Data Deduplication in Web Prefetching Systems

Pedro Jorge do Nascimento Neves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Pedro Manuel Moreira Vaz Antunes de Sousa
Supervisor: Prof. João Pedro Faria Mendonça Barreto
Supervisor: Prof. Paulo Jorge Pires Ferreira
Members of the Committee: Prof. Pável Pereira Calado

May 2013

Agradecimentos

Em primeiro lugar quero agradecer ao meu Orientador, Prof. João Barreto, toda a ajuda na execução desta tese. Nas inúmeras reuniões, na orientação quanto ao rumo a seguir no projecto e nas revisões desta dissertação, esteve sempre disponível para me ajudar e foi de uma enorme paciência! O meu sincero Obrigado.

Quero também agradecer ao meu co-Orientador, Prof. Paulo Ferreira, por toda a ajuda e disponibilidade na execução desta tese.

A todos os colegas que ao longo do tempo me apoiaram deixo também o meu agradecimento.

à Céu e à Sofia

Resumo

O aumento continuado do número de utilizadores e a crescente complexidade do conteúdo Web levaram a uma degradação da latência percebida pelos utilizadores na Web. Web caching, prefetching e deduplicação são técnicas que foram utilizadas para mitigar este efeito. Independentemente da largura de banda disponível para o tráfego na rede, o Web prefetching tem capacidade para consumir a largura de banda livre na totalidade. A deduplicação explora a redundância nos dados para reduzir a quantidade de dados transferida pela rede, libertando largura de banda ocupada. Assim, é expectável que, pela combinação destas duas técnicas, seja possível reduzir significativamente a quantidade de bytes transmitida por cada pedido na Web. É também legítimo esperar que a esta redução corresponda uma melhoria da latência percebida pelos utilizadores na Web. No presente trabalho, desenvolvemos e implementámos o primeiro sistema que, tanto quanto sabemos, combina simultaneamente técnicas de Web prefetching e deduplicação. O principal objectivo deste sistema é melhorar a latência percebida pelos utilizadores na Web, relativamente a sistemas de topo actuais. Os resultados levam-nos a concluir que a aplicação da combinação das técnicas de prefetching e deduplicação à navegação Web carece de cuidado. Quando o mais crítico são as poupanças na quantidade de bytes transferidos, se é tolerável uma pequena degradação da latência, então combinar ambas as técnicas é uma boa opção. No entanto, se o mais crítico é minimizar a latência a qualquer preço, e um aumento na quantidade de bytes transmitida não constitui problema, então a melhor opção é utilizar somente prefetching.

A seguinte publicação descreve parcialmente o trabalho apresentado nesta dissertação:

P. Neves, J. Barreto, P. Ferreira, *Data Deduplication in Web Prefetching Systems*, INForum 2012, Monte de Caparica, Portugal, Setembro 2012 (poster)

Palavras-chave: Web Prefetching, Deduplicação, Detecção de redundância, Tráfego HTTP, Latência

Abstract

The continued rise in internet users and the ever-greater complexity of Web content led to some degradation in user-perceived latency in the Web. Web caching, prefetching and data deduplication are techniques that were used to mitigate this effect. Regardless of the amount of bandwidth available for network traffic, Web prefetching has the potential to consume free bandwidth to its limits. Deduplication explores data redundancies to reduce the amount of data transferred through the network, thereby freeing occupied bandwidth. Therefore, by combining these two techniques, one can expect that it will be possible to significantly reduce the amount of bytes transmitted per each Web request. It is also legitimate to expect that this reduction will correspondingly improve the user-perceived latency in the Web. In the present work, we developed and implemented the first system that, to the best of our knowledge, combines the use of Web prefetching and deduplication techniques. The main objective of this system is to improve user-perceived latency in the Web, relative to current best-of-class systems. The results lead us to conclude that care is needed when applying both prefetching and deduplication to web navigation. When savings in the amount of transferred bytes are most critical, if a small degradation on latency is tolerable, it is a good option to combine both techniques. If however, the most critical is to minimize latency at all costs, and an increase in the amount of bytes transmitted is not an issue, then the best option is to use just prefetching.

The following peer-reviewed publication partially describes the work presented in this dissertation:

P. Neves, J. Barreto, P. Ferreira, *Data Deduplication in Web Prefetching Systems*, INForum 2012, Monte de Caparica, Portugal, Setembro 2012 (poster)

Keywords: Web Prefetching, Deduplication, Redundancy detection, HTTP traffic, Latency

Contents

List of Figures

List of Tables

1 Introduction

The generalized dissemination of the Internet and corresponding growth of the World Wide Web have led to a continued rise in the number of connected users and, consequently, to a dramatic increase in global traffic over the years. In parallel, static Web pages, typical of the first years of the internet, have gradually been replaced by increasingly complex pages, filled with rich-media content, with ever-increasing size [1]. As a consequence, at a given point in time the quality of service and, in particular, the user-perceived latency have experienced some degradation. Nevertheless, the improvements in available bandwidth observed in recent years have reduced the impact of network load in user-perceived latency.

Techniques to improve perceived system performance attempt to explore the locality of reference properties that are observed in Web resources' access patterns [2]: temporal (eg. Web caching [3, 4]), geographical (eg. replication, as used in CDN's [5, 6]) and spatial locality (eg. Web prefetching [7, 8]).

In Web caching, the perceived latency is reduced by storing recently referenced resources closer to the user (either directly at the client or at an associated proxy), in anticipation of future accesses to that information. This technique has been used extensively, with considerable success [15]. However, its impact in latency is limited by the achievable hit rates, that is: regardless of the amount of available space, cache misses will always occur [16]. There can be misses for cached resources due to inconsistencies between the resource versions present in the client cache and in the origin server; misses due to new, previously uncached resources being transferred to the client; or it can even happen that the resources to be transferred are uncacheable (e.g. in the case of dynamic resources). These cache misses are not avoidable even with the most recent dynamic caching strategies [9, 10].

Web prefetching attempts to overcome the above-mentioned limitations by proactively fetching resources without waiting for user requests. Preloading web resources into local caches has shown to have an impact in reducing response times, by significantly improving hit rates [8]. Recent client access history is typically used to predict which resources are likely to be requested in the near future. Due to their speculative nature, prefetch predictions are subject to failure. If the prediction is not accurate, cache pollution, bandwidth wasting and overloading of original servers are bound to occur. Therefore, this technique demands careful application – e.g. during user idle times [11] – otherwise it can significantly degrade performance (even when using broadband), thereby defeating its original purpose. Web prefetching has been already the subject of some research [8, 12, 13]. However, the use of Web prefetching in commercial products is still marginal [14], mainly due to its potential impact on bandwidth.

Deduplication is a technique that reduces overall data footprint through the detection and elimination of redundancies. The specificity of the types of resources transferred on the Web and the increasing dynamicity of Web resources pose additional problems relative to the application of deduplication in other domains. Nevertheless, several attempts made to use data deduplication in the Web have already shown promising results [17, 18, 19, 112]. These works show that data deduplication considerably reduces the overall amount of transmitted bytes and, thereby, also the user-perceived response time. Data deduplication leverages on the redundancy present in the data transferred on the

Web. Some of that redundancy is already eliminated through caching and compression techniques. However, these techniques cannot detect all forms of cross-file and cross-version redundancy [20, 21].

As seen above, independently of the amount of available bandwidth, Web prefetching is able to take bandwidth occupancy to its limits. On the other hand, data deduplication is a technique that allows reducing the volume of transferred data through the network, thereby freeing occupied bandwidth. Therefore, it is reasonable to expect that the combined use of both techniques could potentially bring significant overall improvements to the amount of transmitted bytes per Web request. It is also legitimate to expect that this reduction can have a corresponding effect in the user-perceived latency in the Web.

In the present work we design and implement the first system that, to the best of our knowledge, combines the use of Web prefetching and data deduplication techniques with the aim of improving user-perceived latency in Web navigation, relative to present day standards.

We used as starting point an existing Web prefetching simulator framework [111] and extended it, in order to implement a deduplication technique based on a recent state-of-the-art deduplication system [112]. To this end, we introduced additional functional modules in the system that provide deduplication processing capabilities and also the necessary data and communication infrastructure needed to support the new functionalities and allow the new modules to interface with the existing prefetching framework.

The results show that the combination of prefetching and deduplication techniques can be a two-edged sword and, therefore, should be applied with care in web navigation. If reducing the total amount of transferred bytes is the most important aspect to consider, and a small impact on the latency can be tolerable by the end user, then it is a good option to combine both techniques. If however, the only concern is to minimize latency at all costs, the amount of bytes transmitted being a secondary issue, then the best option is to use just prefetching.

The remainder of this dissertation is structured as follows. In section 2 we present the most relevant related work in the areas of caching, prefetching and deduplication techniques applied to the Web. Section 3 describes the architecture of the proposed system. In section 4 we present and discuss the results of our tests. Finally, in section 5 we summarize our main contributions and present the conclusions to this work. We also suggest some future research possibilities.

2 Related Work

In this survey of related work, we start by describing the characteristics of a generic Web architecture and the overall mechanics of Web requests. We then discuss in detail several techniques currently used to optimize perceived system performance in the Web, namely: caching, prefetching and data deduplication.

2.1 The Web: Architecture and Request Mechanics

A generic web architecture is composed of two main elements: i) user agents, or clients, *i.e.*, the software employed by users to access the Web, and ii) web servers, which contain the information that users request. This generic web architecture is an example of the client-server paradigm: in order to retrieve a particular Web resource, the client attempts to communicate over the Internet to the origin Web server. For a given Uniform Resource Identifier (URI), a client can retrieve the corresponding content by establishing a connection with the server and making a request. To connect to the server the client first needs the address of the host: it queries the domain name system (DNS) to translate the hostname to its Internet Protocol (IP) address. After this, the client can establish a connection to the server. Once the Web server has received and examined the client's request, it can generate and transmit the response. Finally, after receiving the server's response, the client renders the whole page and displays it to the user (Fig. 1).

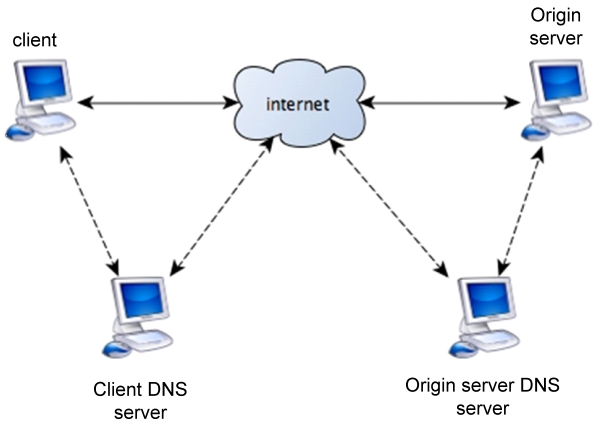
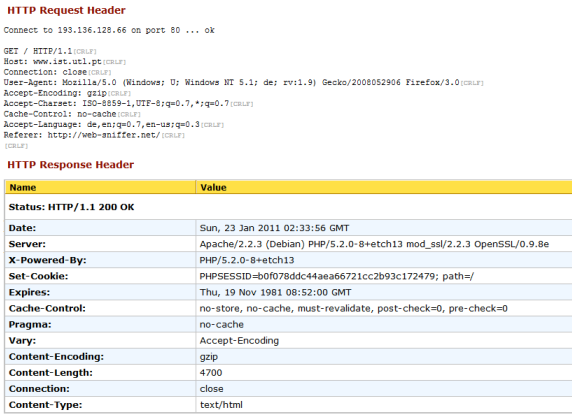


Fig. 1- Generic web architecture- web request mechanics.

Optionally, there may be more elements between clients and servers, *e.g.*, a proxy is usually located near a group of clients to cache the most popular resources accessed by that group. In this way, the user-perceived latency and the network traffic between the proxy and the servers can be reduced. Surrogates (reverse proxies) are proxies located at the server side. They cache the most popular server responses and are usually transparent to the clients, which access the surrogate as if they accessed the original web servers.

The HyperText Transfer Protocol (HTTP) [22] specifies the interaction between Web clients, servers, and eventual intermediary elements. Each request and response include headers and

possibly content (*e.g.*, data of the fields of a submitted form or the HTML of the requested page). The headers of an HTTP transaction, among other things, specify aspects relevant to the cacheability of a resource, such as the resource expiration date (Fig. 2).



HTTP Request Header

```
Connect to 193.136.128.66 on port 80 ... ok

GET / HTTP/1.1[CRLF]
Host: www.ist.utl.pt[CRLF]
Connection: close[CRLF]
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.9) Gecko/2008052906 Firefox/3.0[CRLF]
Accept-Encoding: gzip[CRLF]
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7[CRLF]
Cache-Control: no-cache[CRLF]
Accept-Language: de,en;q=0.7,en-us;q=0.3[CRLF]
Referer: http://web-sniffer.net/[CRLF]
[CRLF]
```

HTTP Response Header

Name	Value
Status: HTTP/1.1 200 OK	
Date:	Sun, 23 Jan 2011 02:33:56 GMT
Server:	Apache/2.2.3 (Debian) PHP/5.2.0-8+etch13 mod_ssl/2.2.3 OpenSSL/0.9.8e
X-Powered-By:	PHP/5.2.0-8+etch13
Set-Cookie:	PHPSESSID=b0f078ddc44aea66721cc2b93c172479; path=/
Expires:	Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control:	no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma:	no-cache
Vary:	Accept-Encoding
Content-Encoding:	gzip
Content-Length:	4700
Connection:	close
Content-Type:	text/html

Fig. 2- Example of HTTP request and response headers.

2.2 Caching

The concept of caching is similar to that of a small notebook where we keep the most frequently used telephone numbers, and that we always carry in our pocket: in this way considerable time is saved because every time one of those numbers is needed we do not need to retrieve it from the phone book, placed in the bookshelf across the room. Rather, the information is immediately available.

2.2.1 Caching in Computing Systems

Caches have been widely used in many areas of computing, implemented both in hardware (*e.g.* CPU's, hard drives) and software (*e.g.* Web browsers, page memory caches), with considerable success. A cache is an area that temporarily stores data, so that future requests for that data can be served faster than they would if they'd have to be retrieved from their original location.

Caches leverage on the fact that, independently of the application, typical data access patterns exhibit locality of reference. Two major types of locality of reference are temporal locality and spatial locality. If at one point in time a particular piece of data is referenced, then it is likely that the same data will be referenced again in the near future. This is temporal locality. Spatial locality, on the other hand, refers to the fact that if a given piece of data is referenced at a particular time, then it is likely that other data, stored physically nearby, will be referenced in the near future.

A cache might store values that have been computed earlier, or duplicates of original values that are stored elsewhere. If requested data is contained in the cache, *i.e.* a cache *hit*, the request can be served by simply reading from the cache. Otherwise – a cache *miss* – the data has to be recomputed

or fetched from its original storage location. Hence, the more requests can be served from the cache, the faster overall system performance will be.

A well-established and successful use of caching is in memory architectures [23, 24, 25, 26]. Modern computers CPU's operate at very high clock frequencies. Typical memory systems are unable to keep pace with these speeds. If the CPU needs to access data in memory it must therefore wait for the memory to provide the required information. If this would always happen, the CPU would effectively be operating at a much slower speed. To help minimize this effect, one or more levels of cache are used. These are small amounts of very fast memory, that operates at speeds identical, or close to, the speed of the CPU. In this way, when the information the CPU needs is already in the cache, slowdowns are avoided.

In case a cache miss occurs, the requested resource must be retrieved from main memory and then placed in the cache, so that the next time that resource is needed it can be accessed quickly. The usefulness of loading the resource in cache is based on the assumption, as mentioned above, of temporal locality. Due to the assumption of spatial locality, memory systems usually retrieve also multiple consecutive memory addresses and place them in the cache in a single operation, rather than just one resource at a time.

A cache is composed of a pool of entries. Each entry contains a piece of data, corresponding to a block of consecutive memory addresses, and a tag, which identifies the data in the original location (of which the cache entry is a copy). All cache entries have the same, fixed size.

At some point in time, the cache will become full. In order to define which items will be removed from the cache to make room for new ones, a cache *eviction policy* is used. Several policies have been proposed and implemented, including variations of first-in/first-out (FIFO), least recently used (LRU), least frequently used (LFU), or random policies. The purpose of these algorithms is to optimize the cache performance, *i.e.* to maximize the probability of a cache hit.

If the data in the original storage location is changed, the copy in the cache may become out-of-date, *i.e. stale*. In these cases, consistency mechanisms are necessary to ensure the information in the cache is kept up-to-date.

2.2.2 Web Caching

Web caching can reduce bandwidth usage, decrease user-perceived latencies, and reduce Web server loads. This is generally done in a way that is transparent to the end user. In this section, we discuss Web caching, its benefits and drawbacks, the main implementation architectures, and to which extent some types of Web content may or may not be cacheable.

Like in other applications of caching, a Web cache stores Web resources that are expected to be requested again (Fig. 3). However, there are some specificities to caching performed in the Web: i) the resources stored by Web caches have variable size; ii) retrieval costs are non-uniform; iii) some Web resources cannot or should not be cached.

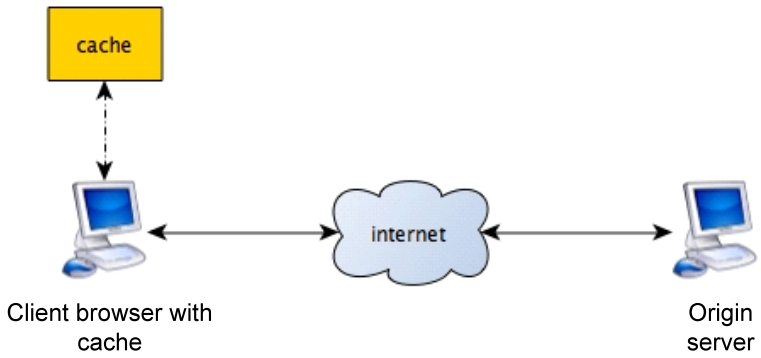


Fig. 3- Web caching.

A consequence of the heterogeneous resource size is that traditional replacement algorithms, such as the ones used in memory systems, often assume the use of a fixed resource size, so variable sizes may affect their efficiency [113]. Additionally, non-uniform resource size contributes to non-uniform retrieval costs: Web resources that are larger are likely to have a higher cost (that is, higher response times) for retrieval. Even for equally sized resources, the retrieval cost may vary as a result of distance traveled, network congestion, or server load. A consequence of this is that, in measurements of cache performance, one tracks both the overall resource hit rate, *i.e.* the percentage of requests served from cache, as well as the overall byte hit rate, *i.e.* the percentage of bytes served from cache. Finally, some Web resources cannot or should not be cached, *e.g.* because the resource is personalized to a particular client or is constantly updated, as is the case of dynamic pages.

2.2.3 Benefits of Web Caching

The main benefits of Web caching are: it can reduce network bandwidth usage, therefore allowing more efficient usage of network resources, which in the end can save money for both clients (consumers of content) and servers (content creators). Caching can also reduce user-perceived latency. Finally, it can reduce loads on the origin servers. This last effect may not be relevant to end users, but it can allow significant savings in hardware and support costs for content providers as well as a shorter response time for non-cached resources.

Web caching saves time and bandwidth because, when a request for a resource is satisfied by a cache, the content no longer has to travel across the Internet from the origin Web server to the cache. This saves bandwidth for both the cache owner and the origin server.

TCP, the network protocol used by HTTP, has a considerable overhead for connection establishment and initially sends data at a slow pace. Also, most requests on the Web are for relatively small resources. The combination of these factors produces a negative impact on performance.

Connections that can handle multiple HTTP requests are termed *long-lived* connections. If long-lived connections are used, so that each one will handle multiple requests, a significant positive effect

in client performance will result [27, 28]. In particular, for clients of forward proxies, the client can retain a long-lived connection to the proxy, instead of taking the time to establish new connections with each origin server that a user might visit during a session. The impact of using long-lived connections is mostly felt in the cases of clients with high network latency [29, 30]. Also, if proxies make use of long-lived connections to send requests from multiple clients to the same server, this allows them to reduce the time to establish connections to servers. Therefore, a proxy cache that supports long-lived connections will be able to cache connections on both client and server sides, besides performing its data caching function.

Caching on the Web works by taking advantage of temporal locality — people request popular resources and the more popular a resource is, the more likely it is to be requested again in the near future, either by the same client, or by another. In one study encompassing more than a month of Web accesses, of all the resources requested by individual users, close to 60% were requested on average more than once by the same user [32]. Also, another caching study showed that of the hits provided, up to 85% were the result of multiple users requesting the same resources [31].

2.2.4 Web Caching Architectures

2.2.4.1 Architectures

Caching is performed in various locations throughout the Web: clients, servers, and, eventually, in intermediary proxies. Starting in the browser, a Web request may potentially travel through multiple caching systems on its way to the origin server. At any point in the sequence, a response may be served if the request matches a valid response in the cache. In this chain of caches, the internal nodes are termed proxies, since each will generate a new request on behalf of the user if it cannot satisfy the request by itself. A user request may be satisfied by a response captured in a browser cache. If not, it may be passed to a department– or organization–wide proxy cache. If a valid response is not present there, the request may be received by a proxy cache operated by the client's ISP. If the ISP cache does not contain the requested response, it will likely attempt to contact the origin server. However, reverse proxy caches operated by the content provider's ISP or content delivery network (CDN) may instead respond to the request. If they do not have the requested information, the request may ultimately arrive at the origin server. Even at the origin server, content can be stored in a server-side cache so that the server load can be reduced.

Cache chains such as the one described above constitute hierarchical caching structures, which are quite common [33]. In order to distribute load, there can be several caches at the same level in the hierarchy. In these cases, each cache may serve many clients, which may be users or other caches. In case a cache cannot satisfy a given request, it will decide independently whether to pass it to

another cache in the same level, or to a higher level in the hierarchy, until reaching a root cache, which requests the resource from the origin server.

An alternative is to use a cooperative caching architecture. In cooperative caching, the caches communicate with each other as peers (*e.g.*, using an inter-cache protocol such as ICP [34]). In the case of a miss, a cache asks a predetermined set of peers whether they already have the missing resource. If so, the request is routed to the first responding cache. Otherwise, the cache attempts to retrieve the resource directly from the origin server. This can prevent storage of multiple copies of a given resource and reduce origin server retrievals. However, in cooperative cache architectures there is an increased overhead due to inter-cache communication. To mitigate this communication overhead, a possibility is for a cache to periodically provide its peers a summary of its own contents [35, 37]. Combinations of these approaches are also used [36].

2.2.4.2 Transparency

Caching may be transparent to the client in different degrees: a client may or may not know about intermediate proxy caches. If the client is configured directly to use a proxy cache, it sends to the proxy all requests not satisfied by a built-in cache. Otherwise, it has to look up the IP address of the origin host. In that case, if the content provider is using a CDN, the DNS servers may be customized to return the IP address of the server (or proxy cache) closest to the client. In this way, a reverse proxy server can operate as if it were the origin server, answering immediately any cached requests, and forwarding the rest.

Even when the client has the IP address of the server that it should contact, along the network path there may be a switch or router that will 'transparently' direct all Web requests to a proxy cache. In this situation, the client believes it has contacted the origin server, but instead has reached an interception proxy which serves the content either from cache, or by first fetching it from the origin server.

2.2.4.3 Content cacheability

Not every resource on the Web is cacheable. Of those that are, some are cacheable for long periods by any cache, whilst others may have restrictions such as being cacheable only for short periods or only by caches of given types, *e.g.*, non-proxy caches. The cacheability of a Web site affects both its user-perceived performance and the scalability of a particular hosting solution. A cached resource can be rendered at the client browser almost instantaneously, instead of taking seconds or minutes to load.

The cacheability of a resource is determined by the content provider: it is the Web server software that sets and sends the HTTP headers that determines a resource's cacheability, and the mechanisms

for doing so vary across Web servers. To maximize the cacheability of a Web site, typically all static content (such as buttons or graphics that seldom change) are given an expiration date far in the future so that they can be cached for weeks or months at a time. By doing this, the content provider is trading off the potential of caching stale data with reduced bandwidth usage and improved response time for the user. A shorter expiration date reduces the chance of the user seeing outdated content, but increases the number of times that caches will need to validate the resource.

Some caches use as consistency mechanism a client polling approach: clients periodically check back with the server to determine if cached resources are still valid. Even when using variants that favor polling for resources that have changed most recently [38], this practice can still generate significant overhead. To mitigate this communication overhead, another solution is to have the origin server invalidate cached resources [39]. There have been proposals for protocols in which Web caches subscribe to invalidation channel(s) (e.g., WCIP [40]), corresponding to content in which the caches are interested. These protocols are intended to allow caching and distribution of large numbers of frequently changing Web resources, with freshness guarantees.

Dynamically generated resources are typically taken as uncacheable, although that is not necessarily true. Resources that are dynamically generated constitute an increasing fraction of the Web. Examples of dynamically generated resources include fast-changing content like stock quotes, real-time images or e-commerce shopping carts. It would seldom be desirable for any of these to be cacheable at an intermediate proxy. Nevertheless, some may be cacheable in the client browser cache, such as personalized resources that do not change regularly. One proposed solution to allow caching of dynamic content is to cache programs that generate or modify the content, such as an applet [41]. Another possibility is to split resources into separate blocks, with different caching properties. This enables separate caching of each block, thereby allowing optimization of server-side operations (*e.g.*, products from companies like SpiderCache [42] and XCache [43]). In dynamic resources, usually a significant fraction of the content is static, therefore it makes sense to just send the differences between resources or between different versions of the same resource [44, 45]. Another possibility is to break the resources into separately cacheable pieces and reassemble them at the client [47], or at edge servers [46].

2.2.5 Limitations of web caching

Despite the undisputed benefits it provides, already discussed above, caching can introduce a number of potential problems as well. The most significant is the possibility of the end user seeing outdated content, compared to what is available at the origin server (*i.e. fresh* content). HTTP does not ensure strong consistency and therefore there is a real possibility that data is cached for too long. The probability that this occurs represents a trade-off between caching stale data, on one side, and reducing bandwidth usage and improving user response times, on the other. This compromise must be explicitly managed by the content provider.

Another drawback is that, even if the resource hasn't changed, it can happen that the timestamp is modified by the server, *e.g.* if there is a new external advertisement in a given web page. In cases like this, when there is a new request for the resource the client may have the same content in its cache, but because of the new timestamp the whole resource will still be downloaded. Furthermore, if client and server have slightly different versions of a given resource, due to minor modifications such as just a change in the date or time, caching will not take advantage of the significant redundancy existing between the versions of the resource.

The benefits of caching are by definition limited, since a cache can only improve the response times in case of hits, *i.e.* cached responses that are subsequently requested (typically maximum hit rates of 40-50% are achievable with sufficient traffic [16]). Cache misses generally present lower speed, as each system through which the request travels will increase the overall user-perceived latency. Caching is also limited by a high rate of change for popular Web resources, and by the fact that many resources are requested only once [48]. Moreover, some responses cannot or should not be cached, as discussed previously.

Nevertheless, if future requests can be anticipated, those resources can be obtained in advance. Once available in a local cache, they can be retrieved with minimal delay, improving the user-perceived latency. That is the purpose of Prefetching, which is discussed in the following section.

2.3 Web Prefetching

Prefetching is a well-known approach to decrease access times in the memory hierarchy of modern computer architectures [26, 49, 50]). The basic principle behind it is to anticipate future data requests, and getting the necessary data into cache in the background, before an explicit request is made by the CPU. Prefetching has also been proposed as a mechanism to improve user-perceived latency in the Web [51, 52]. Web prefetching's purpose is to preprocess a user's request before it is actually demanded and, in this way, hide the request latency.

The idea seems promising, however prefetching is not straightforward to evaluate and has not yet been widely implemented in commercial systems. It can be found in some browser add-ons [53] and workgroup proxy caches [54], that will prefetch the links of the page the user is currently browsing, or periodically prefetch the pages in a user's bookmarks.

Prefetching is usually transparent to the user: there is no interaction between the user and the prefetching system. Prefetching systems are speculative by nature and therefore there is an intrynsic probability for the predictions to fail. If the prediction is not accurate, cache pollution, bandwidth waste and overload of the original server can occur. Another concern is the need to determine what content may be safely prefetched, as some Web requests can have undesirable side-effects, such as adding items to an online shopping cart. Prefetching must thus be carefully applied: *e.g.*, using idle times in order to avoid performance degradation [55]. Despite the mentioned risks, it has been demonstrated that several prefetching algorithms [56, 57, 58, 59, 60] can considerably reduce the user-perceived latency.

The prefetching predictions can be performed by the server, by the proxy, or by the client itself. Some works suggest performing the predictions at the server [57, 58], argumenting that its predictions can be quite accurate because it is visited by a high number of users. Other studies defend that proxy servers can perform more accurate predictions because their users are much more homogeneous than in an original server, and they can also predict cross-server links, which can reach about 29% of the requests [61]. On the other hand, some authors consider that predictions must be performed by the client browser, because it is better aware of users' preferences [62, 63]. Finally, some studies indicate that the different parts of the web architecture (users, proxies and servers) must collaborate when performing predictions [58]. The following sections describe the current state of the art in web prefetching.

2.3.1 Architecture of a Generic Web Prefetching system

A generic architecture for prefetching systems is defined: in order to implement this technique, two new elements are added to the generic Web architecture previously described in section 2.1, the prediction and prefetching engines. These can be located in the same or different elements, at any part of the system.

2.3.1.1 Prediction Engine

The prediction engine is the part of the prefetching system that has the purpose of guessing which will be the next user's requests. It can be located at any part of the web architecture, whether in the clients [62, 63], in the proxies [59, 64] or in the servers [65, 66, 67, 68]. It can even work in a collaborative way between several elements [58]. The patterns of user's accesses differ depending on the element of the architecture in which the prediction engine is implemented, simply due to the fact that the information each one is able to gather is totally diverse. For example, a predictor located at the web server is not able to gather cross-server transitions and therefore is limited to transitions between pages of the same server. Several algorithms that attempt to predict future accesses were devised (please check section 2.3.1.1.1 below).

The prediction engine outputs a hint list, which is a set of URIs (or a set of servers if only the connection is going to be prefetched) that are likely to be requested by the user in a near future. The predictor must distinguish between prefetchable and non-prefetchable resources. A web resource is prefetchable if and only if it is cacheable and its retrieval is safe [16]. Therefore, the hint list can only contain prefetchable URIs (*e.g.*, browsers based on Mozilla [69] consider that a resource is not prefetchable if the URI contains a query string).

2.3.1.1.1 Prediction Algorithms

As was mentioned above, a quick prediction of users' accesses is a key point in web prefetching. A variety of approaches for prediction algorithms can be found in the literature. In Markov and Markov-like models, the possible next user actions are encoded as states with calculated probabilities of transition from the current state. The higher the order of the model, the highest the prediction precision will be. Markov models have shown considerable success in modeling and predicting users' browsing behavior [51, 70, 71, 72, 73, 74]. However, an increase in the order of the model also implies an increase in the computational resources needed to calculate the prediction. Therefore, other approaches were proposed by several authors. Relevant examples are Prediction by Partial Match [75, 76, 77, 78] and TDAG [79, 80], which combine Markov models of varying order. These methods have roots in data compression, and often have complex implementations. Subsequence matching algorithms use past sequences of user actions to provide a possible next action for the current sequence [81, 82, 83].

2.3.1.2 Prefetching Engine

The prefetching engine's function is to preprocess resource requests that were predicted by the prediction engine. By preprocessing the requests in advance, the perceived latency when the resource is actually requested by the user is reduced. Most proposals for preprocessing have focused mainly on the transference of requested resources in advance [8, 57, 84, 85, 86, 87]. Nevertheless, some works consider the preprocessing of a request by the server [88, 89], whilst others suggest the pre-establishment of connections to the server [90]. Therefore, the prefetching engine can be located at the client, at the proxy, at the server, or in several of these elements.

The ability of web prefetching to reduce user-perceived latency is strongly dependent on where the prefetching engine is located: the closer to the client it is implemented, the higher its impact on latency. Therefore, a prefetching engine located at the client can reduce the whole user-perceived latency. This seems to be the current trend as it is included in commercial products like Mozilla Firefox [14].

Furthermore, in order to avoid interference between prefetching actions and current user requests, the prefetching engine can consider external factors to decide whether and when to prefetch a resource hinted by the prediction engine. These factors can be related to any part of the web architecture. For example, when the prefetching engine is located at the client some commercial products only start the prefetching after the user is idle [14, 91].

2.3.2 Commercial and Academic Systems

Despite having been a subject of research already for some years, the commercial penetration of web prefetching is still poor, mainly due to its potential impact on bandwidth.

The main commercial contribution to promote web prefetching as a mean for reducing user-perceived latency is being made by the Mozilla Firefox browser [14]. The Firefox web client contains a web prefetching engine that parses HTTP headers to find prefetch hints included by the server or proxy. Firefox prefetches sequentially all the hinted URLs found in the header, in periods when the user is idle after downloading the page. This technique requires the server to make the predictions and add them to the HTTP headers.

One of the most representative examples of Web sites that take advantage of the prefetching technique implemented by Firefox is Google Search: in some situations, a prediction hint pointing to the URL of the first result is included in the response header [92].

Google Web Accelerator [91] was a software that used web prefetching to improve user-perceived latency. It worked together with Mozilla Firefox or Microsoft Internet Explorer, implementing a variant of the prefetching engine described by [14]. However, GWA presented some serious security problems and Google discontinued it as of January 2009.

PeakJet 2000 [93] and NetAccelerator [94] both implement two prefetching modes: historic and link-based. In the historic mode, the software refreshes all the resources in the cache that were accessed by the user. In the link-based mode, all the linked resources in the page currently being browsed by the user are downloaded. The difference between Peakjet and NetAccelerator lies in where prefetched resources are saved: Peakjet has its own cache, while NetAccelerator uses the browser cache.

Other commercial software implementing web prefetching techniques are Robtex Viking Server [95] and AllegroSurf [54]. However, no relevant details are made publicly available by the developers on how prefetching works in these products.

In Academia, we highlight a simulation framework developed in the University of Valencia [111]. This Web prefetching framework models the generic Web prefetching architecture, allowing to implement and to check the performance of prefetching algorithms. It is composed of:

- the back end part which has both a surrogate proxy server and the real web server. The web server is external to the simulator environment, which accesses it through the surrogate. The surrogate module is where the prediction engine is implemented.

- the front end has the client component, which represents the user behavior in a prefetch-enabled client browser. The user web access pattern is simulated by feeding the framework with real traces obtained from a Squid proxy. The prefetching engine is implemented in the client component.

The authors have presented several examples of experiments successfully simulating real-world operating conditions [68,111].

2.4 Data deduplication on the Web

Deduplication attempts to reduce data footprint through detection and elimination of redundancy in the data. Based on these principles it has been applied to the Internet with the purpose of reducing the total transmitted bytes and, consequently, the user-perceived response time. Since deduplication leverages on the redundancy present in the data transferred on the Web, it is important to characterize the degree to which redundancy is present therein.

2.4.1 Data redundancy on the Web

Several authors have studied the potential existence of redundancy in web content. Mogul *et al* [20], using the trace of a proxy site from the internal network of Digital Equipment Corporation, concluded that 55% redundancy can be found between two versions of a resource referenced by the same name. This can eventually reach as high as 87% if the reference resources are chosen adequately [20, 96]. Furthermore, the number of HTTP responses that show some form of redundancy with other cached resources can, depending on the client cache properties, be doubled when a proxy is used relative to the case of individual clients, reaching up to 38%. However, at the packet level this amounts to only 10%. According with the authors this is due to the static nature of images: 25% of the overall packet level responses show redundancy, whilst for image responses this happens in only 3.5% of the cases. Furthermore, close to half of the redundant responses show almost full redundancy, a result that demonstrates the possibility for major savings using deduplication techniques.

The results at the resource level for different content types show that images are the most static, less prone type to partial deduplication, with a rate of duplicated bytes of 5% to 7%. On the other hand, text resources with changes, either HTML pages or plain text, show rates of 60% to 70%. This means that text resources, changed or unchanged, can provide very good deduplication results.

In the case of query URLs[1] clustering different queries with a common prefix brings benefits relative to using the full URL: 90% responses show redundancy if only the prefix is used for reference files, versus 33% responses when using the full URL, a result confirmed by WebExpress [44]. For Gzip compression applied to full text resources, 55% to 69% compression rates are obtained but the impact of Gzip in the deduplicated responses is not evaluated. The results also show that Gzip does not provide efficient compression for image responses (3% to 5% maximum compression is obtained).

In another study, Spring and Wetherall [17] tested redundancy at the packet level with a protocol-independent technique. They report that approximately 30% of the incoming traffic on their proxy is redundant, in agreement with Mogul's results for a packet trace. The highest volume traffic was for HTTP, which showed a 30% redundancy rate. Next in traffic volume are several media transfer protocols, but these present very low redundancy levels (around 7%). For the protocols with the lowest

traffic volumes, *i.e.* SMTP, POP and Telnet, the redundancy levels are similar to that of HTTP. Anand *et al* [18], on the other hand, found some differences in their study relative to these results. Namely, they detected less redundancy in HTTP, which the authors justify with the increasing usage of port 80 for all types of traffic, such as streaming media and games. This result probably expresses the evolution that happened in the types of web traffic between the times at which the studies by Spring and Wetherall [17] and Anand *et al* [18] were performed.

Kelly and Mogul performed a study on the effects of aliasing [21], using large traces from WebTV Networks (2000) and Compaq Corporation (1999). MIDI resources show the most aliasing, but represent only a small amount of the overall network traffic. However, for image file formats – GIF, JPEG – the results show that they have 63% and 31% aliased bytes, respectively, and represent 34% and 26% of the byte count. In the case of text/HTML resources only 13% aliasing is measured, although these represent 22% of the overall traffic. In the study of Rhea et al. [19], the measurements indicate that aliasing is not a very common occurrence: there were only 3% of aliased bytes in the analyzed trace.

2.4.2 Deduplication techniques

Here we discuss the major types of deduplication techniques currently used for the Web. When discussing deduplication applied to the Web, one should note that work on this area has actually been based upon previous work done on deduplication for distributed file systems, like LBFS [97] or TAPER [98]. These general techniques were ported and then complemented with Web-specific techniques.

2.4.2.1 Cache-based approaches

In the classic cache approach, the browser keeps whole resources in its cache, indexed by URL. When a URL is requested to the server, the browser checks if the resource is already present in its cache. In case it is, the browser checks a timestamp to see if the resource in cache is up to date. In that case, the browser does not download the content. Otherwise, the whole resource is downloaded; the same also happening if the resource is not present in the cache at all.

This approach is currently used on all Internet browsers, and corresponds to the HTTP specification [114]. It has worked well due to its simplicity of use, both on browser and server sides. Nevertheless, as discussed previously in section 2.2.5, it presents some drawbacks: even if a requested resource hasn't changed but its timestamp has been modified by the server, the whole resource will still be downloaded. Additionally, if there are only slight differences between client and server versions of the same resource (*e.g.*, due to minor modifications such as just a change in the date or time), the classic

[1] URLs with a "?" character

cache approach will not take advantage of the significant redundancy existing between the versions of the resource.

The most relevant work in this area is Edge Side Includes (ESI) [99], by Akamai. ESI is a markup language that allows developers to divide Web pages in sections according with their caching properties, thereby allowing a fine grain control over the caching behavior even of single Web pages. In this way, Akamai's servers are then able to handle these pieces separately, as independent pages, thus allowing distribution of the network traffic through different geographic locations, thereby improving overall network flow. This technology, however, is only deployed in Akamai's internal network and, therefore, does not impact directly the user. Moreover, it puts an extra burden on the programmer due to the need of specifying explicitly the sections on which the pages should be split and is, due to that, particularly error-prone. It also lacks extensibility: modifications in a given page imply that the markup must also be changed.

2.4.2.2 Delta-encoding

In delta-encoding, two files are compared and their differences are computed. The result of this direct comparison – the *delta* – is thereby obtained. Examples of delta-encoding are the Unix utility *diff*, its binary-applicable equivalent *bdiff* and the *vdelta* algorithm [100].

When applied to the Web it means that after a first download of a complete page, in a second request, if there were changes to the page, a delta can be computed between the two versions of that same page. The client then downloads only the delta and reconstructs the new version of the page from the one in its cache and from the delta.

One of the issues in this approach is that it demands that the server keeps at all times the latest version of a reference file that was sent to each client. This has a consequent impact on disk space usage, which is much higher than with classic caching. Also, with increasing number of reference resources comes an increasing performance overhead: to improve redundancy detection the delta can be encoded from several reference files and this implies that they all need to be analyzed, in order to optimize redundancy detection for the lowest possible number of references. On the other hand, the redundancy detection algorithm is executed locally on the server with no additional data being transferred between client and server other than the encoded delta and file version metadata. This means that there is no transfer of redundant data.

WebExpress [44] was the first relevant work to be published on deduplication for the Web. Its purpose was to allow web access in wireless environments by decreasing bandwidth consumption. It relied on a client proxy and a server proxy for implementation and used delta-encoding to cache dynamic pages and send to the client only the new content for each request of already stored resources. To save static resources a classic cache system was used. The WebExpress system though was not designed for *ad hoc* browsing, but rather for environments where a high degree of

redundancy would exist, such as applications used by company employees, where most requests would be query URLs on the same pages.

In order to reduce WWW latency, Banga *et al* [101] developed an optimistic deltas system. The system is based on delta-encoding: it saves the resource sent to the client in the server's cache, then when the client sends a subsequent request for that same resource the server creates a delta between both versions and sends the delta to the client, which fully reconstructs the resource. However, when the client still does not have a given resource, on the receipt of a first request for it the server immediately sends to the client a cached version of it. If the cached version is not stale it is necessary only to send an additional message stating that. Otherwise, the server encodes a delta between the two versions and sends it to the client, so it can fully reconstruct the new resource.

The system is optimistic because it expects that there is enough idle time to send most or all of the old version of a resource while the new version is being reconstructed. It also assumes the changes between two versions are small relative to the whole resource size.

Based on the study of the potential benefits of delta-encoding [20], RFC3229 [102] was proposed. In it, Mogul *et al* describe how delta-encoding can be supported as an HTTP/1.1-compatible extension. The authors argue, based on a trace-based analysis, that the use of delta-encoding can be beneficial and suggest specifications for the basic mechanisms, to add some new, extra headers to support delta-encoding, which delta algorithm to be used and also rules for deltas in the presence of content codings.

Wyman produced a compilation of client and server implementations of RFC3229 up to 2006 [103]. These refer mainly to RSS feed systems, on which delta-encoding shows high impact and it is easier to implement than for regular Web pages. Some high profile references are by Microsoft, which used RFC3229 in the Windows RSS Platform (Vista+) and in Internet Explorer 7+.

Cache Based Compaction [104] is a technique developed by Chan and Woo in which the main objective was to create an efficient selection algorithm for reference resources. This was not that much addressed in previous work on the topic: other authors used only previous versions of the same resource or, in the case of query URLs, reference resources of different queries to the same resource. Chan and Woo, on the other hand, devised an algorithm based on URL structure and on a dictionary-based compression algorithm for delta-encoding the resources.

The algorithm they created ends up covering a broad set of scenarios: when there are no reference resources it acts as a simple dictionary-based algorithm, *e.g.* Gzip, for the full resource. With only one reference present, namely a previous version of a given resource, the algorithm behaves similarly to the WebExpress differencing algorithm. In the case there are several references for a resource, these serve as extra sources for the dictionary strings.

The major problem with delta-encoding approaches in general is the need to store reference resources at the server. This means the server has to keep state, which also implies that clients and server must be synchronized. For each request, the server needs to know which resources it is still sharing with the client, in order to be able to calculate a corresponding delta.

2.4.2.3 Compare-by-hash

In compare-by-hash (CBH) both client and server divide resources in data blocks ("*chunks*"), which are identified by a cryptographic hash and are treated as autonomous data units: they are shared by different resources in cases where the data is redundant. When the client requests a new version of a resource the server determines which chunks correspond to that resource version and sends their hashes to the client. The client then compares the hashes sent by the server with the ones it has stored locally and computes which chunks it still needs to reconstruct the new resource version. Following, the client requests the server only these chunks it does not have locally. The server answers the request by sending the new chunks and also the hashes of the redundant chunks. In this way the client is able to properly reconstruct the new resource.

From this description, one of the problems of CBH becomes evident: it needs an additional roundtrip, since besides the resource request the corresponding hashes must be sent from server to client and vice-versa. Delta-encoding, on the other hand, only sends control data in the response for resource reconstruction. Furthermore, the redundancy detection phase requires exchange of information through the network and so is not a local algorithm. Thus, CBH suffers double the network latency and packet loss problems and is slower than delta-encoding, which can be an important issue in a real-time system like the Web.

Another problem to consider in CBH is how to find a right balance regarding chunk size: if it is too small there will be too many hashes to trade between client and server, this resulting in a large communication overhead. Also, a large number of chunks results in an increased memory use by chunk's hashes. Opposite, the larger the size of the used chunks, the less the possibilities for redundancy detection. Therefore, a compromise must be found regarding chunk size, such that good redundancy detection is achieved without a high impact of metadata on overall data transfers.

On the positive side, chunking of client's cache files diminishes their overall memory use. Since only one block of redundant data is kept for all files that share it, some space is saved in comparison to delta-encoding (delta encoding must keep the entire content of one reference version of each file). Of course this also implies that whenever a page is requested it has to be reconstructed from its reference blocks, not just loaded from disk like in the previous approaches.

Spring and Wetherall propose to eliminate redundant network traffic by means of a protocol independent technique [17]. Working at the packet level, it detects redundant chunks of information on different packets that arrive at an enterprise proxy, which indexes them in memory. The proxy then transfers only the non-redundant data chunks and the indexes of the redundant parts to the client proxy. Communication between the client proxy and the clients is done using normal packets.

Redundancy detection is performed using Rabin fingerprints [105]. In these, a rolling hash function is used that goes through the packet byte string and creates a unique integer – the *fingerprint* – that corresponds to a chunk of that string. Each byte is a starting position for a chunk to be hashed. This makes the algorithm incremental, since the next chunk's hash can be computed from the previous hash and by performing additional arithmetic operations with the new byte. This characteristic makes

the Rabin fingerprinting algorithm suitable for real-time computation, as opposed to other commonly used hash functions, such as MD5 or SHA1.

Not all fingerprints can be stored for redundancy detection, since there is almost one fingerprint per byte in the packet string. Therefore, a value-based selection algorithm is used, usually Fingerprint mod M = 0. In this way, fingerprint selection is random and uniformly distributed. Also, it is better than using the Nth fingerprints because, contrary to the latter, the former provides probabilistically good coverage of the data. The Nth fingerprints attributes a fixed length to the chunks, which is provenly not a good approach when the goal is to maximize detected redundancy [106].

According to Schleimer *et al* [107], it is possible to improve the value-based fingerprint selection algorithm regarding redundancy detection, by selecting a maximum or minimum value as fingerprint for the current chunk instead of using the mod M approach. This is called Winnowing. Other authors also confirmed these results: comparisons between regular mod M Rabin fingerprints, Winnowing Rabin fingerprints and other techniques always showed Winnowing to be the best performing solution concerning redundancy detection [18, 106].

Value-Based Web Cache [19] is a system developed by Rhea *et al*, with the purpose of improving perceived latency of Web pages in a bandwidth constrained link. The system has three main components: client, ISP proxy and origin server. Upon a request for a resource by the client, the request is forwarded to the origin server. It then sends the response to the ISP proxy, which divides the resource into chunks and hashes them. The proxy saves only the hashes, mapped by client. It checks if those hashes were not previously transferred to the client and sends it only the hashes of the redundant chunks and the new chunks the client does not have locally. The client is then able to reconstruct the requested resource using the new chunks it just received and also the redundant ones stored in its cache.

To divide resources into chunks, VBWC uses Rabin fingerprints to define each chunk's boundaries. Then the chunk's digest is created by applying MD5 or SHA1 to the data between the boundaries. In this way resource modification and aliasing can be detected, because what is indexed is the digest value of the chunks, not the name of the resource. Therefore, for aliased resources only the hashes will be transferred, not the whole resource.

VBWC presents some issues and inefficiencies. Since VBWC uses a user-centric approach, aliasing detection is not that much efficient. The hash cache is mapped by client and therefore there is no cross-client redundancy detection. Also, the bandwidth savings are confined to the ISP proxy-client connection. On the other hand, if the proxy cache would be implemented in the origin server the bandwidth savings could be propagated to the whole Internet infrastructure.

Hierarchical Substring Caching [108] is an optimization technique to increase the hit rate of the chunks used by VBWC, thereby improving redundancy detection. The authors create a hierarchy of chunks, in which each level has a defined chunk size. The smaller the chunks the higher the probabilities of finding a matching block. However, it must be taken into account that for increasing numbers of hierarchy levels there is a corresponding increase in metadata space in the proxy cache. Nevertheless, the additional bandwidth consumption due to the new chunk hashes is negligible when compared to that of the chunks themselves and the computational overhead associated with their

creation is very low when using composable Rabin fingerprints [105]. The authors report an improvement of 25% in redundancy detection when compared to VBWC.

Kelly and Mogul proposed an HTTP/1.1-compatible extension to allow the suppression of duplicate resource transfers, Duplicate Transfer Detection [21, 109]. DTD can be used at any connection in the infrastructure, whether server-to-proxy, proxy-to-client, server-to-client. DTD works in the following way: when a client requests a resource, the origin server gets the resource and creates its digest with MD5 or SHA1. The server then responds to the client with the digest. The client checks if that digest is indexed in its cache; if so it answers the server not to send the resource; otherwise, the server sends the full resource to the client. In this way, the cache can have the regular check by resource name combined with a check by digest, thereby improving its hit rates. The authors implemented DTD on the Squid proxy cache [110] and report latency improvements over the base Squid cache both for small and large files.

DTD also has its own problems, most notably the fact that it needs an additional roundtrip for digest verification, similarly to what was discussed above for CBH.

DedupHTTP [112] is an end-to-end deduplication system for text resources. Its deduplication algorithm combines delta-encoding and compare-by-hash schemes, acting mostly on the server side, with manageable server state and low communication overhead. According with the authors, it presents several advantages over caching, delta-encoding, compare-by-hash and cache based compaction, namely:

- it can be deployed in several points of the network architecture;
- supports any Web Browser and Web Server when deployed on proxies;
- it does not enforce synchronization between client and server;
- the redundancy detection algorithm for online deduplication has a low computational overhead;
- it employs cross-url and cross-version redundancy detection, and keeps old versions of resources archived on the server, allowing higher detection rates;
- improves precision in redundancy detection relative to other solutions, since it allows to have very small chunk sizes with no communication overhead;
- the communication protocol has a low overhead, involving only one network roundtrip;

2.4.3 Deployment

Data deduplication systems for the Web may be analysed according with the instant in time at which redundancy detection is performed, the location in the network stack where deduplication detection occurs and the location in the network infrastructure where the algorithm is deployed.

Regarding the instant at which redundancy detection is performed, systems can be classified as *online* or *offline*. Online systems perform redundancy detection upon each requested resource at response time (*e.g.* VBWC [19]). These systems are computationally demanding, but are able to detect redundancy on dynamic contents. Opposite, offline systems perform preemptive redundancy

detection on resources that may possibly be requested (*e.g.* Optimistic Deltas [101]). This solution is much less computationally demanding than online deduplication. However, it has a major memory overhead, requires resource synchronization between client and server, and cannot act upon dynamic resources.

As to the location in the network stack, deduplication can be targeted at the *resource level (application layer)* or *packet level (transport layer)*. Working at the resource level (*e.g.* Cache based Compaction [104]) allows to study the existence of redundancy between different resources as a function on several criteria, such as: where most redundancy is found through the comparison of resource URLs, if there is more redundancy between resources of a given directory, domain or between different domains. This allows evaluating how broad the detection algorithm needs to be. The narrower the choice of resources, the lower the search overhead will be. It also allows studying which users share what data, *e.g.* in a university setting it is possible to determine whom are the most files redundant for: only the current user, all users across a room or department or the whole campus. This helps to determine where the deduplication system should be deployed.

When working at the packet level (*e.g.*, Cooperative Caches [17]) one is at a lower level of data granularity, where one is not aware of the resources being processed. This has the main advantage that most of the issues raised by resource level approaches, such as the source of redundancy between documents or users, but also the data protocol(s) being dealt with, can be ignored. However, this protocol independence when using packet level approaches raises some concerns. One is how much data redundancy is there across different protocols. Another issue is if the transferred volume of the less used protocols is enough to justify taking them into account in deduplication. Also, could the space waste due to ignoring all of the previous concerns be large enough to justify working at the packet level?

In what concerns the location of the deduplication system in the network infrastructure, there are several possibilities. The great majority of the produced work is either end-to-end approaches, middlebox systems or hop-to-hop systems. In end-to-end solutions, like VBWC [19], both the end client and origin server are active players in the deduplication system. In middlebox solutions (*e.g.* Cooperative Caches [17]) the core of the deduplication system is an ISP or internal enterprise proxy, which communicates directly with end clients or through end client proxies. On the other hand, hop-to-hop approaches (*e.g.* DTD [109]) can be applied at any stage of the network stack.

The majority of work in deduplication systems, both research and commercial, is in the middlebox area. However, interest has developed lately in deduplication at the packet level on Internet routers, which seems promising in saving redundant data shared by several users.

3 Architecture

As seen above, independently of the amount of bandwidth that is available, Web prefetching can end up consuming all free bandwidth. Data deduplication is a technique that can reduce the amount of data transferred in the network, freeing occupied bandwidth. Therefore, the combined use of both techniques is expected to potentially bring considerable improvements to the amount of transmitted bytes per Web request. It is also expectable that this reduction can have a corresponding reduction in the user-perceived latency in Web navigation.

This section presents the architecture for the system we designed and implemented. The system uses both prefetching and deduplication techniques, with the purpose of optimizing user-perceived latency in Web navigation. We used as starting point the Web prefetching simulator framework previously mentioned in section 2.3.2 [111], made publicly available by its authors. We extended this system in order to implement data deduplication. To this purpose, we used a deduplication technique based on a recent state-of-the-art system that applies deduplication to web traffic, dedupHTTP [112].

This chapter is organized as follows: in section 3.1 we describe the architecture of the Web prefetching framework we used as starting point for our system. Section 3.2 describes the deduplication technique we implemented. Finally, in section 3.3 we present the architecture of the full system we designed and implemented.

3.1 Web prefetching framework

Here we describe the architecture of the Web prefetching framework [111] we extended to build our system. It constitutes a simulation environment, which is composed of two main parts: the surrogate proxy and the client module (Fig. 4). The surrogate connects to a real web server, external to the simulator. The client module represents the user behavior in a prefetch-enabled client browser. Below we describe in detail each of these main components.

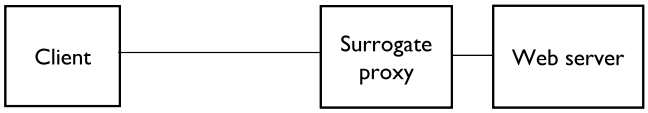


Fig. 4- Architecture of the simulation environment (adapted from [111]).

3.1.1 Surrogate proxy module

As mentioned previously in section 2.1, a surrogate proxy is an optional element of the generic web architecture, located close to an origin server, that generally acts as a cache of the most popular server responses. In the simulator framework, the authors implemented the prediction engine in the surrogate proxy module [111]. The hints generated by the prediction engine about which resources should be prefetched are sent by the surrogate to the client. This is done using an optional header defined in the HTTP/1.1 standard [114], in an identical way to what the Mozilla Firefox browser does [14]. For each response, the surrogate sends to the client a list of hints in the HTTP response headers, which is generated by the prediction engine.

The block diagram of the surrogate proxy is shown in Fig. 5. In it, each block represents a functional unit [111]:

The *Listener* block is basically implemented as a thread that waits for clients to connect to its socket, acting as a TCP server that listens at a given port (the port number is defined in a configuration file). By default, there are no restrictions on the number of simultaneous incoming connections. The function of this block is to create a new *Connection to server* thread.

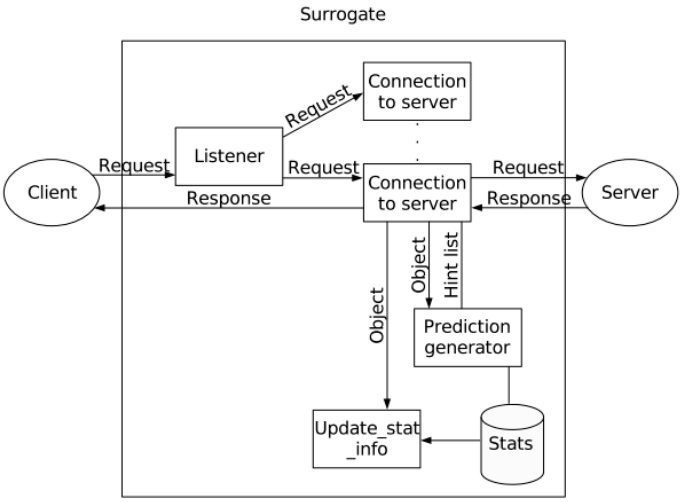


Fig. 5- Block diagram of the surrogate proxy module [111].

The *Connection to server* block, which is the main one in the surrogate, handles client-server connections. The connection involves a request from the client that is transferred to the web server, which processes it and transfers the requested resource back to the client. The *Connection to server* block then intercepts the response and parses it until all the HTTP response headers are received. It then requests a prediction from the *Prediction generator* block. The prediction consists of a list of hints generated by the prediction algorithm [66].

The *Prediction generator* block implements the task of generating predictions each time a resource is requested. It generates a list of URL's of the resources with the highest probability to be accessed in a near future, according with the prediction algorithm [66] and taking into account the *statistical database* contents.

After receiving all the response headers, the *Connection to server* block notifies the *Update_stat_info* block. This information is used to update the contents of the *statistical database*, which is used to generate predictions. Finally, the *Connection to server* block sends the full response data (headers+ resource data) back to the client: it first sends the response headers (piggybacking the hintlist on the original HTTP response headers); then it just transfers the resource data, as received from the server, to the client.

3.1.2 Client module

The client module of the system simulates the behavior of a real user browsing the WWW with a prefetching-enabled web browser. The user's behavior is simulated by feeding the client module with real traces, obtained from logs of real web requests, directed through a Squid proxy operating in transparent mode [110]. The time intervals between successive client requests are respected by taking into account the real timestamp differences read from the input logs.

Fig. 6 shows the block diagram of the client module [111]. This module communicates only with the input traces, an output log file and with the server. The latter is replaced, in the present implementation, by the surrogate proxy component previously mentioned.

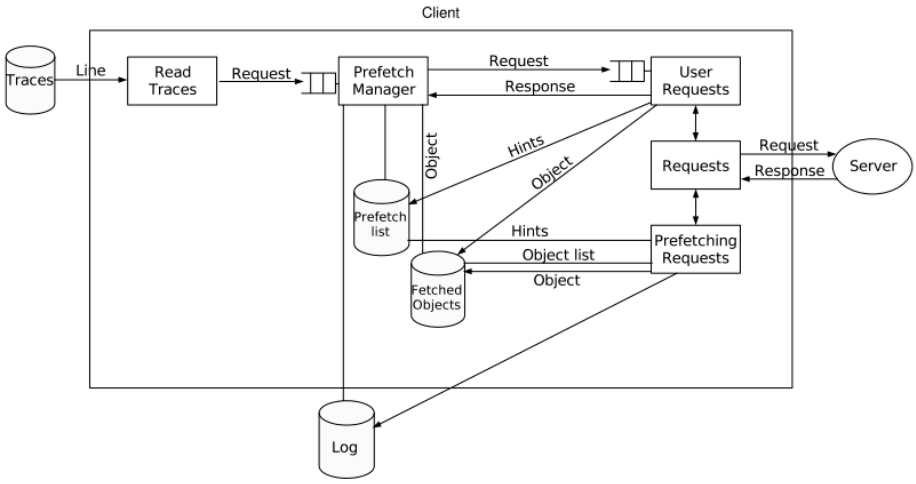


Fig. 6- Block diagram of the client module [111].

There are two data stores inside the client: the *Prefetch List* and the *Fetched Objects*. The *Prefetch List* contains the list of resources that the server hints the client to prefetch. The *Fetched Objects* data store keeps all the fetched resources, emulating the user browser cache. Nevertheless, in the current implementation, the authors do not store the resource data in this data store, keeping only the meta-information of the resource, namely its URL and if the resource was prefetched [111]. If the resources it contains were previously prefetched, *Fetched Objects* stores also information on whether the user has accessed them at a later time or not.

The *Read Traces* block reads the user requests from the input trace. In this module, each access line read from the input trace is inserted in a FIFO request queue. Each insertion is timed according to the timestamps on the input trace file, to match the access pattern of the original user.

As previously mentioned in section 2.3.1.2, the positive impact of web prefetching on user-perceived latency is highest when the prefetching engine is located at the client. The prefetching engine is implemented in the *Prefetch Manager* block. The *Prefetch Manager* block reads requests from the *Read Traces* queue. It then checks the *Fetched Objects* data store, to verify if the resource has not been prefetched yet. In the case that it has not been prefetched, the *Prefetch Manager* block sends a request for the resource to the server. In order to send requests to the server, the *Prefetch Manager* has several *User Request* threads created and waiting for a new request (the total number is defined in the configuration file and, according with the HTTP 1.1 specification [114], is set to 2).

After receiving a requested resource from the server, the *Prefetch Manager* checks if its URL is in the *Prefetch List* data store. In case it is, that URL is removed from the *Prefetch List* data store and is inserted into the *Fetched Objects* data store, in order to avoid prefetching that resource at a later time. The *Prefetch Manager* also checks if the request queue is temporarily empty. If so, the *Prefetch Manager* allows prefetching requests to be sent to the server, until a new user request arises. When a new client request is inserted into the queue, the *Prefetch Manager* indicates that the prefetched downloads must be cancelled and clears the *Hint List* data store.

The *User Requests* block receives requests from the *Prefetch Manager* and redirects them to the *Requests* block. When a requested resource is received from the server, the *User Requests* block inserts the prefetching hints sent in the response headers into the *Prefetch List* data store, and the URL of the resource into the *Fetched Objects* data store, in order to avoid prefetching requests for the resource later.

The *Request* block is a common interface (used both by the *User Requests* and *Prefetch Requests* blocks) for communicating with the web server. This block handles the communication sockets at low level. Since in the current implementation the client is configured to use HTTP/1.1, the *Request* block maintains persistent connections with the server [114].

When the simulator is running, it outputs information about each request to the web server as well as the client request hits to the prefetched resource cache to a log file. Analysis of the log file can then be performed at post-simulation time.

In the next section we describe the deduplication technique we designed and implemented to extend this web prefetching system.

3.2 HTTP traffic deduplication system

Due to its main advantages over other deduplication techniques, that we have already mentioned in section 2.4.2, we decided to implement a deduplication technique based on the dedupHTTP [112] system.

The dedupHTTP system has two implementation points, the client deduplication module (CDM) and the server deduplication module (SDM). The CDM and SDM run as modules inside the client browser and web server, respectively.

The CDM stores only the latest version of each resource. It indexes each resource version by an unique identifier, which is assigned to it by the SDM. The CDM maps the resource versions by host domain. When the client module makes a request for a resource, it first 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. At request time, the CDM serializes these identifiers into a byte array and sends them to the server in a custom HTTP request header.

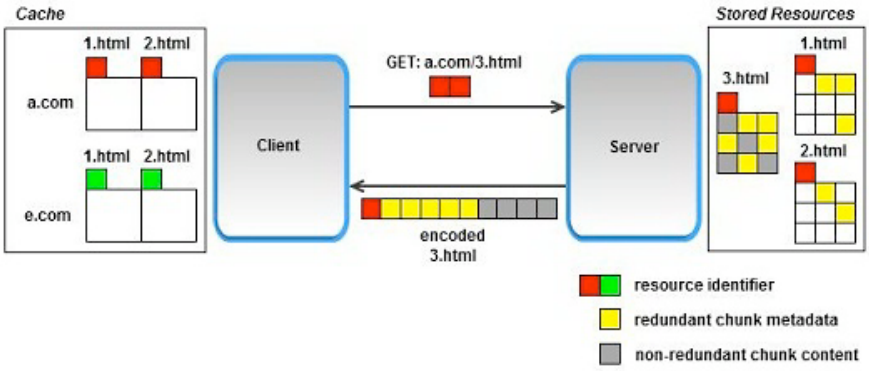


Fig. 7- Architecture of dedupHTTP[112].

When the SDM receives a request for a given resource, it retrieves the reference resource identifiers sent by the CDM in a custom request header. The SDM then fetches the resource from the web server. After receiving the full response headers and data, the SDM attributes a new identifier to the resource. It then divides the resource data into chunks and stores the chunk's meta-information in a data store. In this data store the SDM keeps the meta-information for all the chunks of all resource versions, indexed by chunk hash.

The SDM then goes through all the chunks of the resource. For each of the resource chunks, it searches within the reference resources for chunks with identical hash. If for some chunk no match is found in the reference resources, the SDM also looks it up in the current response's resource chunks. Therefore, redundancy detection is performed not only across the resources at the CDM's cache but also inside the resource being sent to the CDM.

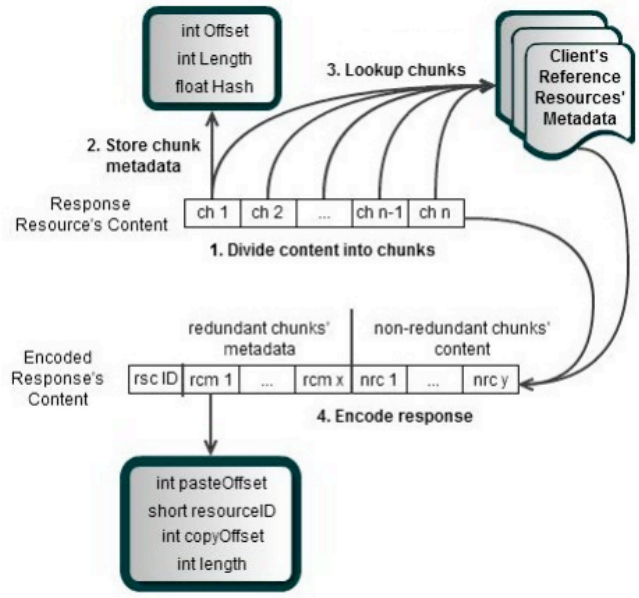


Fig. 8- Diagram of the SDM algorithm [112].

The SDM then composes the final response to send to the CDM. The response begins with a metadata section. The size of the metadata section (in bytes) is stored in a custom HTTP response header. The metadata section's content starts with the response's resource identifier. Then a four-part tuple is sent for each chunk found on a reference resource. Each tuple contains the information needed to allow the CDM to find each redundant chunk in its cache:

- the offset in the current response where the chunk is to be appended;
- the resource identifier where the CDM can find the chunk content;
- the offset in the resource where the chunk content can be found;
- the length of the chunk;

The tuples are arranged in the same order as the corresponding chunks in the original response. At the end of the metadata block, the non-redundant response content is appended, in the same order in which it was present in the original response. Because the order of the tuples and non-redundant chunks is maintained, the CDM only needs the tuple's first offset to know where to append a redundant chunk with a non-redundant one. A diagram with the SDM algorithm is presented in Fig. 8.

When the CDM receives the response, it goes through all the metadata in order to reconstruct the original resource. For this purpose, it copies the chunks referenced by the tuples to the final response (from the locally cached resources), and copies the non-redundant content from the received response to the final response, in the corresponding order. Therefore, the CDM does not have to store meta-information about any chunks. Also, no hashes need to be sent nor stored on the CDM.

3.2.1 Algorithmic complexity

Fig. 9 shows the pseudocode for the SDM hash lookup step [112]. We define:

- *n* as the number of chunks of the resource currently being processed;

- *m* as the number of reference resources whose identifiers have been sent from the CDM (also including the current resource);

- *p* as the number of chunks in the reference resource where the current chunk is being looked up (the complexity of the lookup in the chunks hash table is *O* (*p*/*k*), where *k* is a constant).

The algorithmic complexity of the SDM hash lookup step is *O* (*n***m***p*) in the worst case, when none of the chunks exists on any of the reference resources.

n and *p* are a function of the chunk size and the total size of the current and reference resources, respectively. If *m* is fairly small (*e.g.*, an end user with a low number of resources in its cache) then, for small chunk sizes and big files, we will have *n***p* >> *m* and chunk size will be the dominant parameter in defining algorithmic complexity.

```
{resource data is divided in chunks according with algorithm in section 3.2.2}
resource←newResource(response.getContent())
storedResources.add(resource)
referenceResourceIDs←request.getHeader("X-vrs")
metadata←newMetadata()
content←newContent()
for all chunkHash in resource do
   foundChunk←false
   for all referenceResourceID in referenceResourceIDs do
      referenceResource←storedResources.getResource(referenceResourceID)
      chunk← referenceResource.getChunk(chunkHash)
      if chunk != null then
         metadata.append(chunk.getMetadata())
         foundChunk←true
         break
      end if
   end for
   if foundChunk == false then
      content.append(resource.getContent(chunkHash))
   end if
end for
response.addHeader("X-mtd", metadata.size)
encodedResponse←resource.ID + metadata + content
response.setHeader("Content-Length", encodedResponse.size)
response.setContent(encodedResponse)
```

Fig. 9- Pseudocode for the SDM hash lookup step (adapted from [112]).

3.2.2 Chunk division algorithm

For every new resource version sent by the server in response to a client request, the SDM divides that resource into indexed data chunks.

For chunking of the resources in the SDM, we used an algorithm identical to LBFS [97]. In it, per-byte hashes of the resource data are created, using the rolling hash function of Karp-Rabin

fingerprinting [105], with a fingerprint size of 48 bytes. Chunk boundaries- *breakpoints*- are defined in the following way: when the low-order 13 bits of a region's fingerprint are equal to an arbitrary predefined value, the region constitutes a breakpoint. To avoid pathological cases, a minimum and maximum chunk size are imposed. Finally, each chunk is hashed using the 160 bit SHA1 [115] cryptographic hash function.

3.3 Architecture of the full system

To implement the full system, we extended the web prefetching framework described in section 3.1. We respected the existing architecture of the simulator and extended it by introducing additional functional modules both in the surrogate and the client. Below we describe in detail each of the main components of the full system.

3.3.1 Surrogate module

The block diagram of the surrogate module is presented in Fig. 10.

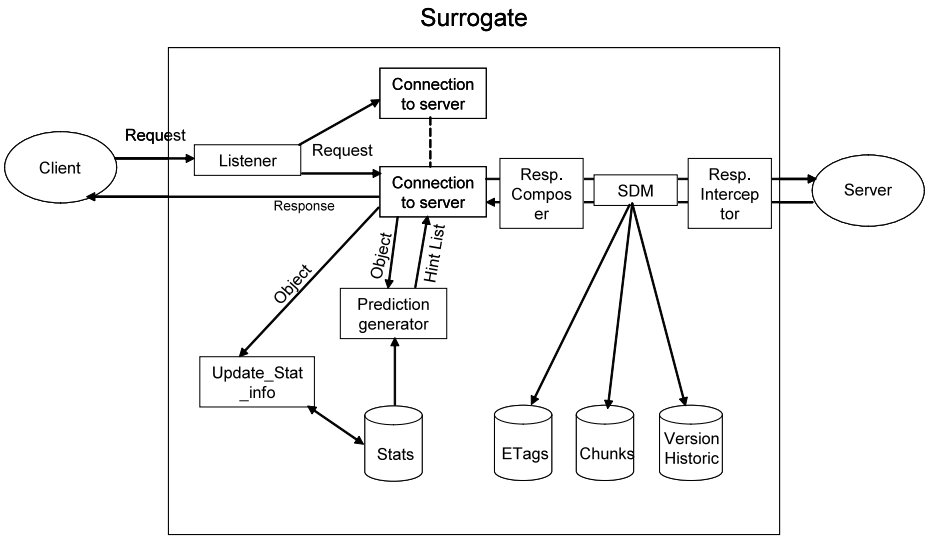


Fig. 10- Block diagram of the surrogate module.

In the simulator framework, as described above in section 3.1.1, the *Connection to server* block of the surrogate handles client-server connections. In particular, in the processing of the response, the *Connection to server* block first receives all the original HTTP response headers. Then it piggybacks

the hintlist to the original headers and sends them to the client. Afterwards it transfers all the resource data received from the server to the client, without any changes.

In our case, we need to introduce modifications in the HTTP response headers that involve not only concatenating additional headers to the existing ones, but also modifying some of the existing header values (*e.g.*, we need to modify the 'Content-Length' header value). Furthermore, because we perform deduplication processing on the resource data (*i.e.*, the *entity content*), we first need to receive the complete entity content data from the server. Then this data will be subject to deduplication processing (which may modify it). Only after this process is concluded will the modified entity content data block be sent to the client.

Therefore, we changed the way the HTTP response is processed in the surrogate module: we introduced a new functional block, the *Response Interceptor* (Fig. 10).

The *Response Interceptor* block intercepts the original HTTP response, coming from the server. It reads the full response headers and stores them in a temporary buffer for further processing. Afterwards, the *Response Interceptor* parses the headers:

- it checks for the existence of the 'ETag' header (resource unique identifier, generated by the server [114]) and in case it is present, gets its value;

- it checks which of the 'Content-Length' or 'Transfer-Encoding' headers is present (the presence of the 'Transfer-Encoding' header signals that the entity content data block will be sent in a chunked transfer [114]; if, instead, the 'Content-Length' header is present then a normal data transfer (*i.e.*, continuous) is performed by the server and the header value is the length of the data block transferred [114]);

Then, the *Response Interceptor* receives the entity content data block from the server, taking into account if the transfer from the server is made in a normal or chunked way. It stores the complete resource entity data in a buffer for further processing.

Finally, the *Response Interceptor* retrieves from the custom header sent by the client ("X-vrs") the resource identifiers of the reference resources the client has in its cache; these identifiers are stored in an array.

The *Response Interceptor* block then delivers the buffers with the stored response headers and the resource entity data, and the array with the reference resources identifiers to the *Surrogate Deduplication Module* (*SDM*) block.

The *SDM* maintains three data stores:

- *ETags*: stores the unique ETag resource identifiers, indexed by resource version;

- *Chunks*: stores chunks meta-information, for the all chunks of all resource versions processed by the *SDM*, indexed by chunk hash value;

- *Versions Historic*: stores the linked-lists of chunks for all resource versions (that allow going through all the chunks of a given resource version in an ordered manner), indexed by resource version;

The *SDM* performs deduplication processing on the resource entity content data, delivered to it by the *Response Interceptor* block. The *SDM* first checks in the *ETags* data store if the resource ETag is new. If so, it generates a new resource identifier for the new resource version and stores it in *ETags*.

Then it chunks the resource, using the chunking algorithm described below in section 3.4. As a result, a linked-list of chunks is obtained, which respects the order in which the chunks appear in the original resource. This linked list is then stored in the *Versions Historic* data store. In case the ETag already exists, this means that the resource version has previously been processed by the *SDM*. Therefore, there is no need to chunk it: the *SDM* just retrieves the chunks linked-list of the resource version from the *Versions Historic* data store.

Afterwards, the *SDM* executes the deduplication algorithm, described in section 3.2. It keeps two buffers where it builds the metadata and the non-redundant content sections that will be sent to the client. The *SDM* goes through the linked-list of chunks of the resource version. For each of these chunks, the *SDM* attempts to find a chunk with matching hash within the reference resources of the client. If no match is found, then the *SDM* also performs a lookup in the current resource version chunks that have already been processed (*i.e.*, self-redundancy cases).

The cases where a match is found correspond to redundant chunks. For these, the *SDM* adds the chunk metadata to the metadata buffer. For non-redundant chunks, the entity content data of the chunk is added to the non-redundant content buffer. In this way, the *SDM* builds the metadata and the non-redundant content blocks of the final response. The *SDM* then signals the *Response Composer* block that deduplication processing is finished.

The *Response Composer* prepares and sends the deduplicated response. It composes the response headers:

- it updates/creates the 'Content-Length' header with the new resource entity content data length;
- adds two new headers: 'X-vrs', containing the resource version identifier; 'X-mtd', containing the metadata section length;

The *Response Composer* then composes the new entity content data block, by appending the non-redundant content block to the end of the metadata block. Finally, it sends the newly composed full response, comprised of the headers and entity content data block.

The newly composed response is then intercepted by the *Connection to server* block which, maintaining its functional role, piggybacks the hintlist on the headers and leaves unchanged the entity data block. The fully composed HTTP response is then finally sent to the client.

All other blocks left unmentioned in this section have kept their functional roles, as described previously in section 3.1.1.

3.3.2 Client module

The block diagram of the client module is presented in Fig. 11.

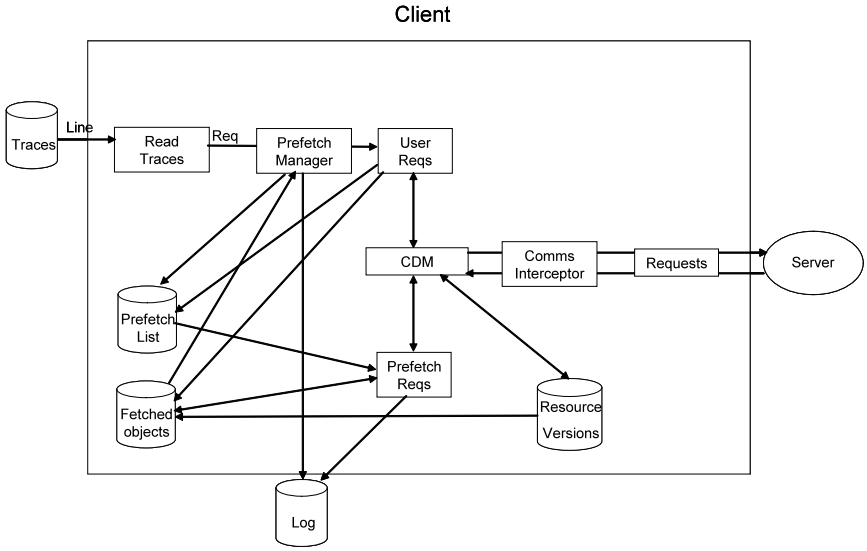


Fig. 11-Block diagram of the client module.

We added a new data store to the client module: *Resource Versions*, which stores the reconstructed resources entity content data, indexed by resource identifier. In practice this acts as the client browser cache. We synchronize the information of the resources contained in this data store with the information in *Fetched Objects* which, as mentioned above in section 3.1.1, only stores references to resources, not entity content data. The *Resource Versions* data store is maintained by the *Client Deduplication Module* (*CDM*) block.

The *CDM* intercepts client HTTP requests to the server (whether sent by the *User Requests* or *Prefetch Requests* blocks). It queries the *Resource Versions* data store in order to get the list of resource identifiers of all the resource versions contained in the client cache. The *CDM* then notifies the *Communications Interceptor* block, which composes the custom 'X-vrs' header with this information. Therefore, the 'X-vrs' header contains the resource identifiers for the client reference resources. The *Communications Interceptor* then appends this header to the HTTP request headers and directs the request to the *Requests* block, which sends it to the server.

When the HTTP response is received from the surrogate module, The *Communications Interceptor* parses the headers and extracts the resource version identifier and the metadata block length from the 'X-vrs' and the 'X-mtd' headers, respectively. It then splits the received data into three different buffers, for further processing: one buffer stores the headers, the others store the metadata block and the non-redundant content block, respectively. The *Communications Interceptor* then delivers all this data to the *CDM*.

The *CDM* then allocates a buffer in which it will reconstruct the original resource sent by the server. It performs the resource reconstruction algorithm previously described in section 3.2. The *CDM* goes through the metadata block and, for each chunk referenced by a tuple, it copies the corresponding entity content data stored in the *Resource Versions* data store to the reconstruction buffer. When a gap appears between tuple referenced chunks, the *CDM* copies the data from the non-redundant content block, maintaining the original order of the chunks. In this way, the original resource version is reconstructed.

The *CDM* then stores the new resource version in the *Resource Versions* data store. If a previous version of the same resource was kept in *Resource Versions*, it is then evicted. In this way, the client only keeps the most recent version of each resource, at all times.

All other blocks left unmentioned in this section have kept their functional roles, as described previously in section 3.1.2.

4 Evaluation

4.1 Metrics

The main objective of the present work was to implement a system that improved user-perceived latency in Web navigation, by combining prefetching and deduplication techniques.

In order to evaluate the performance of the implemented system, we use the following metrics:

- *Latency*: we define request latency as the time measured between the start point of a client HTTP GET request for a given resource, until the full reconstruction of that resource on the client module. We report the *latency per resource* normalized value, relative to the standard HTTP transfer. Standard HTTP transfer values correspond to operation of the simulator with both prefetching and deduplication turned off (PFOFF_DDPOFF).

- *Bytes saved*: we define the saving in bytes transferred per request as the sum of redundant chunks byte sizes, *i.e.* the sum of the byte sizes of the data chunks that are not sent from surrogate to client, but rather obtained from the client reference resources. We report the *bytes saved* normalized value, relative to operation with prefetch settings identical to the prefetching setting of the measurement (*e.g.*, prefetching turned on and deduplication turned on relative to prefetching turned on and deduplication turned off- PFON_DDPON relative to PFON_DDPOFF).

4.2 Experimental methodology

The experimental setup used in our tests consisted of two machines:

- one machine performing the role of the web client, which runs the client module (prefetch engine+client deduplication module); this machine contains also the traces that simulate the user's web accesses, which are fed to the simulator framework;

- a second machine performs the role of the surrogate (prediction engine+server deduplication module), attached to the web server; therefore, this machine is running both an instance of the simulator's surrogate module and an instance of a HTTP server(Apache); the HTTP server has stored locally all the workloads files;

The technical specs for both machines are:

- Client: Intel Core2Duo@2.66GHz processor, 4GB RAM; OS: Linux Mint 14-32b

- Server: Intel Core i5-2450M@2.5GHz, 6GB RAM; OS: Linux Mint 14-32b

The experiments were run in a LAN, with a bandwidth of 54Mb/s. For the tests with constrained bandwidth, we simulated the use of Bluetooth, i.e. effective bandwidth of 2.1Mbs. To force the bandwidth to this maximum value, we used the 'tc' [118] utility by adjusting parameters in its configuration file, and we monitored the effective bandwidth value using the 'iperf' [117] utility.

The workloads used in the various tests were obtained by downloading the files of a typical news website (www.dn.pt), up to 3 levels of depth. In the experiments we request all the workload files, which amounts to approx. 20MB of files transferred between client-server. Workload statistical information is presented in Table 1.

Client requests are automated using the wget utility [116]. In order to simulate the web navigation by a real user, introducing idle time between user requests to allow prefetching actions, we used the wget switches '-w10' and '--randomwait', which introduces a random delay of [5, 15] seconds between requests.

All the tests reported were performed 5 times for each experimental condition set, the results presented correspond to the average of those trials.

Workload file size data	
298	min (B)
29672	avg (B)
214595	max (B)
36905	stdev (B)
19.7	total (MB)

Table 1- Workload characterization.

4.3 Results

The first tests we performed regarded the deduplication algorithm: previously reported results [112] showed a dependency of the redundancy detection efficiency on chunk size. Since we implemented a similar deduplication algorithm, we wanted to confirm if this relationship was still true when using prefetching combined with deduplication.

We ran tests with prefetching (PF) and deduplication (DDP) both turned on (PFON-DDPON), with several average chunk sizes: 32, 64, 128, 256, 512, 1024 and 2048 bytes. Chunk size varies around the reference average value, therefore in all cases we force a minimum size equal to average chunk size / 4 and a maximum size equal to average chunk size * 2 (*e.g.*, for an expected average chunk size of 256 bytes, the minimum chunk size will be 64 bytes and the maximum chunk size will be 512 bytes). The results are presented below, in Fig. 12 and Fig. 13:

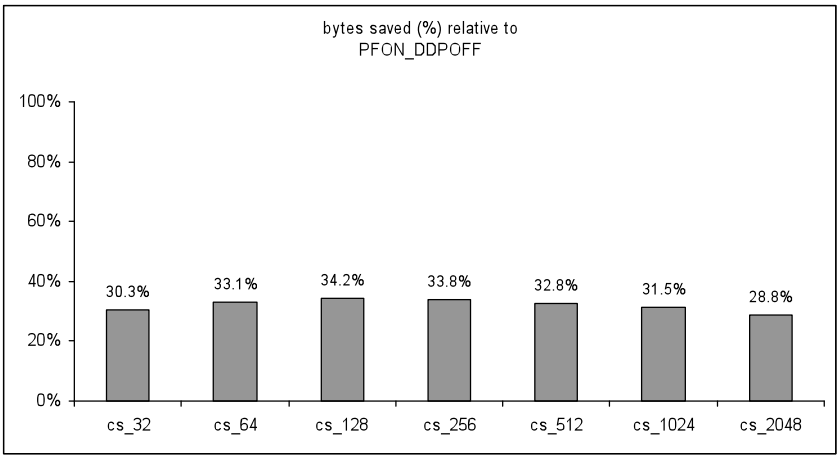


Fig. 12- Graph of bytes saved for PFON_DDPON, relative to PFON_DDPOFF vs Chunk Size (Bw=54Mbs).

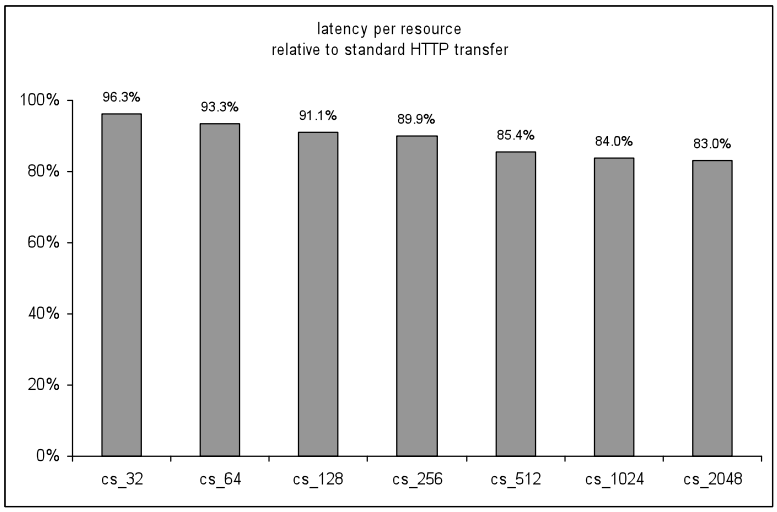


Fig. 13- Graph of latency per resource, relative to standard HTTP transfer vs Chunk Size (PFON_DDPON; Bw=54Mbs).

The graph in Fig. 12 shows a curve of percentage of bytes saved as a function of chunk size which is in agreement with [112]. These results show that the best redundancy detection is obtained for a chunk size of 128 bytes, corresponding to a saving of 34.2% in the amount of bytes.

The latency results in Fig. 13 show an inverse relation between latency and chunk size. This result can be explained by the computational overhead introduced by the deduplication processing: with increasing chunk size, the number of chunks decreases, and so does the deduplication algorithm local execution time (please check section 3.2 above).

Due to the higher savings obtained in the amount of bytes, for the remaining tests with deduplication turned on, we used a chunk size of 128 bytes.

We then tested the following running conditions: PFOFF-DDPOFF vs PFOFF-DDPON and PFON-DDPOFF vs PFON-DDPON. Results for bytes saved and latency for PFOFF-DDPOFF vs PFOFF-

DDPON are shown in Fig. 14 and Fig. 15, respectively; Fig. 16 and Fig. 17 present the results obtained for PFON-DDPOFF vs PFON-DDPON.

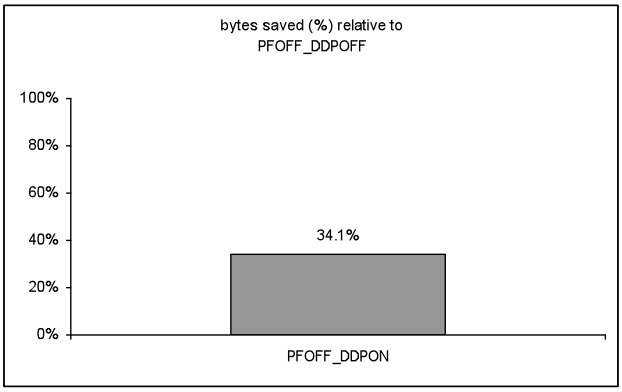


Fig. 14- Graph of bytes saved for PFOFF_DDPON, relative to PFOFF_DDPOFF (Bw=54Mbs).

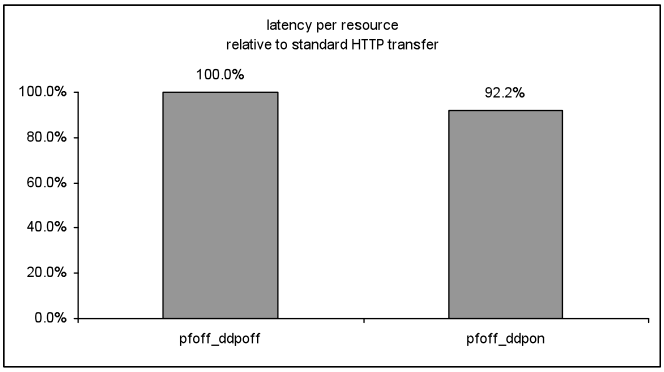


Fig. 15- Graph of latency per resource, relative to standard HTTP transfer: PFOFF_DDPOFF vs PFOFF_DDPON (Bw=54Mbs).

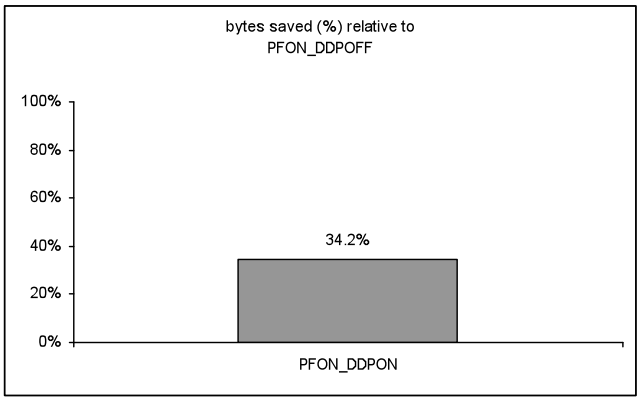


Fig. 16- Graph of bytes saved for PFON_DDPON, relative to PFON_DDPOFF (Bw=54Mbs).

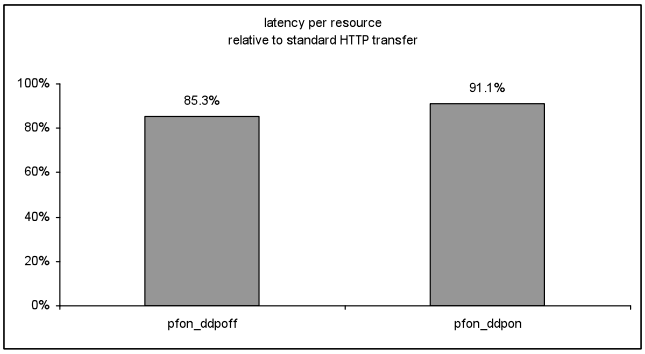


Fig. 17- Graph of latency per resource, relative to standard HTTP transfer: PFON_DDPOFF vs PFON_DDPON (Bw=54Mbs).

The results for normalized bytes saved in Fig. 14 and Fig. 16 are coherent and show that deduplication allows to obtain approximately 34% in the amount of bytes saved.

The latency results in Fig. 15 show that deduplication allows to reduce the latency in approx. 8% relative to the standard HTTP transfer. However, the comparison in the graph of Fig. 17 shows that, when prefetching is used, the savings in latency are clearly higher when deduplication is turned OFF than when deduplication is used (14.7% and 8.9% saving in latency, respectively). This difference is significant and may be justified by the computational overhead introduced by the deduplication processing.

These results show that when both prefetching and deduplication are used, deduplication allows to obtain savings in the amount of bytes, but at the cost of having an increased latency relative to the case when only prefetching is turned on.

In order to evaluate the effects of constraining the bandwidth in the achievable redundancy values, we performed comparison tests of PFON_DDPON operation using LAN conditions (54Mbs) vs Bluetooth conditions (2.1Mbs). The results are presented in Fig. 18 and Fig. 19:

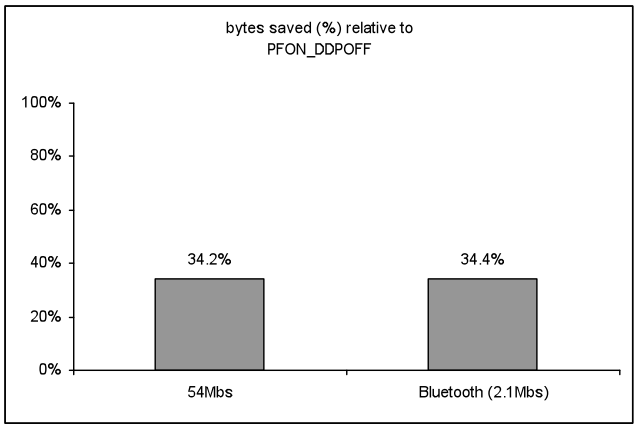


Fig. 18- Graph of bytes saved for PFON_DDPON, relative to PFON_DDPOFF: 54Mbs vs 2.1Mbs.

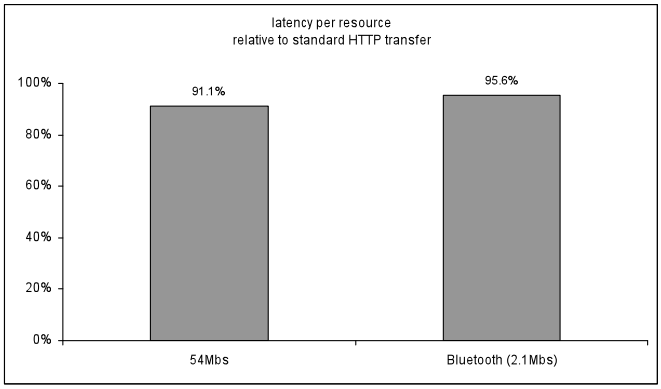


Fig. 19- Graph of latency per resource, relative to standard HTTP transfer: 54Mbs vs 2.1Mbs (PFON_DDPON).

The values obtained in both cases for bytes saved are coherent with previous measurements. The reduction achieved in latency per resource in the Bluetooth case is half of that obtained for LAN conditions (4.4% and 8.9% reduction in latency, respectively), which can be taken as a consequence of operating under constrained bandwidth. A lower number of prefetching events may eventually occur as a consequence of the reduced bandwidth, since prefetching is highly dependent on idle times between user requests. Since the increased latency means that requests take a longer time to be completed, in consequence idle times may be reduced, thereby reducing the opportunity to start prefetching requests.

5 Conclusions

In the present work we evaluated the possibility of combining deduplication and prefetching techniques to improve the user-perceived latency in the Web. We developed and implemented a system that combines the use of Web prefetching and deduplication techniques with the main objective of improving user-perceived latency in Web navigation.

Our results show that significant reductions in the volume of bytes transferred relative to a normal HTTP transfer are achievable when the two techniques are used in combination, with savings of approximately 34%. Although the gains in latency were more modest, nevertheless we were able to achieve a reduction of approximately 8% in user-perceived latency. Operation under constrained network conditions impacts the attainable gains. Still, over 4% reduction in the latency was achieved.

The results obtained when using both prefetching and deduplication show high gains in the amount of bytes saved (34.2%). Nevertheless, when both techniques are used simultaneously, the latency reduction (8.9%) is lower than in the case where only prefetching is used (14.7%). This is attributable to the computational overhead introduced by the deduplication algorithm.

These results lead us to conclude that the combination of prefetching and deduplication techniques should be applied carefully to Web navigation. In cases where savings in the amount of bytes transferred are critical (such as pay-per-byte Web access), if the impact of approximately 6% on the latency is considered tolerable, then it is a good option to combine both techniques. If however, the most critical for the user is to minimize latency at all costs, with no concerns regarding the amount of bytes transmitted, then the best option is to use just prefetching.

5.1 Future Work

The present work showed a large impact of the deduplication algorithm local execution time in the attainable gains in latency. One possible path for improvement would be to optimize the deduplication algorithm execution time, in order to lower its impact on the latency. One possibility would be to use a larger chunk size, since we have seen this can be the most important factor affecting the execution time. These tests would have to take into account the impact of using a larger chunk size on the detectable redundancy.

Also, we should keep in mind that the present tests were performed in a LAN environment. We would like to perform tests in the internet environment, in order to evaluate the impact of internet delays in the performance of the system.

Another test we would like to perform would a so called 'warm' start of the surrogate. This means to preload in the surrogate all the chunks of the resources it is expected that the client will request. In this way, when the client makes the requests for the resources, the surrogate will no longer need to perform the chunking phase, and only the lookup phase will impact the execution time. This would allow to assess the possible positive impact on latency of the 'warm' start condition.

References

[1] http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white paper c11-520862.html.

[2] Azer Bestavros. *WWW Traffic Reduction and Load Balancing through Server-Based Caching*. IEEE Concurrency, vol. 5, no. 1, pages 56–67, 1997.

[3] Charu C. Aggarwal, Joel L. Wolf and Philip S. Yu. *Caching on the World Wide Web*. IEEE Transactions on Knowledge and Data Engineering, vol. 11, no. 1, pages 95–107, 1999.

[4] Shudong Jin and Azer Bestavros. *Popularity-Aware GreedyDual- Size Web Proxy Caching Algorithms*. In Proceedings of the 20th International Conference on Distributed Computing Systems, Taipei, Taiwan, 2000.

[5] Kirk L. Johnson, John F. Carr, Mark S. Day and M. Frans Kaashoek. *The measured performance of content distribution networks*. Computer Communications, vol. 24, no. 2, pages 202–206, 2001.

[6] Arun Iyengar, Erich Nahum, Anees Shaikh and Renu Tewari. *Enhancing Web Performance*. In Proceedings of the IFIP World Computer Congress, Montreal, Canada, 2002.

[7] Evangelos Markatos and Catherine Chronaki. *A Top-10 Approach to Prefetching on the Web*. In Proceedings of the INET'98, Geneva, Switzerland, 1998.

[8] Li Fan, Pei Cao, Wei Lin and Quinn Jacobson. *Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance*. In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 178–187, Atlanta, USA, 1999.

[9] Persistence Software, Inc. Persistence home page. http://www.persistence.com/, 2010.

[10] Akamai Technologies, Inc. Akamai home page. http://www.akamai.com/, 2010.

[11] Mark Crovella and Paul Barford. *The network effects of prefetching*. In Proceedings of the IEEE INFOCOM'98 Conference, San Francisco, USA, 1998.

[12] Azer Bestavros. *Using Speculation to Reduce Server Load and Service Time on the WWW*. In Proceedings of the 4th ACM International Conference on Information and Knowledge Management, Baltimore, USA, 1995.

[13] Xin Chen and Xiaodong Zhang. *Popularity-Based PPM: An Effective Web Prefetching Technique for High Accuracy and Low Storage*. In Proceedings of the International Conference on Parallel Processing, Vancouver, Canada, 2002.

[14] Darin Fisher and Gagin Saksena. *Link prefetching in Mozilla: A Server driven approach*. In Proceedings of the 8th International Workshop on Web Content Caching and Distribution (WCW 2003), New York, USA, 2003.

[15] Andrzej Sieminski, *The impact of Proxy caches on Browser Latency*. International Journal of Computer Science & Applications, Vol.II, No.II, pp. 5-21

[16] Brian Davison, *The Design and Evaluation of Web Prefetching and Caching Techniques*, PhD Thesis, 2002

[17] Neil Spring and David Wetherall, *A Protocol-Independent Technique for Eliminating Redundant Network Traffic*, In Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 2000

[18] Ashok Anand, Chitra Muthukrishnan, Aditya Akella, Ramachandran Ramjee, *Redundancy in Network Traffic: Findings and Implications*, In Proceedings of SIGMETRICS/Performance'09, Seattle, USA, 2009

[19] Sean Rhea, Kevin Liang, Eric Brewer, *Value-Based Web Caching*, In Proceedings of the 12th international conference on World Wide Web, Budapest, Hungary, 2003

[20] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. *Potential benefits of delta encoding and data compression for http*. In Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication, SIGCOMM '97, pages 181–194, New York, NY, USA, 1997. ACM.

[21] T. Kelly and J. Mogul. *Aliasing on the world wide web: Prevalence and performance implications*, In Proceedings of the 11th international conference on World Wide Web, May 07-11, 2002, Honolulu, Hawaii, USA, 2002.

[22] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk, L. Masinter, P. Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol — HTTP/1.1*. RFC 2616, http://ftp.isi.edu/in-notes/rfc2616.txt, June 1999.

[23] Fredrick J. Hill and Gerald R. Peterson. Digital Systems: Hardware Organization and Design. John Wiley & Sons, New York, 1987. Third Edition.

[24] M. Morris Mano. Computer System Architecture. Prentice-Hall, Englewood Cliffs, NJ, 1982. Second Edition.

[25] Miles J. Murdocca and Vincent P. Heuring. Principles of Computer Architecture. Prentice Hall, 1999.

[26] Alan Jay Smith. *Cache memories*. ACM Computing Surveys, 14(3):473– 530, September 1982.

[27] Venkata N. Padmanabhan and Jeffrey C. Mogul. *Improving HTTP latency*. In Proceedings of the Second International World Wide Web Conference: Mosaic and the Web, pages 995–1005, Chicago, IL, October 1994.

[28] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. *Network performance effects of HTTP/1.1, CSS1, and PNG*. Computer Communications Review, 27(4), October 1997. In Proceedings of SIGCOMM '97. Also available as W3C NOTE-pipelining-970624.

[29] Ramón Cáceres, Fred Douglis, Anja Feldmann, Gideon Glass, and Michael Rabinovich. *Web proxy caching: The Devil is in the details*. Performance Evaluation Review, 26(3):11–15, December 1998. In Proceedings of the Workshop on Internet Server Performance.

[30] Anja Feldmann, Ramón Cáceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. *Performance of Web proxy caching in heterogeneous bandwidth environments*. In Proceedings of IEEE INFOCOM, pages 106–116, New York, March 1999.

[31] Bradley M. Duska, David Marwood, and Michael J. Feely. *The measured access characteristics of World-Wide-Web client proxy caches*. In Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97), pages 23–36, Monterey, CA, December 1997. USENIX Association.

[32] Linda Tauscher and Saul Greenberg. *How people revisit Web pages: Empirical findings and implications for the design of history systems*. International Journal of Human Computer Studies, 47(1):97–138, 1997.

[33] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. *A hierarchical Internet object cache*. In Proceedings of the USENIX Technical Conference, pages 153–163, San Diego, CA, January 1996.

[34] Duane Wessels and Kimberly Claffy. *Internet Cache Protocol (ICP), version 2*. RFC 2186, http://ftp.isi.edu/in-notes/rfc2186.txt, September 1997.

[35] Alex Rousskov and Duane Wessels. *Cache digests*. Computer Networks and ISDN Systems, 30(22–23):2155–2168, November 1998.

[36] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. *On the scale and performance of cooperative Web proxy caching*. Operating Systems Review, 34(5):16–31, December 1999. In Proceedings of the 17th Symposium on Operating Systems Principles.

[37] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. *Summary cache: A scalable wide-area Web cache sharing protocol*. Computer Communication Review, 28(4), October 1998. In Proceedings of ACM SIGCOMM.

[38] James Gwertzman and Margo Seltzer. *World-Wide Web cache consistency*. In Proceedings of the USENIX Technical Conference, pages 141– 151, San Diego, CA, January 1996. USENIX Association.

[39] Chengjie Liu and Pei Cao. *Maintaining strong cache consistency for the World-Wide Web*. IEEE Transactions on Computers, 47(4):445–457, April 1998.

[40] Dan Li, Pei Cao, and M. Dahlin. *WCIP: Web cache invalidation protocol*. Internet-Draft draft-danli-wrec-wcip-01.txt, IETF, March 2001. Work in progress.

[41] Pei Cao, Jin Zhang, and Kevin Beach. *Active Cache: Caching dynamic contents on the Web*. Distributed Systems Engineering, 6(1):43–50, 1999.

[42] SpiderCache Inc. http://www.spidercache.com

[43] Port80 Software. https://secure.xcache.com/

[44] Barron C. Housel and David B. Lindquist. *WebExpress: A system for optimizing Web browsing in a wireless environment*. In Proceedings of the Second Annual International Conference on Mobile Computing and Networking (MobiCom '96), pages 108–116, Rye, New York, November 1996. ACM/IEEE.

[45] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. *Potential benefits of delta-encoding and data compression for HTTP*. In Proceedings of ACM SIGCOMM, pages 181–194, Cannes, France, September 1997. An extended and corrected version appears as Research Report 97/4a, Digital Equipment Corporation Western Research Laboratory, December, 1997.

[46] Oracle Corporation and Akamai Technologies. Edge side includes home page. Online at http://www.edge-delivery.org/, 2001.

[47] Fred Douglis, Antonio Haro, and Michael Rabinovich. *HPP: HTML macro-preprocessing to support dynamic document caching*. In Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97), pages 83–94, Monterey, CA, December 1997. USENIX Association.

[48] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey C. Mogul. *Rate of change and other metrics: A live study of the World Wide Web*. In Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97), December 1997.

[49] Elizabeth Shriver, Christopher Small, and Keith Smith. *Why does file system prefetching work?*. In Proceedings of the 1999 USENIX Annual Technical Conference, pages 71–83, Monterey, CA, June 1999.

[50] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Computer Systems Laboratory, Stanford University, March 1994.

[51] Venkata N. Padmanabhan and Jeffrey C. Mogul. *Using predictive prefetching to improve World Wide Web latency*. Computer Communication Review, 26(3):22–36, July 1996. In Proceedings of SIGCOMM '96.

[52] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. *Exploring the bounds of Web latency reduction from caching and prefetching*. In Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97), December 1997.

[53] http://richardyusan.wordpress.com/fasterfox-mod-by-richardyusan-speed-up/

[54] Rhino Software, Inc. http://www.allegrosurf.com

[55] Mark Crovella and Paul Barford. *The network effects of prefetching*. In Proceedings of the IEEE INFOCOM'98 Conference, San Francisco, USA, 1998.

[56] Azer Bestavros. *Using Speculation to Reduce Server Load and Service Time on the WWW*. In Proceedings of the 4th ACM International Conference on Information and Knowledge Management, Baltimore, USA, 1995.

[57] Venkata N. Padmanabhan and Jeffrey C. Mogul. *Using Predictive Prefetching to Improve World Wide Web Latency*. Computer Communication Review, vol. 26, no. 3, pages 22–36, 1996.

[58] Evangelos Markatos and Catherine Chronaki. *A Top-10 Approach to Prefetching on the Web*. In Proceedings of the INET' 98, Geneva, Switzerland, 1998.

[59] Li Fan, Pei Cao, Wei Lin and Quinn Jacobson. *Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance*. In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 178–187, Atlanta, USA, 1999.

[60] Xin Chen and Xiaodong Zhang. *Popularity-Based PPM: An Effective Web Prefetching Technique for High Accuracy and Low Storage*. In Proceedings of the International Conference on Parallel Processing, Vancouver, Canada, 2002.

[61] Dan Duchamp. *Prefetching Hyperlinks*. In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, USA, 1999.

[62] Yuna Kim and Jong Kim. *Web Prefetching Using Display-Based Prediction*. In Proceedings of the IEEE/WIC International Conference on Web Intelligence, Halifax, Canada, 2003.

[63] Kelvin Lau and Yiu-Kai Ng. *A Client-Based Web Prefetching Management System Based on Detection Theory*. In Proceedings of the Web Content Caching and Distribution: 9th International Workshop (WCW 2004), pages 129–143, Beijing, China, 2004.

[64] Christos Bouras, Agisilaos Konidaris and Dionysios Kostoulas. *Predictive Prefetching on the Web and Its Potential Impact in the Wide Area*. World Wide Web, vol. 7, no. 2, pages 143–179, 2004.

[65] Stuart Schechter, Murali Krishnan and Michael D. Smith. *Using Path Profiles to Predict HTTP Requests*. In Proceedings of the 7th International World Wide Web Conference, Brisbane, Australia, 1998.

[66] Themistoklis Palpanas and Alberto Mendelzon. *Web Prefetching Using Partial Match Prediction*. In Proceedings of the 4th International Web Caching Workshop, San Diego, USA, 1999.

[67] Heung Ki Lee, Gopinath Vageesan, Ki Hwan Yum and Eun Jung Kim. *A PROactive Request Distribution (PRORD) Using Web Log Mining in a Cluster-Based Web Server*. In Proceedings of the International Conference on Parallel Processing (ICPP'06), Columbus, USA, 2006.

[68] Josep Domènech, José A. Gil, Julio Sahuquillo and Ana Pont.DDG: *An Efficient Prefetching Algorithm for Current Web Generation*. In Proceedings of the 1st IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), Boston, USA, 2006.

[69] Darin Fisher and Gagin Saksena. *Link prefetching in Mozilla: A Server driven approach*. In Proceedings of the 8th International Workshop on Web Content Caching and Distribution (WCW 2003), New York, USA, 2003.

[70] Azer Bestavros. *Using speculation to reduce server load and service time on the WWW*. In Proceedings of CIKM'95: The Fourth ACM International Conference on Information and Knowledge Management, Baltimore, MD, November 1995.

[71] Mukund Deshpande and George Karypis. *Selective Markov models for predicting Web-page accesses*. In Proceedings of the First SIAM International Conference on Data Mining (SDM'2001), Chicago, April 2001.

[72] Doug Joseph and Dirk Grunwald. *Prefetching using Markov predictors*. Transactions on Computers, 48(2), February 1999.

[73] Peter L. Pirolli and James E. Pitkow. *Distributions of surfers' paths through the World Wide Web: Empirical characterization*. World Wide Web, 2:29–45, 1999.

[74] Zhong Su, Qiang Yang, Ye Lu, and Hong-Jiang Zhang. *WhatNext: A prediction system for Web request using n-gram sequence models*. In Proceedings of First International Conference on Web Information Systems and Engineering Conference, Hong Kong, June 2000.

[75] Themistoklis Palpanas and Alberto Mendelzon. *Web Prefetching Using Partial Match Prediction*. In Proceedings of the Fourth International Web Caching Workshop (WCW99), San Diego, CA, March 1999. Work in progress.

[76] Li Fan, Quinn Jacobson, Pei Cao, and Wei Lin. *Web prefetching between low-bandwidth clients and proxies: Potential and performance*. In Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99), Atlanta, GA, May 1999.

[77] Xing Dongshan and Shen Junyi. *A New Markov Model For Web Access Prediction*. Computing in Science and Engineering, vol. 4, no. 6, pages 34–39, 2002.

[78] Xin Chen and Xiaodong Zhang. *A Popularity-Based Prediction Model for Web Prefetching*. IEEE Computer, vol. 36, no. 3, pages 63–70, 2003.

[79] Philip Laird. *Discrete sequence prediction and its applications*. In Proceedings of the Tenth National Conference on Artificial Intelligence, Menlo Park, CA, 1992. AAAI Press.

[80] Philip Laird and Ronald Saul. *Discrete sequence prediction and its applications*. Machine Learning, 15(1):43–68, 1994.

[81] Brian D. Davison and Haym Hirsh. *Toward an adaptive command line interface*. Advances in Human Factors/Ergonomics: Design of Computing Systems: Social and Ergonomic Considerations, pages 505–508, San Francisco, CA, August 1997. Elsevier Science Publishers. In Proceedings of the Seventh International Conference on Human-Computer Interaction.

[82] Stuart Schechter, Murali Krishnan, and Michael D. Smith. *Using path profiles to predict HTTP requests*. Computer Networks and ISDN Systems, 30:457–467, 1998. In Proceedings of the Seventh International World Wide Web Conference.

[83] James E. Pitkow and Peter L. Pirolli. *Mining longest repeated subsequences to predict World Wide Web surfing*. In Proceedings of the Second USENIX Symposium on Internet Technologies and Systems, October 1999.

[84] Dan Duchamp. *Prefetching Hyperlinks*. In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, USA, 1999.

[85] Tamer I. Ibrahim and Cheng Zhong Xu. *Neural Nets based Predictive Prefetching to Tolerate WWW Latency*. In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems, Taipei, Taiwan, 2000.

[86] Ravi Kokku, Praveen Yalagandula, Arun Venkataramani and Michael Dahlin. *NPS: A Non-Interfering Deployable Web Prefetching System*. In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Palo Alto, USA, 2003.

[87] Bin Wu and Ajay D. Kshemkalyani. *Objective-Optimal Algorithms for Long-term Web Prefetching*. IEEE Transactions on Computers, vol. 55, no. 1, pages 2–17, 2006.

[88] Stuart Schechter, Murali Krishnan and Michael D. Smith. *Using Path Profiles to Predict HTTP Requests*. In Proceedings of the 7th International World Wide Web Conference, Brisbane, Australia, 1998.

[89] Heung Ki Lee, Gopinath Vageesan, Ki Hwan Yum and Eun Jung Kim. *A PROactive Request Distribution (PRORD) Using Web Log Mining in a Cluster-Based Web Server*. In ICPP(2006) 559-568.

[90] Edith Cohen and Haim Kaplan. *Prefetching the means for document transfer: a new approach for reducing Web latency*. Computer Networks, vol. 39, no. 4, pages 437–455, 2002.

[91] Google Web Accelerator. http://webaccelerator.google.com/.

[92] Google Search. http://google.com/intl/en/help/features.html.

[93] PeakSoft Multinet Corporation. http://www.peaksoft.com/

[94] Rizal Software Developers. https://www.rizalsoftware.com

[95] Robtex. http://www.robtex.com/

[96] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. *Delta algorithms: an empirical analysis*. ACM Transactions on software Engineering and Methodlogy, Apr 1998

[97] Athicha Muthitacharoen , Benjie Chen , David Mazieres, *A low-bandwidth network file system*, In Proceedings of the eighteenth ACM symposium on Operating systems principles, October 21-24, 2001, Banff, Alberta, Canada

[98] Navendu Jain , Mike Dahlin , Renu Tewari, *TAPER: tiered approach for eliminating redundancy in replica synchronization*, In Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, p.21-21, December 13-16, 2005, San Francisco, CA

[99] ESI - Edge Side Includes: Overview. http://www.esi.org/overview.html

[100] Vdelta differencing algorithm: http://tools.ietf.org/html/rfc3284

[101] Gaurav Banga , Fred Douglis , Michael Rabinovich, *Optimistic deltas for WWW latency reduction*, In Proceedings of the USENIX Annual Technical Conference, p.22-22, January 06-10, 1997, Anaheim, California

[102] Delta-encoding for HTTP: http://tools.ietf.org/html/rfc3229

[103] Implementations of RFC3229:http://wyman.us/main/2004/09/implementations.html

[104] Mun Choon Chan and Thomas Y. C. Woo, *Cache-based compaction: A new technique for optimizing web transfer*. In Proceedings of IEEE INFOCOM, March 1999

[105] Michael O. Rabin. *Fingerprinting by random polynomials*. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981

[106] Miroslav Ponec , Paul Giura , Herve Bronnimann , Joel Wein, *Highly efficient techniques for network forensics*. In Proceedings of the 14th ACM conference on Computer and communications security, October 28-31, 2007, Alexandria, Virginia, USA

[107] Saul Schleimer , Daniel S. Wilkerson , Alex Aiken, *Winnowing: local algorithms for document fingerprinting*. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, June 09-12, 2003, San Diego, California

[108] Utku Irmak , Torsten Suel, *Hierarchical substring caching for efficient content distribution to low-bandwidth clients*. In Proceedings of the 14th international conference on World Wide Web, May 10-14, 2005, Chiba, Japan

[109] Jeffery C. Mogul, Yee Man Chan, Terence Kelly, *Design, implementation, and evaluation of Duplicate Transfer Detection in HTTP*. In Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation, p.4-4, March 29-31, 2004, San Francisco, California

[110] Squid Web proxy cache: http://www.squid-cache.org/

[111] Josep Domènech, Ana Pont, Julio Sahuquillo and José A. Gil. *An Experimental Framework for Testing Web Prefetching Techniques*. In Proceedings of the 30th EUROMICRO Conference 2004, pp. 214–221, Rennes, France, 2004.

[112] Ricardo D. Filipe. *Deduplication in HTTP traffic- Redundancy detection and supression on the Web*. MSc thesis, Departamento de Engenharia Informática, Instituto Superior Técnico, October 2010.

[113] Yahiko Kambayashi and Kai Cheng, *LRU-SP: a size-adjusted and popularity-aware LRU replacement algorithm for web caching* in *Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000*, 2000, pp. 48–53.

[114] R. Fielding, J. Gettys, J. Mogul *et al*, *Hypertext transfer protocol–HTTP/1.1*, RFC 2616, pp. 1–114, 1999.

[115] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995

[116] http://www.gnu.org/software/wget/

[117] http://dast.nlanr.net/Projects/Iperf/

[118] http://lartc.org/manpages/tc.txt