called aliasing [4]. Most importantly, the most common phenomenon is *resource modification* [3], when a resource changes by little pieces over time. We see this everyday in our social Web. News sites are updated several times per day, they let you comment on those news and see other people's comments. Social networks let you comment on other people's profiles. Blogs make it possible to post new content and have it commented too. Forums and discussion sites have threads of responses from their users. Hence, pages loaded at different times will often be very similar in content, with marginal changes introduced by the new pieces of information.

In order to exploit the above phenomena, we need techniques for full *deduplication* [5] of the Web. In other words, techniques that are free from the limitations of data compression and classical Web caching, thus able to eliminate any form of redundancy: be it within the individual resource, or across different resources (or across distinct versions of the same resource); going from the granularity of the individual resource down to much finer-grained chunks of redundant information.

Perhaps that older deduplication technique that breaks the above mentioned limitations is delta encoding [3]. In delta encoding, a client that already holds some version of a given resource and requests a newer version will only receive a compact set of differences from the latter to the former, which the client should then apply upon its local (older) version. Delta-encoding is still very much in use, mostly when the deltas can be precomputed, which is often the case when distributing software patches and service packs. However, since the algorithms used for computing deltas are typically very time-expensive [3], delta encoding is not suitable for distributing dynamic content like most Web pages that are created on-the-fly.

More recently, much research has proposed techniques for on-line deduplication of dynamic Web content. However, the proposed techniques exhibit important drawbacks that severely limit their usefulness in the large-scale Web. Some assume strong synchronization between client and server state [6], while the memory requirements of others do not scale well to large-scale Web servers (e.g. [9], [8]). Some are only able to detect redundancy across the contents transmitted from the ISP proxy to its clients, neglecting the proxy-server path [11]. More importantly, most solutions have never been implemented nor evaluated experimentally (e.g. [6], [8]).

In this paper we present a novel deduplication system we called dedupHTTP. The proposed system works from the end client to the origin server. It detects redundancy in text resources transferred through the HTTP protocol. It uses the client's cached files from the requested resource's domain as reference resources. It eliminates redundant transfers originating from resource modification and aliasing of resources inside each domain. The current implementation is browser and server transparent and has a very lightweight communication protocol. Our results show that dedupHTTP:

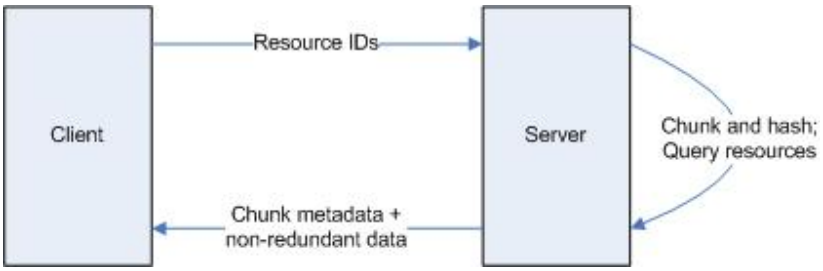

Fig. 1. Architecture Overview

- Outperforms plain HTTP transfer in the number of bytes transferred by 82% and time consumed by 4%, at large bandwidth;
- Has 10% better time performance than delta-encoding with a previous version of the requested resource;

The rest of the paper is structured as follows. First we will explain the resource division algorithm into chunks. Then we will overview the architechture of the system and the communication protocol. We then show the results of our tests, comparing our system with regular transfers, gzip compression and a simple delta-encoding system for Web transfer. Finally we go through some of the related work and draw our conclusions to this work.

2 Architecture overview

The dedupHTTP system has two implementation points, we'll call them client and server for simplicity. The client stores the resources and maps them by host domain. When it makes a request for a resource the client fetches all the identifiers from stored resources of the same host domain as the requested resource. These identifiers represent the reference resources for the request.

When the server responds to a request it divides the response into chunks using the algorithm from Section 4. Those chunks' hashes are stored on the server along with their offset within the resource and their length (16 bytes metadata per chunk on the current implementation). The server then retrieves the client resource identifiers from the HTTP header. The server searches for the response resource's chunks in the identified client resources, and the metadata required for resource reconstruction is sent to the client.

When the client receives the response it starts reconstructing the requested resource by fetching the referenced chunks from local cache. Then it combines them with the non-redundant content on the response in the correct order. The resource is saved on the client and mapped to its host domain.

3 Protocol

When a new request is done by the client, the reference resources identifiers are serialized into a byte array and placed inside a new HTTP header called "Versions", which goes on the request header. This accomplishes two important goals. First there is no need for the server to keep client based state, as it is each client's responsibility to provide the right list of reference resources. Secondly, it allows the client cache to have the regular resource cache eviction policies, the resources the client tells the server are in its cache are the ones the server will use as reference resources. [1]

After dividing the resource, the corresponding chunk meta information is stored on the server, mapping the resource. The server goes through all chunks' hashes of the requested resource. For each such chunk the server queries each of the client's reference resources for the chunk's existence. If the chunk is not found the server also queries the current response's respective resource. This way we not only detect redundant chunks between different resources but also inside the resource being served. Whenever a chunk is found on a queried resource we only need to tell the client where to find it. If the chunk is not found in any of the queried resources, we copy the chunk data to the final response.

The final response begins with a metadata section. The size of this section is stored on a new HTTP header we called "Metadata". In this section goes all the information needed for the client to find the redundant chunks in its cache. Each redundant chunk is here identified by a four-part tuple: the offset in the current response where the chunk is to be appended, the resource identifier where the client can find the chunk content, the offset of the resource where the chunk content can be found and the length of the chunk. At the end of the metadata block we append the non-redundant response content.

When the client receives the response it only has to go through the metadata, copy to the final response the chunks referenced in the metadata, from the locally cached resources, and copy the non-redundant content from the received response to the final response, in the respective order. Thus, the new resource is reconstructed on the client and the client does not even have to store meta information about the chunks, only the resource identifier and the resource itself.

Each response has an identifier created on the server, which is sent in the metadata block. We have opted to store this value in 2 bytes only, which helps shorten the metadata block. This means there can only be 65536 different resources referenced in the server. We believe this is more than enough for a regular server, even with dynamic resources generated on-the-fly, since after some days or weeks the older resources metadata can be evicted. Even if the client's cache could accommodate for resources that were older than that, the usefulness of such resources is expectably low. Redundancy with fresh downloaded contents

[1] As we show next, each client will be handling look-ups to its list of reference resources. At the same time, it can try to evict some of such entries. We can easily ensure safe synchronization of such accesses by using regular read-write locks.

should mostly come from recent versions, rather than older ones, which we intend to explore in future work.

For this we use an age value on each resource. When the age value for the resource has been reached, if the client still has the resource cached it is evicted from the client's cache. At the same time the server will evict the chunk metadata it possesses for that resource. This way the client and server caches are kept synchronized without additional network messages.

The other pieces of the metadata are all stored in 4 byte integers which makes for a 14 bytes total of metadata to reference a chunk. This number should be taken into account when choosing the chunk size, since there will be no gain if there can be chunks smaller than this.

3.1 Optimizations

We have seen there are consecutive chunks detected in the same resource many times. Since we check the client's resources in the same order for every chunk, we can save on metadata by regarding the sequence of consecutive chunks as a single chunk. Hence, we only send a meta-data block for the large coalesced chunk (instead of one block per consecutive redundant chunk). A particular case where such optimization is very effective is when a requested resource is aliased on the domain; the only thing that will be sent to the client is a chunk's worth of metadata with the whole resource length.

The same resource can be requested by different clients. On the first request the algorithm will be the one explained in this section. For the second and subsequent client requests we can save an important step of processing, the chunk division algorithm.

The server stores all served request's ETags. These are unique resource identifiers present in the HTTP specification. When a new request is to be served we check if the ETag has already passed before. If it has, we retrieve the corresponding resource chunk metadata and continue to the next phase. Only the first request to each resource has to go through the chunk division phase.

4 Chunk division algorithm

The first step to take on the server when a resource is requested from a client is to divide that resource into indexed chunks. This way we can easily identify which parts of the resource the client already has and which are new and have to be sent.

We start by creating per-byte hashes of the resource content. These hashes represent smaller chunks of the resource. This is done by using the rolling hash function of Karp-Rabin fingerprinting [13]. Let c_i be the byte at index i of the resource stream and k be the length of the Karp-Rabin chunk. Let b be a prime number as base. The hash of the Karp-Rabin chunk is given by:

$$H(c_i...c_{i+k}) = c_i * b^{k-1} + c_{i+1} * b^{k-2} + ... + c_{k-1} * b + c_k$$

Since b is a constant, the iteratibility of this function allows us to calculate the next byte's hash with some simple operations on the previous byte's hash with the new byte:

$$H(c_{i+1}...c_{i+1+k}) = ((H(c_i...c_{i+k}) - c_i * b^k) + c_{k+1}) * b$$

Now we have to select the hashes that represent the resource. We could select all hashes, but that is unfeasible memory-wise since we have an hash per byte of content. So, we select certain hashes to be the boundaries of larger chunks of data.

For this step we chose the Winnowing technique since it has been experimentally proven to give the better redundancy detection [7]. We enforce a minimum and maximum chunk size because these content based methods can create too big or too small chunks compared to the desired size.

After selecting the chunk boundaries we hash each larger chunk using a 64 bit MurmurHash [15]. This choice was driven by the fact that MurmurHash outperforms cryptographic hash functions such as MD5 or SHA1, while retaining a very low collision rate. We do not need cryptographic security in our hashes, so using MD5 or SHA1 would be overkill. They are much slower and offer similar collision rates (in fact MurmurHash offers better hash distribution than MD5 and very similar to SHA1 [15]).

5 Implementation

Our system was implemented with a proxy for the client machine and a reverse proxy for the server machine. Both proxies were implemented using the OWASP Proxy [16] library, which takes care of every aspect in HTTP communication. Each proxy was extended with the proper functionalities described in the algorithms and architecture to deal with the resource content. This was done in about five hundred lines of Java code, since that was the language of OWASP Proxy. The MurmurHash code was used from a public library [17] that ported the hash code to Java.

Using proxies for the implementation allows us to be browser and server independent, not having to familiarize with a unique browser and server implementation, nor favor one over another. It has the inconvenients of adding proxy latency to the connection and having it's own cache on the client side. The proxy cache and browser cache will certainly overlap in most of their data. On the server side this is not an issue since we only store resource metadata.

The resource metadata is stored per chunk on the server. It is composed of two 32 bit integers, offset in the resource content and chunk length, and a 64 bit long for the chunk hash.

6 Evaluation

The test server machine is an Intel Pentium 4 @ 3.20 GHz with 2 GB of RAM, while the client machine is an Intel Core 2 Duo @ 2.16 GHz with 4 GB of RAM.

Number of files	Total Size	Server memory overhead
337	41,5 MB	16 bytes * Number of Chunks

Table 1. Workload specification

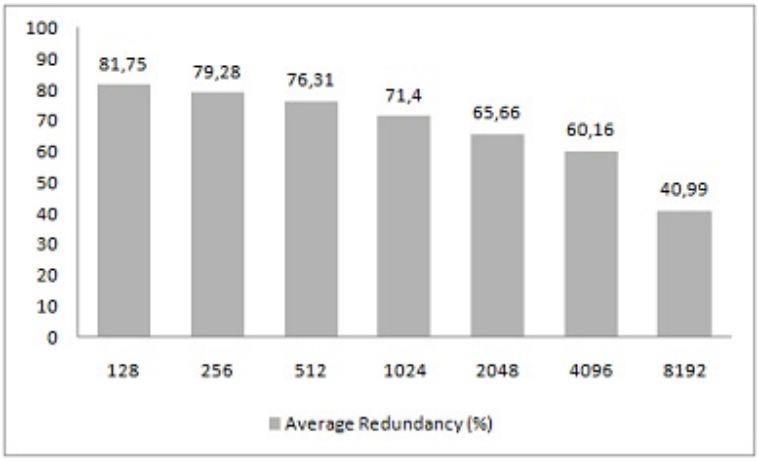

Fig. 2. Experiment average redundancy for several chunk sizes

The workload for our experiments was created by downloading every news and comments page from www.cnn.com (Table 1). These pages change very frequently which makes them a perfect fit for a system like this. We did this in two consecutive days so that we could compare the redundancy that remains from one day to the other, mimicing a regular user that likes to read the news every day. We removed unchanged pages from the second day, since they would be treated by a regular browser cache.

Our chunk division algorithm has two parameters that should be experimented for the best redundancy detection: the smaller Karp-Rabin fingerprint length (Table 2) and the larger chunk length (Figure 2).

The large chunks were experimented with sizes ranging from 128 bytes up to 8192 bytes (8 KB). The smaller the chunk size the more metadata the server has to store, since we have more chunks per file. Since our metadata is not too heavy we decided the 128 bytes chunks should be used for the rest of the experiments. The increase in number of chunks to search for could factor in the choice of chunk size. Although, this has not been noticeable in our experiments, so we disregard it.

We have experimented the Karp-Rabin fingerprint sizes of 16, 32 and 48 bytes. The 16 bytes fingerprints give a better redundancy detection, although by a small margin, so we decide to use it for the remainder of the experiments. This parameter has no influence in the number of chunks nor in the complexity of the division algorithm, so the value with better results should be chosen.

Size of Karp-Rabin fingerprint	Redundancy detected
16	81,75%
32	80,64%
48	80,74%

Table 2. Redundancy detected with varying Karp-Rabin fingerprint size

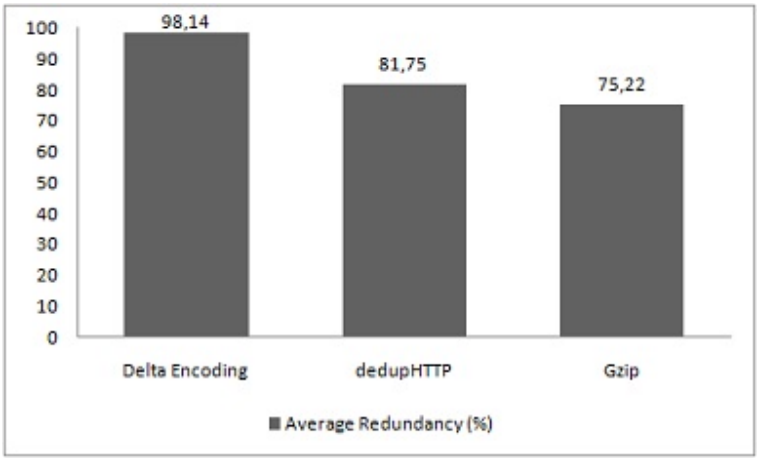

Fig. 3. Experiment average redundancy for several solutions

We compared our solution with three others. The first one is plain HTTP transfer, then compressing the resources with Gzip and finally we tested against a delta encoding solution we created.

This delta encoding system stores the resources on the server and client. If the client is downloading a resource that already exists in its cache, but is outdated, the server creates a delta between both versions of the resource on-the-fly. Then it sends the delta to the client and it reconstructs the requested resource. This solution does not work for the first request of a resource, so we have not included those results in our measures.

We want to see how does dedupHTTP compare to the other solutions in redundancy detected and Time to Display (Figures 3 and 4). Time to Display is the time it takes since the resource is requested in the client until it is displayed on the client's machine. If the deduplication algorithm takes more processing time than that which is spared by less transfer traffic, the Time to Display will worsen which is undesirable. Our Time To Display results are the mean average time of downloading all of the workload's files one by one.

Delta encoding detects most existing redundancy that we could hope to detect, since it is a lossless algorithm that works with only previous versions of the same resource. This should be the comparison point of our solution as far as redundancy detection goes. On the other hand it is a very costly algorithm, which brings very high Time to Display. Plain HTTP transfer should be the ref-

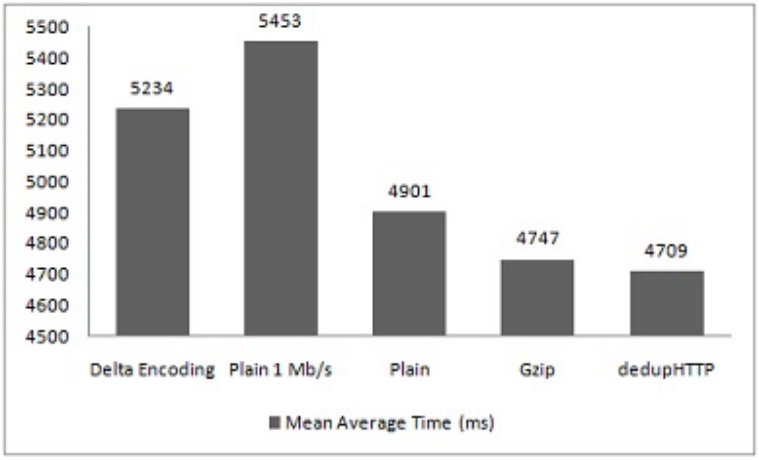

Fig. 4. Experiment average Time to Display for several solutions

erence point here, since we want to improve on the every day solution's Time to Display. We have also compared to plain HTTP transfer on limited bandwidth (1 Mb/s) to assert that where fewer bandwidth is available our system would be most benefitial.

As far as redundancy detection is concerned, we outperform Gzip by a meaningful margin. Comparing to delta encoding we are a bit far from detecting as much redundancy, although we didn't isolate only the responses that could be delta-encoded to get dedupHTTP results. If we did that our redundancy detection is much closer to delta encoding (95%) and plus we still detect much redundancy on the first time resource requests, where delta-encoding cannot be used.

Regarding Time to Display we outperform every single solution we have tested against, achieving comparable results to Gzip encoding. This is very significant as we prove our system's feasibility to complement regular Web caching. We can also confirm that using delta encoding on-the-fly is much more computationaly heavy than our system.

7 Related work

Chunk fingerprinting has been widely used in distributed file systems [10] and other deduplication systems for the Web (e.g., [11]). For chunk boundary selection these systems used the scheme proposed by Manber [12], but as Anand et al. [7] have shown the Winnowing [14] technique is more profitable in this regard.

Manber's fingerprint selection scheme selects each hash that complies with $H \bmod P = 0$ where P is the desired larger chunk size. On average they expect to have an hash selected in P bytes as a chunk boundary. Winnowing differs by selecting the smaller hash in each consecutive window of P bytes.

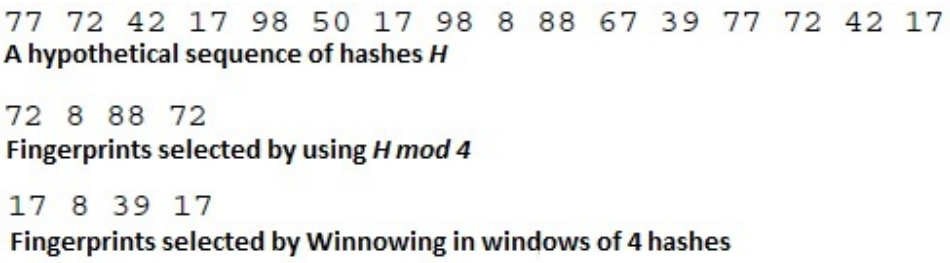

Fig. 5. Manber vs. Winnowing: chunk boundary selection

There have been many systems proposed for Web deduplication, each with its own limitations. Spring and Wetherall [6] proposed a shared cache architecture where two caches store chunks of data in the same way. When redundant content is identified by the transmitting cache only the corresponding chunks fingerprints are sent. They didn't implement this architecture, it was used as an example of where to use their redundancy detection algorithm. They also do not address how to keep both caches synchronized. They suggest this approach for a protocol independent system, but in their results we can see that only HTTP is useful to analyse since it is the most used protocol and with most redundant data too.

Anand et al. [7] have recently confirmed these results for an enterprise setting (on a university setting P2P traffic is a bit more relevant). They also suggest that an end-to-end approach is preferable to a middlebox one since most users do not share the same surfing habbits, therefore the gain of detecting cross-user redundancy would be little when compared to the cost of detecting redundancy across more data. We applied these studies results to devise dedupHTTP.

Chan and Woo [8] presented a general approach to cache based compaction. It includes two main ideas: a selection algorithm to choose reference objects, and an encoding/decoding algorithm that acts upon a new object using the selected reference objects. They proposed a dictionary based solution for encoding/decoding. When there are no reference resources it would act as gzip and when there are some it would get redundant strings from them. Their approach is not scalable, if the number of reference objects increases the complexity of the algorithm increases at the same time. While our approach also increases in complexity with the increase of reference objects, it is much less noticeable. They have also not implemented and tested their proposal. Their selection algorithm though gave interesting results. They tell us that resource redundancy is not limited to previous versions of that resource. Most resources in the same folder and even in the same domain still have much redundancy to be taken advantage of as reference resources for deduplication.

Banga et al. [9] developed an optimistic deltas system. The server stores the resources it sends to the clients. When a new version of a resource is requested it creates a delta between the new version and the version the client has and sends it. When a resource is to be sent for the first time to a client, an older version

is immediately sent when the request is received on the server. Then if the new resource is the same as the sent one the only thing necessary is to respond to the client saying that. If it is different a delta is created between the two versions and sent. The system is optimistic since it expects to have enough idle time to send the old version of the resource while the new version is being created. It also assumes the changes between the two versions are small relatively to the whole document. With the bandwidths of today this kind of latency reduction would be inefficient, more often than not overlapping the transport of the older resource with the response with the new one.

Rhea et al. developed a Value Based Web Cache (VBWC) [11]. It is a system similar to Spring and Wetherall's but they implemented and tested it. They aimed to use it at the ISP proxy, where it would store the chunk's hashes that went to the clients. The proxy did not store the actual data. When a chunk goes to a client, the proxy cache checks if that client already has that chunk. If it does it only transfers its fingerprint. The bandwidth savings of VBWC are only on the ISP proxy-client connection. There is no cross-client redundancy detection since their hash cache is mapped by client. Since each client also stores the chunk's hashes they are redundant in the ISP proxy for all clients that requested the same chunks. All of these problems could be solved if the hash cache was implemented at the origin server. Our approach is an example of how this can be implemented.

8 Future work and Conclusions

This work presents evidence that suggests that current HTTP transfer systems can be much improved with simple changes on both client and server side and fosters further research on this topic, which has been stagnant for quite some time. Even if bandwidth is ever less a concern for desktop users, we must remind ourselves we live in a wireless world where bandwidth is still a scarce resource.

We have devised a new on-line deduplication system for the Web. We have shown that it outperforms the solutions available to current Web servers, including plain HTTP transfer and Gzip compression. We are the first to implement and test such a system in real internet conditions.

There are several items we can work further in our system. We plan to test how does redundancy between two versions vary over time, instead of just two days test up to thirty days difference. This will help us determine at what point in time we should evict resource metadata on the server and client.

We want to test an hybrid algorithm that uses our system and then compresses the responses with Gzip. It should yeld even better redundancy detection, although it may cost more Time to Display than affordable.

We plan to test the system in a controlled network environment, with the client and server nodes interconnected by a network link with tunable bandwidth and latency, in order to avoid external arbitrary interferences on those parameters.

We also plan to run more detailed benchmark tests, which will allow us to understand the factors contributing to the latency of our solution; namely, time

spent on Web server CPU processing, message handling, message transmission, among others. From these results, we should be able to further optimize our solution and evaluate its true scalability.

References

1. http://www.websiteoptimization.com/speed/tweak/average-web-page/
2. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html
3. Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In Proc. of ACM SIGCOMM, 1997
4. Terence Kelly , Jeffrey Mogul, Aliasing on the world wide web: prevalence and performance implications, Proceedings of the 11th international conference on World Wide Web, May 07-11, 2002, Honolulu, Hawaii, USA
5. Nagapramod Mandagere , Pin Zhou , Mark A Smith , Sandeep Uttamchandani, Demystifying data deduplication, Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion, December 01-05, 2008, Leuven, Belgium
6. Neil T. Spring , David Wetherall, A protocol-independent technique for eliminating redundant network traffic, Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, p.87-95, August 28-September 01, 2000, Stockholm, Sweden
7. Ashok Anand , Chitra Muthukrishnan , Aditya Akella , Ramachandran Ramjee, Redundancy in network traffic: findings and implications, Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, June 15-19, 2009, Seattle, WA, USA
8. Mun Choon Chan and Thomas Y. C. Woo, Cache-based compaction: A new technique for optimizing web transfer, Proceedings of IEEE INFOCOM, March 1999
9. Gaurav Banga , Fred Douglis , Michael Rabinovich, Optimistic deltas for WWW latency reduction, Proceedings of the USENIX Annual Technical Conference, p.22-22, January 06-10, 1997, Anaheim, California
10. Athicha Muthitacharoen , Benjie Chen , David Mazires, A low-bandwidth network file system, Proceedings of the eighteenth ACM symposium on Operating systems principles, October 21-24, 2001, Banff, Alberta, Canada
11. Sean C. Rhea , Kevin Liang , Eric Brewer, Value-based web caching, Proceedings of the 12th international conference on World Wide Web, May 20-24, 2003, Budapest, Hungary
12. Udi Manber. Finding similar files in a large file system. In Proceedings of the USENIX Winter 1994 Technical Conference, pages 110, San Fransisco, CA, USA, 1721 1994
13. Karp, R. M. and Rabin, M. O. 1987. Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. 31, 2 (Mar. 1987), 249-260
14. Saul Schleimer , Daniel S. Wilkerson , Alex Aiken, Winnowing: local algorithms for document fingerprinting, Proceedings of the 2003 ACM SIGMOD international conference on Management of data, June 09-12, 2003, San Diego, California
15. A. Appleby, MurmurHash 2.0, http://sites.google.com/site/murmurhash/, 2009
16. R. Dawes, OWASP Proxy, http://www.owasp.org/index.php/Category:OWASP_Proxy, 2010
17. V. Holub, Java implementation of MurmurHash, http://d3s.mff.cuni.cz/ holub/sw/javamurmurhash