Improving the Efficiency of HTTP Caching by Hash Based Resource Identifiers

Chris Drechsler and Thomas Bauschert

TU Chemnitz, Germany
{chris.drechsler,thomas.bauschert}@etit.tu-chemnitz

Abstract. Internet traffic is continuously growing and contributes substantially to rising costs for network operators. Evaluations have shown that today multimedia content accounts for a major part of the transferred bytes in the Internet and that HTTP is the dominant protocol. A natural solution for reducing these network costs is caching of frequently requested content. Already in the beginning of the 90s HTTP caches have been proposed, which were deployed in the domains of the network operators. These traditional HTTP caches rely on URLs to identify resources and to avoid transferring the same data twice. Unfortunately today a specific content might be available under different URLs. Furthermore many HTTP connections are personalized and therefore caching is often disabled by content producers. So traditional HTTP caching became inefficient for the network operators. In this paper we propose a method to improve the efficiency of HTTP caching. Our approach is based mainly on hash keys as additional identifiers in the header of HTTP messages. By that identification of the transferred content is more precise than with URLs. Beside this we show how caching can be achieved even in the presence of personalization in HTTP messages and how content producers remain full control over their content although it is cached by ISPs.

Keywords: HTTP caching, caching efficiency, network optimization.

1 Introduction

The data traffic in the Internet is continuously growing and challenges the network operators. According to Cisco [1] the traffic in the Internet doubles about every three years until 2015.

An analysis of several Internet studies [1], [2], [3], [4] shows two main trends: a) Multimedia content (like videos, photos, music) accounts with a growing tendency already today for a major part of the transferred bytes in the Internet. b) The dominant protocol wrt. data volume is HTTP and it accounts for more than 50% of the traffic in the Internet [5], [6].

A natural solution for operators to cope with the Internet traffic growth is caching of frequently retrieved resources. Indeed, a large part of the content (especially multimedia content) is static and therefore there is a large potential for caching. A recent study has shown that nearly 68% of bytes transferred via

R. Szabó and A. Vidács (Eds.): EUNICE 2012, LNCS 7479, pp. 362–371, 2012.

HTTP are cacheable [5]. Thus, operators started to establish HTTP caches (also known as web caches or proxy caches) in their networks in the beginning of the 90s. More recently Content Delivery Networks (CDNs) were established in the Internet to perform a replication of specific resources in order to make them available locally [5], [6].

The efficiency of classical HTTP caches is drastically reduced after CDNs had been widely established. A main requirement for high efficiency of traditional HTTP caches is that a resource should be available only under one URL. The load balancing and the localization of content over several servers within a CDN leads often to different URLs for one specific resource. Therefore classical HTTP caches are not able to identify them as a single resource and thus store several copies of the same resource. Further reasons for the decreased efficiency are the personalization of HTTP messages (e. g. via cookies) and the explicit suppression of caching by content producers [6]. In the US network operators are not running HTTP caches any more because the bandwidth savings are not big enough to justify the costs of the caches [5].

In this paper we propose a concept for increasing the efficiency of HTTP caches in the presence of CDNs. Our approach is based on putting additional information in the header of an HTTP message to uniquely identify the resource independently of its URL and storage location. Furthermore we present a solution on how personalization of HTTP message exchange can be performed despite caching.

This paper is organized as follows: In section 2 we provide a short overview on caching mechanisms in the Internet and where caching is done today. In section 3 it is outlined which problems traditional HTTP caches encounter. Section 4 lists key requirements for new solutions in the field of HTTP caching. After an overview about related work on caching improvements in section 5, our approach is presented in section 6. In section 7 we evaluate of our approach. The paper concludes with a discussion of the advantages of our solution in section 8.

2 Classification of Caching Methods

Caching methods can be classified according to client-side and server-side approaches [7]. In both cases a copy of the respective identified resource is kept in the cache. However, in the client-side approach caching is in the responsibility of the content consumers, whereas in the server-side approach the content producers perform the caching and thus have full control over the caching of their resources [7].

HTTP caches (also denoted as forward caches, web caches or proxy caches) and web-browser caches belong to the client-side caching schemes. Here caching is in the responsibility of the ISPs and end-users, respectively. Server-side caching schemes are better known under the term replication, see [7]. Here identical copies of a resource are distributed over different servers in the Internet. CDNs and so called reverse caches belong to this type of caching.

3 Problem Statement

HTTP caches have a significant potential for reducing Interdomain traffic. Recent investigations show very promising results regarding the cacheability of HTTP traffic. Erman et al. [5] analyzed Internet traffic traces and found that 92% of all requests and 68% of all transferred bytes can be cached. Similar results are presented by Ager et al. [6]. They stated that 71% of all requests and 28% of all transferred bytes can be cached.

Despite these promising results most large ISPs in the US do not run HTTP caches anymore [5]. To understand the reasons, we have to look at the functionality of classical HTTP caches and investigate how they are influenced by CDNs and content producers.

A HTTP cache tries to cache (all cacheable) resources which are requested by the clients [7]. Important caching criteria are the following:

- The URL serves as an unique identifier for the HTTP cache. With the URL the cache can identify the resources and all incoming requests are compared to the URLs of stored objects in the cache. If there is a cache hit, then the cache sends the right response to the client as a representative of the origin server. Otherwise (cache miss) the cache forwards the request to the origin server [7].
- Special information in the header (like Cache-Control, Expires, Last-Modified, ...) of a HTTP message indicate whether this message can be cached or must not be stored. Every cache must accept these information – if they are missing caching is not possible [7].

A limitation of HTTP caches is the physical storage volume. Therefore not all cacheable resources that are requested by the clients can be stored in the cache. Because of that HTTP caches are running replacement algorithms (like Least Recently Used or Least Frequently Used) to replace resources by new ones [7].

In contrast to that, replication mechanisms are storing only specific resources on different servers in the Internet. Those servers are also working as a large distributed cache (reverse cache), which is under the control of the content producer [7]. The copies of a specific resource on the various servers are often available under different URLs like in CDNs [6], [8].

The actual efficiency of existing HTTP caches in ISP networks is very low. According to the results of Ager et al. [6], only 16% of all requests and 9% of all transferred bytes can be delivered by an HTTP cache. In contrast to that, 92% of all requests and 68% of all transferred bytes would be cacheable [5]. The reasons for the bad performance of HTTP caches are as follows:

- As explained, in CDNs one specific resource might be available under different URLs. If one client retrieves this resource under URL1 and another client in the same network retrieves it under URL2 then from the perspective of a HTTP cache these requests address two different resources. At the moment

there is no way for HTTP caches to detect that these different URLs are related to the same resource and therefore the cache stores several identical copies. Because of the limited storage volume and the above-mentioned replacement strategies the cache replaces other resources and the efficiency drops.
- Many HTTP request (method GET) as well as response messages are personalized by mechanisms that are related to the header of an HTTP message (like cookies and parameters in the query string). A cache does not store such requested resources although the transferred HTTP body of these HTTP response messages may be always the same and just the HTTP header differs. Personalization is a main reason for non-cacheable resources [6].
- Some content producers explicitly prohibit the caching of their content by putting special information in the header of an HTTP message (Cache-Control, Pragma, Expires,) that every cache has to accept. One reason could be that they want to log all requests belonging to a specific resource for statistical purposes. Other content providers never consider the advantages of caching and so their default is not to cache at all.
- Sometimes the necessary information (Cache-Control, Expires, ...) are missing in the HTTP header. Therefore they cannot be cached. According to the low frequency of such requests, the results in [5] show only a 4% potential for caching.
- Even if one resource is accessed via one URL by different clients, the status code in the HTTP response message can differ and therefore caching of the requested resource is difficult. This may occur when only pieces of a resource are transferred (Partial Content, Delta Encoding) or the resource is temporarily available under another URL (Redirection). In most of the cases an actual cache is not able to detect this [5].
- In some cases identical content is distributed on different websites and different webspaces in the Internet. Especially software repositories or ISO images are available under different URLs. A cache cannot detect this.

Due to the low caching efficiency of current HTTP caches most of the large ISPs in the US do not operate HTTP caches in their domains as the bandwidth savings obviously can not justify the costs for the server hardware on which the cache service is running. According to [4] an operator pays for transit traffic about 12 $/Mbps in 2008 and 1,20 $/Mbps in 2014. Thus, from an economic point of view there would be a huge potential in HTTP caching (even in 2014) if the caching efficiency can be improved.

4 Requirements for Improved Caching

In the following we list the key requirements that have to be fulfilled for increasing the efficiency of HTTP caching.

- Detect duplicate transfers in HTTP. In HTTP resources are identified by URLs and identical copies of one specific resource can be available under different URLs. Moreover one specific resource can exist in different versions and in different representations which will be transferred to the client based on the content negotiation mechanism (described in the HTTP/1.1 specification in RFC 2616).
- Come up with both, caching and personalization. A large amount of HTTP messages are personalized via additional information in the HTTP header. The bodies of several HTTP messages which are related to one specific resource remain mostly unchanged. Caching in this context is often not possible or is explicitly disabled by the content producers. One reason is that (in case of a cache hit) the cache does not establish a connection between the server and the client and so the personalized data in the header of an HTTP message can not be transferred between them (and therefore caching is omitted by the content producers).
- Do not disable caching for obtaining statistical data. Some content producers disable caching as they want to log every request for their hosted resources in order to gain statistical data (e.g. on how many requests occur per hour, or from which region in the world the requests are coming, or which browser do my customer use, etc) or to measure QoS.

Thus new solutions have to be found to detect duplicate transfers as well as to allow message personalization and obtaining statistical data. Furthermore the new solution should be simple and easy to implement.

5 Related Work

Mogul et al. describe in [9] a method to detect and avoid duplicate payloads in HTTP messages which is mainly based on RFC 3230. They introduce a so called instance digest which better describes the transferred content. The instance digest is a hash key computed over the resource (or to be more precise over the concrete representation of the relevant resource in the terminology of the HTTP/1.1 specification in RFC 2616). This key is added to the header of the HTTP message and serves as a unique identifier of the transferred content in the body of the message. Unfortunately the described method is quite complex and does not deal with personalized HTTP transfers. However it is a base for our approach, described in section 6.

A similar approach to describe the transferred content in HTTP messages more precisely is presented by Bahn et al. in [10]. They use the Content-MD5 header field, which was specified in RFC 1864 and was adopted by the HTTP/1.1 specification in RFC 2616. Unfortunately, according to RFC 1864 and 2616, the Content-MD5 is completely optional and the computation of the MD5 value is done right before the message is transferred to the client. This requires additional processing power at web servers. Because of that the Content-MD5 header field did not succeed in general and the approach of Bahn et al did not become accepted.

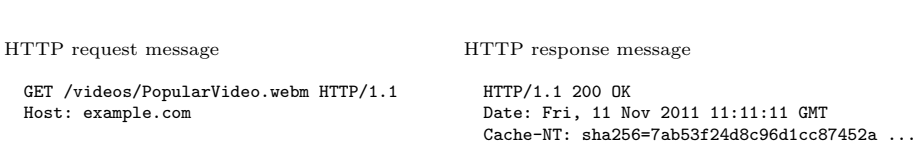

Fig. 1. Example of how a hash key of the transferred content is added to the header of an HTTP response message

6 New Hash Based Caching Method

Our approach is based on two key concepts: a) a special HTTP header field is introduced to precisely identify the transferred content. This header field acts as an one-to-one identifier for our smart HTTP cache. b) the cache operation is modified, e.g. the cache does not abort the connection between client and server. In the following we explain these concepts in more detail.

6.1 HTTP Header Field Extension

Similar to [9] in our solution a hash key for identifying the HTTP content is used. The hash function should meet the following requirements: the hash value should be easily computable for any given HTTP content and for a given hash value it should be infeasible to generate the resource that fits to this hash value. Furthermore it should be infeasible to find two different resources with the same hash value (collision resistance). These requirements are similar to those of cryptographic hash functions. We propose to use the Secure Hash Algorithm with 256 bit length (SHA256), as it is widely used and actually no security flaws have been identified.

The hash key is computed over the specific representation of a requested resource and is added to the header of a HTTP message as depicted in figure 1. The left side of figure 1 shows the request by the client and the right side the corresponding response by the server. Note, that the hash value not necessarily needs to be computed over the body of a HTTP message. This has the following background: in some cases only pieces of the requested resource respectively representation are transferred (partial content, delta encoding, ...) and for that it is important to identify the resource itself instead of identifying just the partial content in the body of the HTTP message. With that the HTTP cache can identify the resource itself and can deliver the requested pieces of the resource.

With the hash key as identifier every bit identical content which is transferred over HTTP can be identified and delivered by a cache. Even bit identical content which is unintentionally available under different URLs (like videos and pictures in social networks or video portals) can be detected, cached and transferred by the cache.

For precisely identifiying the content it is required, that the hash value is computed and added to the HTTP message header by the corresponding HTTP server. Thus the major challenge for a practical realization of our approach is to convince the content producers to add this piece of information to the HTTP headers.

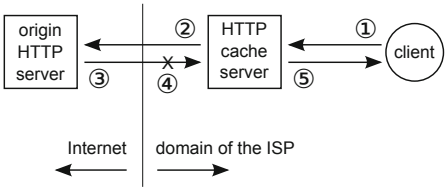

Fig. 2. Modified HTTP cache operation

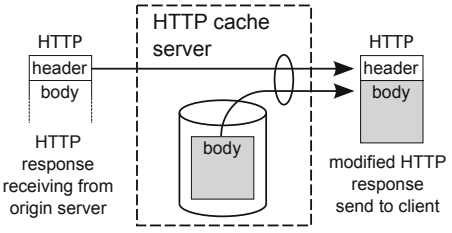

Fig. 3. HTTP response (in case of valid cache entry)

6.2 Modified HTTP Cache Operation

In our solution the HTTP cache operates in a modified way that differs from classical HTTP caches. The transfer of personalized information is possible even if the requested content is delivered by the cache.

As shown in figure 2 we assume a transparent cache which accepts the request of the client (1) and forwards it to the corresponding server (2). The response is coming from the origin server and passes the cache (3). Assuming the server has added the above-mentioned SHA256 hash key to the HTTP header, two situations may arise:

- The HTTP cache does not have a valid cache entry. In this case it simply forwards the response to the client. Additionally a copy of the transferred content is stored. For the purpose of consistency the HTTP cache may compute the hash key over the local copy and compare it to the hash value in the HTTP header.
- The HTTP cache has a valid cache entry. In this case it extracts the header out of the incoming HTTP response message and aborts the connection to the origin server (4). Afterwards it concatenates the HTTP header (coming from the origin server) with the locally stored HTTP body of the requested content and sends it to the client (5), see figures 2 and 3.

With this modification of HTTP cache operation personalized information in the HTTP header is always transferred between the origin server and the client even if the real content is coming from the cache. Content producers keep full control over their resources (as if there is no resource on the origin server also no transfer of cached content happens) and can log all requests for statistical purpose.

This approach also faces some challenges. In case of a cache hit the HTTP cache server aborts the connection to the origin server and sends a copy of the requested resource from its local storage to the client. For large-sized content (like videos) an abort is always reasonable. However for small-sized content the whole content already might have been transferred from the origin server before the abort message reaches the origin server. We evaluate this in detail in section 7.

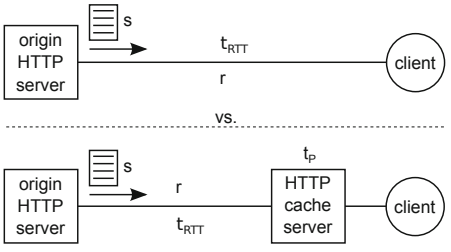

Fig. 4. Scenario without and with a HTTP cache server between client and origin server

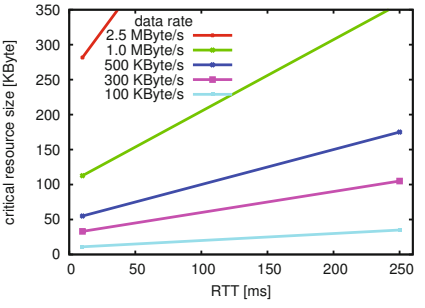

Fig. 5. Critical Resource size vs. RTT and data rate

Moreover further challenges arise in the case of consistent connections (HTTP header field Connection: keep-alive) and request pipelining. As there is no way on the level of HTTP to stop the transfer of a requested resource, the HTTP cache server has to abort the TCP connection. By this all following HTTP response messages in the pipeline (initiated by the pipelined requests of the client) will also be aborted. We propose the following solution to cope with this challenge: the client establishes a TCP connection to the HTTP cache server and performs request pipelining as usual. For every request in that pipeline the HTTP cache server then establishes a new TCP connection to the origin server. All incoming HTTP response messages are rearranged in the right order (as they were requested by the client) and are sent to the client.

7 Evaluation of the Critical Resource Size

The critical resource size is the content size threshold for which it is worth to abort the HTTP transfer of the origin server and to send a local copy of the requested content to the client. In the upper part of figure 4 a scenario without a HTTP cache server is shown. The client sends the HTTP request message and receives the HTTP response message directly from the origin HTTP server. Thus, the transfer time is determined by dividing the size s of the resource by the average data rate r of the connection.

In the lower part of figure 4 a scenario is shown where a HTTP cache server is located between origin server and client. Analyzing the header of the receiving HTTP response message and aborting the HTTP transfer with the origin server in case of a cache hit consumes some time. During this time the origin HTTP server still sends the resource. Thus the critical resource size is calculated so that the response time of the HTTP cache server is less than the time required for the transfer of the whole resource by the origin HTTP server. According to equation 1 the response time is composed of two different components: the processing time of the HTTP cache server and the time for aborting the HTTP transfer with the origin HTTP server.

$$s \geq r \cdot (\underbrace{t_P}_{\substack{\text{processing speed} \\ \text{of the HTTP} \\ \text{cache server}}} + \underbrace{t_{RTT}}_{\substack{\text{time to abort the TCP connection} \\ \text{between origin server and HTTP cache} \\ \text{server and during data is still on the line}}}) \quad (1)$$

The processing time of the HTTP cache server is the time for analyzing the incoming HTTP response message from the origin server and concatenating the HTTP header from the origin HTTP server with the HTTP body of the locally stored copy. The time required for aborting the HTTP transfer with the origin HTTP server can be approximated by the RTT between HTTP cache server and origin HTTP server.

For giving a coarse grained estimation about the critical resource size s we make some assumptions. At first we assume a maximum transfer rate of 2.5 MByte/s – the data rate required by HD videos (H.264, 1080i, main profile). To obtain an overview of common RTTs in the Internet we measured the RTTs (seen from our campus network) of popular websites. These websites were taken from the Alexa top 500 index [11]. The result of our RTT measurement is shown in figure 6. Most RTTs are in the range up to 200 ms. Regarding the average response time of an HTTP cache server we take the results obtained by Lee et al. [13] for the proxy cache server Squid. Lee et al. measured about 100 ms response time of the proxy cache server for a request rate of about 150 requests per second.

Figure 5 shows the critical resource size s vs. the RTT and the data rate a resource is send out from the origin server. It can be seen that the critical resource size s increases with the data rate and the RTT. Even for an average data rate of only 100 KByte/s s is at least 11 KByte (for 10 ms RTT) or 35 KByte (for 250 ms RTT). For a high speed connection with a data rate of 2.5 MByte/s s should be at least 282 KByte.

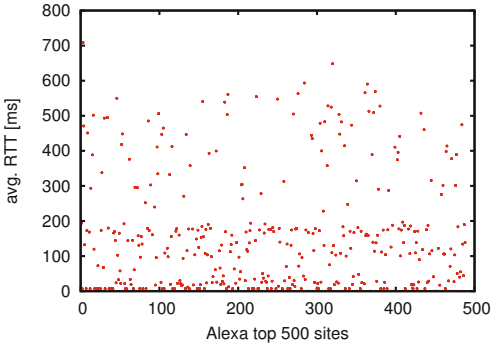

Fig. 6. RTT measurement results

8 Conclusion and Future Work

The potential for caching of HTTP traffic is very high but remains unused today. In this paper we present an approach for improved HTTP caching in order to leverage this potential. The two key concepts for this are a special HTTP header field for more precisely identifying the contents transferred over HTTP and a modified cache operation. Both ISPs (via transit traffic and cost reduction) as well as consumers (via QoE improvement) might profit from our solution. Currently we are implementing our approach in a demo setup to analyze how it will scale on real servers. We want to find out how much additional processing power the HTTP cache server requires to scan incoming HTTP response messages for the new header field and to organize the cache. As future work we plan to investigate how such a HTTP cache can be realized as a distributed and cooperative cache with support of the clients and what are the feasibility constraints for such a cooperative caching approach.

References

1. Cisco Systems Inc., Entering the Zettabyte Era, http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/VNI_Hyperconnectivity_WP.pdf
2. Sandvine Inc., Global Internet Phenomena Report, http://www.sandvine.com/general/document.download.asp?docID=40&sourceID=0S
3. Ipoque, Internet Study 2008/2009, http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf
4. Labovitz, C.: ATLAS Internet Observatory 2009 Annual Report (2009), http://www.nanog.org/meetings/nanog47/presentations/Monday/Labovitz_ObserveReport_N47_Mon.pdf
5. Erman, J., Gerber, A., Hajiaghayi, M., Pei, D., Spatscheck, O.: Network-aware forward caching. In: 18th International Conference on World Wide Web (WWW 2009), pp. 291–300. ACM, New York (2009)
6. Ager, B., Schneider, F., Juhoon, K., Feldmann, A.: Revisiting Cacheability in Times of User Generated Content. In: INFOCOM IEEE Conference on Computer Communications Workshops, pp. 1–6. IEEE Press, New York (2010)
7. Rabinovich, M., Spatschek, O.: Web Caching and Replication. Addison-Wesley, Boston (2002)
8. Su, A., Choffnes, D., Kuzmanovic, A., Bustamante, F.: Drafting behind Akamai (travelocity-based detouring). In: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2006), pp. 435–446. ACM, New York (2006)
9. Mogul, J., Chan, Y., Kelly, T.: Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP. In: 1st Symposium on Networked Systems Design and Implementation. USENIX Association, Berkeley (2004)
10. Bahn, H., Lee, H., Noh, S., Min, S., Koh, K.: Replica-aware caching for Web proxies. Computer Communications 25(3), 183–188 (2002)
11. Alexa Top 500 index, http://www.alexa.com/topsites
12. Alexa Top 500 sites in the US, http://www.alexa.com/topsites/countries/US
13. Lee, D., Kim, K.J.: A Study on Improving Web Cache Server Performance Using Delayed Caching. In: 2010 International Conference on Information Science and Applications (ICISA), pp. 1–5. IEEE Press, New York (2010)