Improving the Effectiveness of Web Caching*

Jean-Marc Menaud, Valérie Issarny, Michel Banâtre

INRIA
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
email: {menaud,issarny,banatre@irisa.fr}

Abstract: The bandwidth demands on the (WORLD-WIDE) WEB continue to grow at an exponential rate. It is thus becoming crucial to provide solutions improving the WEB latency. In that framework, the most promising low cost solution lies in the use of caches at the level of the clients, network and servers. Caching effectiveness then relies upon adequate cache management so as to keep in the cache the WEB objects that are the most likely to be re-accessed. However, the effectiveness of a single cache remains poor as it is in general no higher than 40%. One way to further improve caching effectiveness is thus to make caches cooperate so as to increase the probability of retrieving an object at the caching level. The cache cooperation protocol must then be such that it induces a negligible load for the network and cooperating caches.

This paper presents our solutions to improving the effectiveness of WEB caching concerning both cache management and cache cooperation. Regarding cache management, we propose two novel algorithms that exploit the latest results about WEB usage, enabling us to undertake replacement decisions that are more accurate than the one taken by existing algorithms. From the standpoint of cooperating caches, we propose a cooperation protocol, which minimizes the associated network bandwidth, processing load, and storage consumption among caches.

Key words: cache management, caching effectiveness, cooperative caching, replacement algorithm, simulation, (WORLD-WIDE) WEB.

1 Introduction

The ever-increasing popularity of the INTERNET is raising an urgent need for solutions aimed at masking the resulting traffic congestion and hence allowing improved response time. Two types of solutions have been explored: (*i*) increasing the bandwidth of the network links, which is of limited help due to the associated financial cost and technical problems; (*ii*) using caches over the INTERNET for replication of the most frequently accessed data. Although more affordable than the first solution, the actual benefit of caches for improving response time over the INTERNET still remains negligible [28]. Various approaches have been examined in order to increase the effectiveness of caches. These include the use of large caches and of more efficient cache management techniques. However, such approaches may only have a limited impact since most of the documents are accessed only once [2]. Furthermore, the use of large caches raises

*This paper is an extended version of the paper published in Proc. of DISC'98 (Formerly WDAG) – 12th International Symposium on Distributed Computing.

financial and technical problems. Other efforts have been focusing on the prefetching of data within caches but the resulting traffic overhead is too costly (*e.g.* [19]). Another way to increase cache effectiveness is not to propose solutions at the level of a single cache but rather to make a set of caches cooperate. Because they take advantage of the network topology, the most popular types of cooperative cache systems are the hierarchical and the transversal ones, which are both implemented by the Squid software [26] from the Harvest project [7]. This paper introduces solutions to increasing the effectiveness of Web caching, addressing support both at the level of a single cache and at the one of cooperating caches.

From the standpoint of cache management, we propose novel replacement algorithms that take advantage of a survey on Web usage (Section 3). In particular, it has been noticed that the wide majority of requests handled by a Web server relates to few documents. Thus, this raises the existence of hot Web objects, which may be exploited to favor the replacement of cold objects. In addition, regarding documents stored within Web caches, it appears that 40% of the objects are images. Given that an image may be degraded at low cost using Jpeg-like algorithms, without affecting considerably the resulting document readability, image degradation is an appealing alternative to object removal within a cache.

From the standpoint of cooperating caches, the major drawback of used techniques is that they introduce a significant bandwidth and workload overhead as they rely on the exchange of messages among all the cooperating caches to locate an object. Recent proposals address this performance penalty through a centralized knowledge of the cache system state, which induces a significant network and storage load for state management. On the other hand, a distributed solution to cooperative caching may be devised, yet introducing a bearable bandwidth and storage overhead. We propose such a solution, which is based on the adequate distribution of the state of the overall cooperating caches (Section 4).

Prior to present our solutions to improving effectiveness of Web caching together with performance evaluation, the next section gives a more thorough introduction to caching in the Web. Finally, we conclude in Section 5, summarizing our contribution and addressing our current and future work.

2 Caching in the Web

Basically, a cache is a software that is in charge of storing on disks, data elements that are accessed by a number of clients. The *cache management* policy drives data replacement within the disk so as to preferably keep in the cache, the data elements that are the most likely to be re-accessed. Within the Web, there are three types of caches in use, which are located at the three points of the client-server interaction (*i.e.*, at the level of the client, network, and server). These caches are further detailed in the following subsection. Given that most of the Web documents are accessed only once [2], techniques complementary to cache management have been proposed. In particular, due to the network topology, the idea of making network caches cooperate has emerged. Subsection 2.2 sketches base cooperative cache systems that have been proposed for network caches. Finally, Subsection 2.3 gives criteria that are used to measure the effectiveness of caching techniques, which will be used throughout this paper to compare various caching techniques.

2.1 Web caches

WEB caches subdivide into client, network, and server caches. A client cache executes on the client machine and is in general integrated in a browser. It aims at improving the response time for the client's WEB requests. Client caches enable to efficiently handle document re-accesses (*e.g.* backward and forward re-accesses). On the other hand, since client caches are non-shared, they have no impact on the response time for newly accessed documents. One way to improve the WEB latency when accessing a document for the first time, is to use predictive prefetching techniques. Predictive prefetching consists of anticipating the client requests so as to prefetch a document on the WEB server before the client actually requests it. Predictive prefetching subdivides into: *user-directed prefetching* where prefetching decisions are based on user-provided hints [24]; and *automatic prefetching* where prefetching decisions are based on information gathered at the level of the WEB server (*i.e.* documents that are frequently accessed together) [20, 4]. Prefetching has been shown to drastically reduce the time taken to get a requested WEB document. For instance, the technique proposed in [20] improves the WEB mean latency of up to 60%. However, it has also been shown that predictive prefetching causes a non-negligible increase in the network traffic due to erroneous prefetching decisions. Moreover, prefetching techniques are hardly compatible with network caches, which are considered as a key element to reducing the WEB latency.

In general, a client never interacts directly with the server holding the document it requests for. Instead, the request is issued to a network cache, which is shared by a number of clients. Thus, the client may benefit from earlier access to the same document by clients sharing the cache. If the cache owns a requested object, then it transfers the object to the client. Otherwise, the cache sends the request to the server, and gets the object. Then, in parallel, the cache stores the object on its disk and forwards it to the client.

The purpose of server caches differs from the one of client and network caches. It aims at reducing the processing capacity required for the server node, through (partial) replication of the server over a number of nodes that are physically distinct in terms of both network and machine. Thus, the primary goal of a server cache is not to improve the WEB latency but to reduce the server node's workload. To our knowledge, the distribution of user requests among server caches is not achieved according to load criteria but is instead based on a simple distribution scheme [14]. The DNS server owns a list of cache servers, and, upon a connection, the least recently used cache is selected to actually handle the request. This basic strategy enables to significantly decrease the server nodes' load while being easy to install and administrate; it is thus commonly used as it is for instance the case in the NCSA server [15].

The effectiveness of a WEB cache in particular depends on the cache replacement algorithm that is used. Ideally, the algorithm must keep in the cache the objects that will be re-accessed by the cache's clients. In the light of latest results about WEB usage, we have proposed novel cache management strategies which are more accurate than existing solutions regarding the undertaken replacement decisions (see Section 3). It is important to note that the various WEB caches are complementary rather than competing. In particular, it has been shown in [1] that the best way to achieve scalability over the INTERNET is to couple client, network, and server caches, data compression techniques, and possibly increase the bandwidth of the network links. The next subsection outlines yet another complementary method, which lies in the introduction of cooperative network cache systems.

2.2 Base cooperative cache systems

The hierarchical approach to network cache cooperation is among the pioneering one and has been proposed by P. Danzig [9] in the light of a study on the INTERNET traffic in the USA. Results of this study showed that the INTERNET traffic can a priori be reduced by 30% through the introduction of a cache on every network node. The hierarchical structure of the cache system then came from the hierarchical organization of national networks. In a hierarchical cache system, a cache gets a missing requested object by issuing a request to the cache at the upper level in the hierarchy. The process is iterated until either the object is found or the root cache is reached, which may ultimately contact the object's server. The object is then copied in all the caches that got contacted when returned to the client. A transversal system enriches the hierarchical one by integrating a set of sibling caches, which are close in terms of latency time, at each level of the hierarchy (see Figure 1). Then on a cache miss, a cache not only contacts its ancestor cache but also its siblings.

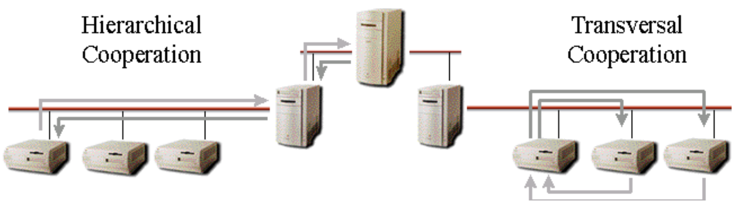

Figure 1: Cooperative caching

Hierarchical and transversal cache systems are being evaluated on various national networks such as RENATER in France and JANET in the UK. Early results show that hierarchical systems bring actual benefit but there is no significant result for transversal systems. The effectiveness of a transversal system mainly depends on the number of siblings and hence on the number of caches composing the system. But, network administrators voluntarily set this number to 3 or 4 in order to limit the bandwidth overhead caused by the cooperation protocol of the transversal system. This overhead is a direct consequence of how a cache miss is handled in the implementation artifact: a request message is sent to all the siblings, and each of them replies by a hit or miss message, depending on the presence or absence of the requested object. Thus, the number of messages that are exchanged to get a global information about an object within a transversal system is equal to $2 \times (N-1)$ with N being the number of siblings. This network overhead is far too costly and cannot be reduced since multicast is prohibited due to technical and security problems [26]. Another critical factor of a transversal system is the resulting workload for cache machines: a cache has to handle messages received from its siblings in addition to messages received from its clients. Actually, the workload of a cache in a transversal system is increased by a factor of N in terms of handled requests. Based on the aforementioned evaluation of transversal systems, we have designed a new protocol for transversal cooperation so as to gain the foreseen benefit of such cache systems while having negligible network and machine overhead, and providing a scalable solution (see Section 4).

Although transversal cooperative caching was initially proposed for network caches, it is important to note that it applies as well for server caches. In particular, we have used this strategy for management of the server of an electronic newspaper [8], which leads to increased server scalability. Transversal cooperation among client caches may also be exploited at the level of an organization.

2.3 Measuring caching effectiveness

So as to propose an evaluation of various caching techniques from the standpoint of their effectiveness, we use the following usual criteria: the *Hit Rate* (HR), the *Weighted Hit Rate* (WHR) and the mean response time to user requests (TIME). Precisely, the HR criterion gives the percentage of requested objects that are present in the cache, and the WHR criterion gives the percentage of bytes that are directly transferred from the cache to the client (*i.e* without contacting the server). Ideally, a caching technique must exhibit good performance with respect to the three above criteria.

3 Cache Management

The effectiveness of a single cache primarily depends upon the accuracy of the replacement decisions that are taken upon the cache's saturation. The following subsection provides an overview of the major replacement algorithms that have been proposed for WEB caches. We then introduce two novel algorithms, which respectively take advantage of the existence of hot WEB pages, and of the fact that 40% of the documents stored within a WEB cache are degradable images. Finally, an overview of the effectiveness of the replacement algorithms is given in the last subsection.

3.1 Replacement algorithms

Existing replacement algorithms for WEB caches subdivide into two categories depending on whether the replacement decision is based on a single criterion or on a set of criteria.

3.1.1 Single-criterion algorithms

Single-criterion replacement algorithms rely on the selection of a criterion according to statistical analysis of users behaviors. The main algorithms in use are the LRU, LFU, SIZE, and LAT algorithms.

The LRU (*Least Recently Used*) algorithm remains the most used one. For storing an object within a full cache, the LRU algorithm iteratively removes the least recently used object until there is enough place available. In some situations, this replacement strategy leads to remove a large number of small-size objects so as to store a large object. In order to avoid such a drift, LRU algorithms that further take into account the objects sizes have been proposed. These include the LRU-MIN algorithm that is presented in the next subsection.

Instead of considering the time a cached object was last accessed for replacement decision, the LFU (*Least Frequently Used*) algorithm considers the number of times a cached objects was accessed. Then, for storing an object, the algorithm iteratively removes the least frequently accessed object until there is enough place available in the cache. This algorithm has the same drawback as the LRU algorithm; it does not take into account the objects sizes. In addition, let us remark that since most of the cached objects are accessed only once, the behavior of the LFU algorithm becomes close to the one of a simplistic random algorithm.

Unlike the two previous algorithms, the SIZE algorithm takes replacement decisions based on the size of the WEB objects. Upon the cache saturation, the largest object is removed from the cache, and the removal process is iterated until there is enough place available to store the new object. Despite its simplicity, this algorithm is among the most efficient of the single-criterion replacement algorithms (see Figure 4 in Subsection 3.4).

The LAT (*Latency Access Time*) algorithm was the first replacement algorithm taking into account the time taken to get an object. Upon the cache saturation, the objects that are first removed from the cache are the ones whose access time is the lowest. The object's access time is based on the previous access times observed for the object. The major drawback of this algorithm is that it induces a significant management overhead. In addition to the storage of the objects' access times, the selection of the object to be removed causes a large processing overhead. This algorithm thus decreases the cache's efficiency and scalability. Moreover, since most of the objects are accessed only once, the access time that is estimated for an object is in general wrong. This has led the *Network research group* at Virginia Technology Blackburg to propose an estimation of objects access times with respect to servers and not the objects themselves [29]. Such a solution decreases the algorithm's memory consumption and further allows to estimate the access time for a newly requested object.

Performance evaluation of the above algorithms is presented in Figure 4 of Subsection 3.4. We see that the SIZE algorithm gives a good HR because it removes the largest objects and hence enables to store more objects within the cache. The LRU and LFU algorithms give a good WHR because, in general, large objects that get re-accessed, are in a small time interval. The LAT algorithm is effective with respect to TIME because any object that has a high access time is kept in the cache.

3.1.2 Multi-criteria algorithms

In general, single-criterion replacement algorithms are efficient with respect to one of the effectiveness caching criteria. Thus, one way to have a cache replacement algorithm that is efficient from the standpoint of HR, WHR, and TIME, is to make replacement decisions according to the conjunction of some criteria. Among existing multi-criteria algorithms, we present the two most efficient ones, *i.e.*, LRU-MIN and HYB.

The LRU-MIN algorithm minimizes the number of objects to be removed upon the cache saturation. Let V be the size of the object O to be placed in the cache, and L be the –possibly empty– list of the cached objects whose size is greater than V. If the list is non empty then an object is removed according to the LRU strategy. Otherwise, the value V is divided by two and the object list L is updated accordingly. Then objects of L are removed according to the LRU strategy until either O can be stored within the cache or L is empty. In the latter case, the algorithm iterates.

The HYB (*HYBrid*) algorithm takes into account a number of factors for object removal: object access time, object access frequency, and object size [6]. The objects that get removed from the cache are those which are optimal with respect to the three above factors. The algorithm is further generic in that it allows to weigh the importance of each of the factors in the removal process. The main drawback of this algorithm is that it induces a large processing overhead as for the LAT single-criterion algorithm. In particular, this has led to apply the optimization proposed for the LAT algorithm to the HYB algorithm [29]: access times are considered with respect to the servers and not to the objects.

As shown by the performance measures given in Figure 4 of Subsection 3.4, none of the two above algorithms exhibit good performance with respect to HR, WHR, and TIME. The two next subsections introduce two complementary replacement algorithms, which enable to significantly increase caching effectiveness.

3.2 LRU-hot

The survey about WEB usage presented in [3] shows that the requests issued to a WEB server relates to a small number of documents, highlighting the existence of *hot objects*. We thus have designed a replacement algorithm based on this criterion. The major difficulty then resides in identifying hot objects within a cache. We were not able to devise a reliable method to automatically differentiate hot and cold objects at the cache level. On the other hand, a WEB server can easily and efficiently detects its hot objects, by simply associating an access frequency to each of its objects. This has thus led us to use a solution based on server cooperation. An object is considered as being hot if its access frequency exceeds a given threshold (*e.g.* 0.10 was chosen for our performance evaluation[1]), and being cold otherwise. Then, when a server transfers an object, it includes a binary flag within the MIME header of the HTTP response message, notifying whether the object is hot or cold.

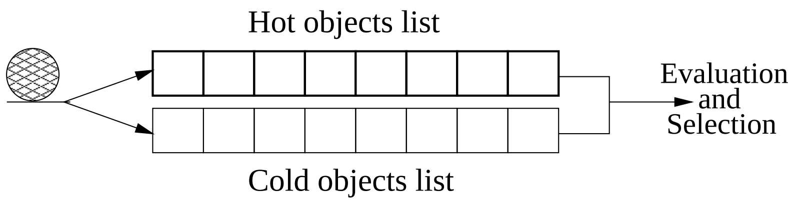

Figure 2: The LRU-HOT algorithm

Our replacement algorithm, called LRU-HOT, relies on the management of two object lists that respectively give hot and cold objects (see Figure 2). In each list, the objects are ordered according to their *re-accessibility*, which depends upon the time the object was last used, in a way similar to the LRU strategy. Thus, the re-accessibility of a stored object periodically

[1]This frequency value is set according to access frequency values observed for objects in WEB servers; it appears that most frequently accessed objects are those with an access frequency of at least 0.10 while the access frequency of remaining objects drops to 0.02.

decreases. The idea is then to make the re-accessibility of cold objects decrease faster than the one of hot objects. However, updating the re-accessibility values of objects must induce a bearable processing overhead. We thus chose to use two reference counters: a base counter and a counter for hot objects. Both counters are initialized to 0. Then, upon the reception of a client request, the base counter is increased by one. On the other hand, the hot counter is increased by one every α (*e.g.* 2 was chosen for our performance evaluation) client requests.

When a hot (resp. cold) object is stored within the cache, it is added to the hot (resp. cold) object list and is assigned an *access value* that is equal to the base counter value. The objects of the hot and cold lists being ordered according to the decreasing order of their access value, the newly received object becomes the head of the list to which it is added. In the same way, when a stored object gets re-accessed, the object is moved up to the head of its embedding list, and its access value is updated with the base counter value.

Upon the cache's saturation, the LRU-HOT algorithms computes the re-accessibility values for the two objects that are the tail of the hot and cold lists. Let *tail_hot* and *tail_cold* denote the access values associated to the tail object of the hot and cold lists, respectively. We get the following re-accessibility values for these two objects:

$$\begin{aligned} \textit{hot\_value} &= \textit{tail\_hot} - \textit{hot reference counter} \\ \textit{cold\_value} &= \textit{tail\_cold} - \textit{base reference counter} \end{aligned}$$

Then, the object that is removed from the cache is the one that has the smallest re-accessibility value. If the cache can still not host the newly received object, the algorithm iterates until there is enough place available.

As shown in Figure 4 of Subsection 3.4, the LRU-HOT algorithm performs better than existing replacement algorithms from the standpoint of HR, WHR, and TIME. It is further important to note the simplicity of the algorithm. Additional improvements could be envisioned for the algorithm. For instance, the selection of removed objects could be based on the objects' actual access frequencies instead of the basic binary hot status. However, this would lead to a major storage and processing overhead. Another point relates to the evaluation of the α parameter used for keeping hot objects longer than cold objects within the cache. So far, this value has been set arbitrarily two 2 based on the authors' intuition. It would be interesting to examine a fine tuning of this value.

3.3 LRU-QoS

The LRU-HOT algorithm has been proposed in the light of the observation of the documents that are requested from WEB servers. The statistical analysis of WEB caches further shows that 40% of the documents that are stored are images, which may be degraded at low cost using a JPEG-like algorithm. We thus have devised a replacement algorithm based on image degradation so as to save the disk space used by images.

The principle of the algorithm, called LRU-QOS, is similar to the one of LRU-HOT (see Figure 3). In the same way, the algorithm manages two lists: one relating to degradable objects (*i.e.* images) and the other to remaining objects. The base idea then consists of keeping as long as possible non-degradable objects at the expense of the degradable object quality. For image degradation, we choose to degrade an image by a factor of 50% although a higher factor could be taken while ensuring image readability. Here again, we use reference counters: a base counter

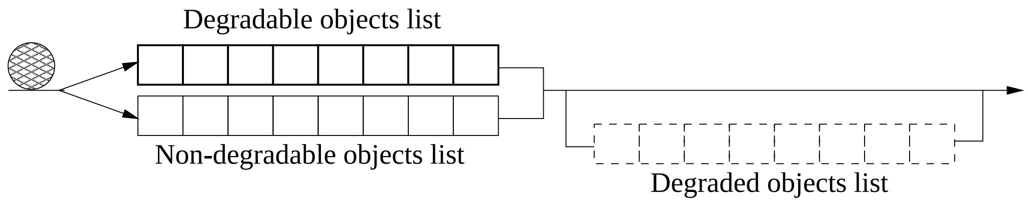

Figure 3: The LRU-QOS algorithm

and a counter for non-degradable objects, the latter being increased two times less frequently than the former. Upon the cache's saturation, either a degradable or a non-degradable object is chosen according to the respective re-accessibility values of the tail objects in the degradable and non-degradable lists. Re-accessibility values are computed as in the LRU-HOT algorithm. If a degradable object is selected for removal, it is actually degraded and put in a third list relating to degraded objects. If a non-degradable object is to be removed, then a second selection process occurs between the non-degradable object and the least recently used degraded object; the selected object is then removed from the cache. The second selection criterion that we use here is subjective and consists of setting that the cache stores no more degraded objects than degradable ones. Finally, the removal/degradation process iterates until there is enough place available in the cache.

Performance evaluation shows that the LRU-QOS algorithm enables to improve caching effectiveness although the benefit is not as drastic as with the LRU-HOT algorithm (see Figure 4 in Subsection 3.4). Different variants of the LRU-QOS algorithm can be considered. For instance, one could consider to degrade a bunch of images at once rather than a single image. Parameters of the algorithm could further be tuned in a finer way, including the degradation factor, the frequency with which the non-degradable reference counter is updated, and the size of the degraded object list.

3.4 Evaluation

For evaluation of the replacement algorithms discussed in this section, we have run simulations using the UC Berkeley traces [13]. These traces were collected over a 18-days period, starting November 1st, 1996. During this period, about 10,000 different persons have been accessing the WEB over the INTERNET, issuing about 2,500,000 requests for a total data volume of about 1 TeraByte. For confidentiality purpose, traces have been modified using cryptography techniques. Precisely, the HTTP requests have been replaced by their MD5 code [18] and the IP addresses of clients and servers have been replaced by anonymous addresses. We selected these traces because they enable to have a highly accurate evaluation of the replacement algorithms effectiveness from the standpoint of TIME. Actually, unlike most of the other available WEB traces, the UCB traces gives the time at which requests are issued and at which responses are received. Thus, it allows us to precisely infer the mean response time for client requests.

Let us consider the evaluation of the LRU-HOT algorithm, it requires to know the access frequency of each object. This frequency has been set to the number of times the object was accessed divided by the number of times the server owning the object was accessed. The objects' access frequencies are thus biased since they are computed with respect to the specific set of clients that were traced and not with respect to the overall requests handled by servers contacted by traced clients. It follows that evaluating the LRU-HOT algorithm using the overall traces would not be fair for the other algorithms. Hence, we have run simulations for a subset of the clients that were traced. Precisely, we chose 1,000 clients at random among all the clients, and the requests issued by these clients were used for the evaluation of all the replacement algorithms. Finally, the simulations were run on a 100 MHz PC PENTIUM running the NT 4.0 operating system.

Algorithms	HR	WHR	TIME
PERFECT	30.2	28.7	8.97
LRU	24.6	21.4	10.54
LFU	25.3	21.8	10.62
SIZE	26.0	18.4	9.43
LAT	24.7	20.4	9.51
LRU-MIN	25.9	20.2	10.20
HYB	25.4	22.0	9.31
LRU-HOT	27.1	23.5	9.41
LRU-QOS	25.6	23.1	10.01
LRU-HOT+QOS	27.3	24.1	9.39

Figure 4: Effectiveness of the replacement algorithms

Figure 4 gives performance measures for the replacement algorithms with respect to HR, WHR, and TIME. In addition to the algorithms that have previously presented, we also measured the effectiveness of the PERFECT and LRU-HOT+QOS algorithms. PERFECT corresponds to an ideal cache with an infinite storage capacity and serves to set the optimal caching effectiveness. LRU-HOT+QOS combines the two algorithms that we propose (see Figure 5). Precisely, this algorithm subdivide each of the degradable, non-degradable, and degraded object lists into two sub-lists corresponding respectively to hot and cold objects. The selection process for object removal/degradation then consists of applying the LRU-HOT strategy on sub-lists, and the LRU-QOS strategy among the lists.

As was already stated, our algorithms performs better than existing algorithms from the standpoint of HR, WHR, and TIME. However, the LRU-HOT algorithm has the drawback of requiring cooperation from the server; in the absence of server cooperation, its performance will be close to the one of the LRU algorithm since objects received by non-cooperating servers will

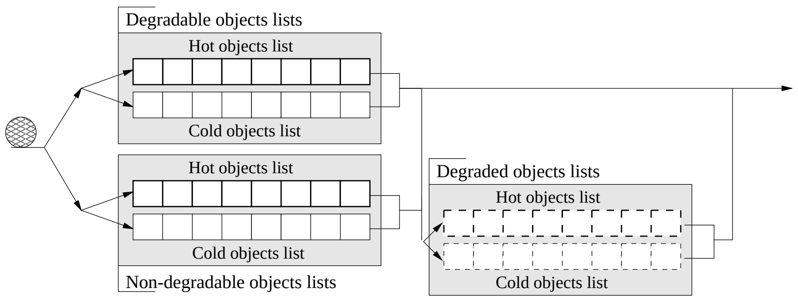

Figure 5: The LRU-HOT+QOS algorithm

be considered as cold. Practically, it is thus better to implement a LRU-HOT+QOS algorithm within caches.

Regarding the performance measures given for the PERFECT algorithm, one can notice that the effectiveness of a single cache is bounded. Precisely, the HR of a single cache with respect to the cache size is a logarithmic function [21]. Hence, although *a priori* beneficial, increasing the size of a cache has a bounded impact on the effectiveness of a single cache. Another more promising way to make caching more effective is to have a system of cooperating caches as discussed in Subsection 2.2. In this way, the number of clients of a cache increases, leading to a higher hit rate for the cache system. Existing work in this area together with our proposal to make more effective cooperative caching is discussed in the next section.

4 Cooperative Caching

We have seen that cooperative cache systems subdivide into complementary hierarchical and transversal systems. Although appealing, transversal cooperative caching suffers from the introduction of significant cache and network load since each cooperating cache has to handle the requests issued by its siblings in addition to the ones received from its clients. Solutions to this performance penalty have recently been proposed; they are outlined hereafter. Due to our focus on transversal caching, the term *cache system* will be used to mean *transversal cache system* in the remainder.

4.1 Improving the performance of transversal Web caching

To our knowledge, the CRISP proposal [12] was the first to address a solution to the ICP bandwidth overhead problem of (transversal) cache systems. This solution lies in the use of a centralized mapping server, which maintains the list of all the objects present in each of the cooperating caches. Thus, for a local cache miss, only 2 requests (central mapping server interrogation and response) are needed to localize an object. However, this solution has several drawbacks: it requires a specialized machine, the central mapping server, which can easily becomes a bottleneck; moreover, it needs strong synchronization among caches and the central mapping server for the maintenance of the object list. Many solutions are proposed to address this problem [11] but none of them guarantees to deliver a hit if the requested object is present in the system. Other work on the improvement of cooperative caching aims at reducing the memory consumption of caches by limiting object replication among the cooperating caches [16, 22]. However, they do not solve the problem of the resulting cache and network load.

A solution to minimizing the number of messages exchanged among sibling caches without using a centralized server has been proposed in the RELAIS project [23]. It lies in the local knowledge of the state of the transversal cache system on each cache. Then, on the receipt of an object request, the cache is able to locally identify whether the object is in the transversal cache system, based on its knowledge of the system's state. Should the object be within one of the cache's siblings, a request is issued to it in order to get a copy of the object. Local knowledge of the system's state has the advantage of introducing negligible workload for siblings when a cache handles an object request. However, such a local knowledge induces high memory consumption for storing the system state. It further causes a non negligible network load since messages need to be issued to siblings each time a cache is updated.

A protocol similar to the one of RELAIS but that induces less memory consumption has been proposed in [10]. The idea of this protocol is to encode the objects' URLs using Bloom filters [5] to reduce the size of the cache system state. Thus, the system state is encoded as a list of 4 bits signatures instead of a list of URLs. However, this space saving is at the expense of an approximation of the system state: several URLs may have the same encoding. Furthermore, this protocol still suffers from the same drawback as the RELAIS protocol; it causes a non negligible network load for update of the system state stored by the caches. In order to minimize the resulting overhead, only the code of the removed and added objects are transferred, and update messages are not sent on every cache update but at regular time intervals (*i.e.* about every five minutes). Notice that differing the update of the system state within caches decreases the efficiency of cooperative caching; it may lead to consider as absent an object while the object being actually stored within one of the caches.

In the following, we describe our solution to transversal cooperation based on the idea of having a local knowledge of the system state, while minimizing the resulting memory consumption and network load.

4.2 Design principles

One way to alleviate the penalty of using local knowledge of the whole system state from the standpoint of memory consumption, is to fairly distribute this knowledge among the caches composing the transversal system. Let us denote by $\mathcal{A}(C_i)$, the set of sibling caches for which cache C_i knows locally the content, $\mathcal{A}(C_i)$ including C_i. In other words, for each cache C_j of $\mathcal{A}(C_i)$, C_i maintains a list, which gives the objects identities that are present on C_j. Let us further denote by $\mathcal{S}$, the set of caches composing the transversal system. Then, to get knowledge of the whole system state and hence to locate an object that is not within a cache of $\mathcal{A}(C_i)$, C_i must contact a set of caches, noted $\mathcal{D}(C_i)$, as exemplified in Figure 6. The value of $\mathcal{D}(C_i)$ should then be such that:

$$\boxed{\mathcal{A}(C_i) \bigcup \cup_{(\forall C_j \in \mathcal{D}(C_i), j \neq i)} \mathcal{A}(C_j) = \mathcal{S}}$$

Thus, a cache C_i in the system can know the state of another cache C_x if C_i maintains locally a list for C_x or if one of the caches contacted by C_i, noted C_j (*i.e.* $C_j \in \mathcal{D}(C_i)$), maintains a list for C_x (*i.e.* $C_x \in \mathcal{A}(C_j)$). Note that when $\forall C_i \in \mathcal{S} : \mathcal{A}(C_i) = \{C_i\}$, we are in the presence of a traditional transversal system while when $\forall C_i \in \mathcal{S} : \mathcal{A}(C_i) = \mathcal{S}$, we are in the presence of a transversal system with local knowledge of the whole system state on each cache (*e.g.* systems proposed in [23, 10]). Our objective is to find a *scalable* solution that is a tradeoff between these two alternatives in terms of object retrieval latency, memory consumption, and message exchanges.

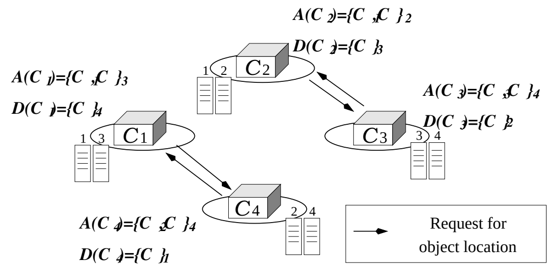

Figure 6: Exploiting distributed knowledge of the system state

Let us first examine the issue of object retrieval latency in the worst case, where we omit communication between the client and its cache since the associated latency is the same for any cache system. In a traditional transversal system, the object retrieval latency is equal to 4 message exchanges (considering that the time taken to retrieve an object within a cache is negligible compared to the cost of message exchanges): (*i*) For retrieving an object, a cache C_i sends, in parallel, the object request to its siblings, (*ii*) each sibling replies by a hit or a miss message depending on whether it holds the object or not, (*iii*) C_i sends a message to get the object, to one of the caches that replied by a hit, (*iv*) the object is finally sent to C_i. We observe the same latency using distributed knowledge of the system state. The only difference lies in the achievement of the first two steps: C_i sends messages to every cache C_j of $\mathcal{D}(C_i)$ instead of all the system caches, each C_j then returns the list of caches that own the object with respect to the object lists held for the caches of $\mathcal{A}(C_j)$.

Let us now consider the issue of memory consumption. For every cache C_i of $\mathcal{S}$, we want the cardinal of $\mathcal{A}(C_i)$, noted $|\mathcal{A}(C_i)|$, to be much smaller than $|\mathcal{S}|$. Furthermore, the scalability criterion requires load balancing, which leads to set an identical value for the cardinal of every $\mathcal{A}(C_i)$.

In the worst case, the number of messages that are sent by a cache upon the treatment of an object request, to locate the object, is equal to $|\mathcal{D}(C_i)|$ and hence we want the cardinal of

$\mathcal{D}(C_i)$ to be much smaller than the one of $\mathcal{S}$. Furthermore, as for the cardinals of the $\mathcal{A}(C_i)$s, the cardinals of all the $\mathcal{D}(C_i)$s should be identical for our scalability criterion.

Figure 6 gives an example of a transversal system with distributed knowledge of the system state, for a system composed of 4 caches. Setting such a system depends upon the evaluation of $\mathcal{D}(C_i)$ and $\mathcal{A}(C_i)$ for every cache C_i of $\mathcal{S}$. Furthermore, the value of these sets must satisfy the aforementioned condition of load balancing, and should be such that the number of messages sent for updating the system state on a cache update is minimal.

It can be shown [17] that we have an optimal system meeting our scalability requirement if $|\mathcal{A}(C_i)| = \sqrt{|\mathcal{S}|}$ and $|\mathcal{D}(C_i)| = \sqrt{|\mathcal{S}|} - 1$, the number of exchanged messages on a cache update being then of $\sqrt{|\mathcal{S}|} - 1$. With such values, the network load to get an object within a cache is equal to $3 \times (\sqrt{|\mathcal{S}|} - 1)$ messages, while it is equal to $2 \times (|\mathcal{S}| - 1)$ messages in a traditional transversal system.

4.3 Dealing with cache updates

Up to this point, we have not addressed the protocol used for updating the distributed system state in the presence of cache update. In the following, we introduce an update protocol whose bandwidth cost is negligible.

Let us first consider the addition of an object $\mathcal{O}$ within a cache C_i, C_i must *a priori* send an update message to every cache C_j whose set $\mathcal{A}(C_j)$ includes C_i. Beforehand, for getting $\mathcal{O}$, C_i already sent messages to caches belonging to $\mathcal{D}(C_i)$[2]. Caches belonging to $\mathcal{D}(C_i)$ can anticipate that $\mathcal{O}$ will be within C_i since C_i requested for it. The two cases where the anticipation is wrong is when either (*i*) $\mathcal{O}$ no longer exists or (*ii*) C_i did not get $\mathcal{O}$ due to some failure (*e.g.* temporary communication failure). The latter case is too rare to have a real incidence on the correctness of anticipation and is thus not considered. The former case can be seen as a special value for the object and is hence cached in place of an actual object value. Thus, whenever a cache C_i is requested an objet $\mathcal{O}$ by a sibling cache C_j, it can anticipate that C_j will hold C_i. Setting further the following *Rule 1*, this allows us to withdraw the network load that was expected for handling the addition of an object within a cache.

Rule 1: *For every C_i in $\mathcal{S}$, if C_i is in $\mathcal{D}(C_j)$ then C_j is in $\mathcal{A}(C_i)$.*

In other words, a cache C_i that is directly contacted by another cache C_j, *i.e.* $C_i \in \mathcal{D}(C_j)$, to get an object, maintains the state of this cache, *i.e.* $C_j \in \mathcal{A}(C_i)$.

Let us now examine removal of an object within a cache C_i of the system. Using simulation, achieved in the framework detailed in Paragraph 4.5.2, we have compared three approaches: (*i*) notification to caches of $\mathcal{D}(C_i)$ upon the object's removal, (*ii*) notification of the object removal to a cache of $\mathcal{D}(C_i)$ when such a cache requests for the object while the object is not present, (*iii*) absence of notification to caches of $\mathcal{D}(C_i)$ with periodical emission of C_i's state to caches of $\mathcal{D}(C_i)$. Figure 7 gives performance results obtained through simulation for the three solutions; it shows that the third solution gives better performance in terms of the overall bandwidth use, and was thus preferred for our protocol.

[2] As will be shown in Subsection 4.4, this is because a cache C_i that searches remotely for an object seeks the object, in parallel, within the caches of both $\mathcal{A}(C_i)$ and $\mathcal{D}(C_i)$.

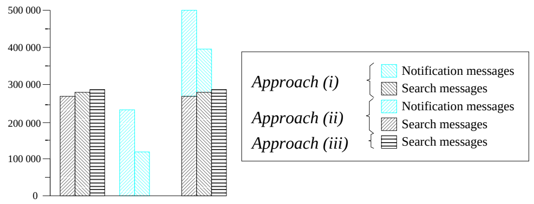

Figure 7: Number of exchanged messages with different notification policies for object removal

Figure 8 gives an example of a system composed of 4 caches, with distribution of the system's state according to Rule 1. As previously, setting such a system depends on the evaluation of $\mathcal{A}(C_i)$ and $\mathcal{D}(C_i)$ for every cache C_i of $\mathcal{S}$, where we recall that Rule 1 leads to have: $C_j \in \mathcal{D}(C_i) \Leftrightarrow C_i \in \mathcal{A}(C_j)$ for every C_i of $\mathcal{S}$.

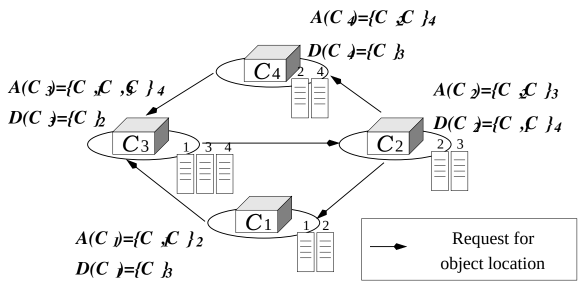

Figure 8: Distributed knowledge of system state

Let us detail the computation of an optimal cardinal for both $\mathcal{A}(C_i)$ and $\mathcal{D}(C_i)$, under the criterion of scalability. The principle of our protocol lies in the use of a knowledge relation, noted K, where $K(i,j)$ holds if cache C_i knows the state of C_j. For the cache system to be correct, $K(i,j)$ must hold for all the pairs of system caches. Given the proposed distribution of the system state, we are able to set that:

For two caches C_i and C_j of $\mathcal{S}$, $K(i,j)$ holds if either:
$C_j \in \mathcal{A}(C_i)$, or
$\exists C_k | C_k \in \mathcal{D}(C_i)$ and $C_j \in \mathcal{A}(C_k)$.

Since $C_i \in \mathcal{A}(C_i)$ always holds for any cache C_i of $\mathcal{S}$, we may simplify the above and get:

$$K(i,j) \text{ holds if } \exists C_k | C_k \in \mathcal{D}(C_i) \text{ and } C_j \in \mathcal{A}(C_k).$$

We can now compute the value of $|\mathcal{A}(C_i)|$ for each C_i where we recall that the load balancing condition leads to have $|\mathcal{A}(C_i)| = |\mathcal{A}(C_j)|$ for any two system caches C_i and C_j. The idea consists of distributing the knowledge relations, $K(i,j)$, $\forall C_i \in \mathcal{S}$, $\forall C_j \in \mathcal{S}$, among the caches of $\mathcal{S}$. Given that $K(i,i)$ always holds, the number of knowledge relations that must be distributed among the system caches is the combination of $|\mathcal{S}|$ elements, taken 2 at a time, noted $\binom{|\mathcal{S}|}{2}$, which is equal to $(|\mathcal{S}| \times (|\mathcal{S}|-1))/2$. We further want all the system caches to manage the same number of relations, leading to have $(|\mathcal{S}|-1)/2$ knowledge relations per cache. We are now able to infer the value of $|\mathcal{A}(C_i)|$ for every C_i of $\mathcal{S}$. By construction of $\mathcal{A}(C_i)$ and $\mathcal{D}(C_i)$, there are $\binom{|\mathcal{A}(C_i)|}{2}$ knowledge relations per cache, which should be equal to $(|\mathcal{S}|-1)/2$. Hence, we obtain:

$$\boxed{|\mathcal{A}(C_i)| = \sqrt{|\mathcal{S}| - 3/4} + 1/2, \forall C_i \in \mathcal{S}}$$

From the above, we are able to infer $|\mathcal{D}(C_i)|$. By construction, each cache C_j of $\mathcal{D}(C_i)$ knows the state of C_i, *i.e.* $C_i \in \mathcal{A}(C_j)$, and of some other caches. Thus, when C_i issues a request to a cache $C_j \in \mathcal{D}(C_i)$, only $(|\mathcal{A}(C_j)|-1)$ cache states known by this cache are significant for C_i to get knowledge of the whole system state. It follows that we should have:

$$|\mathcal{A}(C_i)| + (|\mathcal{D}(C_i)| \times (|\mathcal{A}(C_j)|-1)) = |\mathcal{S}|.$$

Hence, we get:

$$\boxed{|\mathcal{D}(C_i)| = |\mathcal{A}(C_i)| - 1, \forall C_i \in \mathcal{S}}$$

Given the values of $|\mathcal{D}(C_i)|$ and $|\mathcal{A}(C_i)|$ for all the system caches, it is straightforward to map the communication graph used by our protocol over the cache system, leading to set up the cooperation architecture. It amounts to compute an adjacency matrix such that the resulting boolean matrix has $|\mathcal{A}(C_i)|$ rows evaluating to true, $|\mathcal{D}(C_i)|$ lines evaluating to true, and $K(j,k)$ holds for any two caches C_j and C_k of $\mathcal{S}$. In other words, the matrix must contain $\binom{|\mathcal{S}|}{2} + |\mathcal{S}|$ elements evaluating to true with appropriate values for $|\mathcal{A}(C_i)|$ and $|\mathcal{D}(C_i)|$, $\forall C_i \in \mathcal{S}$. The matrix computation is based on a game algorithm using the branch and bound method: the algorithm searches an adjacency matrix with successive tests, and terminates when a valid matrix for our construction criteria is found.

4.4 The complete protocol

The cooperation protocol resulting from the use of a fair distributed knowledge of the system state is depicted in Figure 9. When a cache C_i receives an object request, it checks whether the object is locally present (**1**). If so, the object is sent to the client. Otherwise, the object is sought within the lists corresponding to the states of the caches belonging to $\mathcal{A}(C_i)$ (**2**). In parallel, a request for the object is sent to each cache of $\mathcal{D}(C_i)$ (**3**). A cache C_j receiving such a request updates the list corresponding to the state of C_i (**4**), *i.e.* C_j anticipates the presence of the object within C_i. C_j then searches for the object locally and within the state held for the caches belonging to $\mathcal{A}(C_j)$ (**5**). If the object is within C_j, C_j sends a hit to C_i while it sends a *miss* to C_i otherwise. Furthermore, for each cache C_k of $\mathcal{A}(C_l)$, $l \in \{i, j\}$, that has (supposedly) the object, C_l sends a request message to C_k (**6**). Then C_k either replies by a hit or a miss to C_i depending on whether the object is present or absent within the cache (**7**).

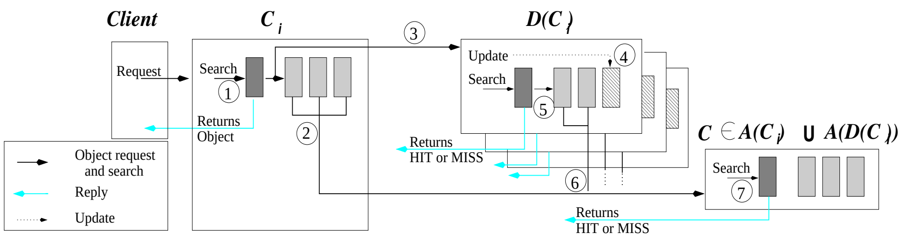

Figure 9: The cooperation protocol

Note that the object retrieval latency in the worst case is now equal to the time taken by 5 message exchanges instead of 4 as discussed in Subsection 4.2. This results from our handling of cache update. The knowledge of the state of the caches in $\mathcal{A}(C_i)$ for any cache C_i is now an approximation of the cache's state. Thus, when a cache C_j of $\mathcal{D}(C_i)$ gets an object request from C_i, it cannot send a hit or a miss on behalf of the caches in $\mathcal{A}(C_j)$, to C_i. Instead, C_j sends a request message to these caches, which reply directly to C_i. The penalty of one message is negligible from the standpoint of latency since the time taken for message exchanges between sibling caches is in general far less than the one between a cache and the server.

For illustration of the proposed cooperation protocol, let us consider a system composed of 7 caches. Each cache of the system must maintain the states of 3 caches (*i.e.* $|\mathcal{A}(C_i)| = 3$) and send requests to 2 caches (*i.e.* $|\mathcal{D}(C_i)| = 2$) to get knowledge of the whole system state. Figure 10 depicts two examples of cache misses in such a system. Let us concentrate on the example on the left hand side of the figure where there is a miss on cache C_1. C_1 seeks the object within the lists it has for the states of C_5 and C_7 (*i.e.* $\mathcal{A}(C_1) = \{C_1, C_5, C_7\}$), and, in parallel, sends request messages to C_4 and C_2 (*i.e.* $\mathcal{D}(C_1) = \{C_4, C_2\}$). If C_1 believes the object is present within either C_5 or C_7, or both, C_1 sends a request message to C_5 and/or C_7. Upon the receipt of the request from C_1, C_i, $i \in \{2, 4\}$, updates the state of C_1 and seeks locally the object. Depending on the presence or absence of the object, C_i either sends a hit or a miss to C_1. In parallel with the local search, C_i seeks the object within the state held for caches of $\mathcal{A}(C_i)$ (*i.e.* cache C_6 for C_2, and cache C_3 for C_4). If the object is supposed to be within such a cache, a request message is sent to it and the cache will send a hit or a miss to C_1 depending on its actual local state.

Unlike traditional cache systems, our approach to transversal cooperation introduces an interdependency among caches. In particular, removal of a cache due to failure must be addressed.

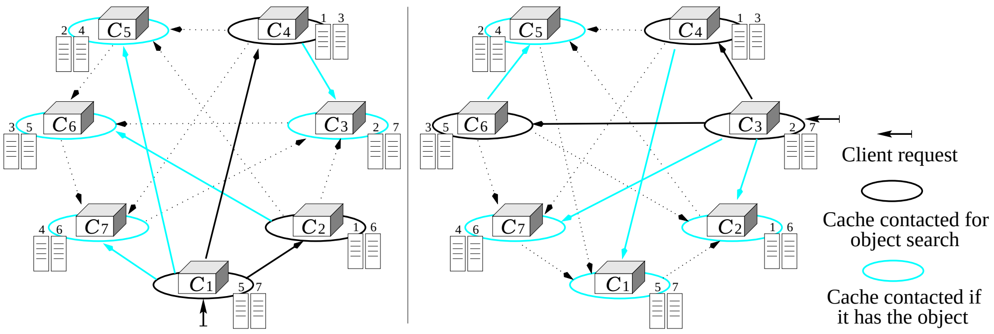

Figure 10: An example with 7 caches

In general, addition or removal of a cache requires to update the cooperation architecture, which implies the exchange of cache states (*i.e.* lists corresponding to cache states) between some caches. To cope with this issue, we have designed a dynamic reconfiguration method minimizing the resulting network load. The reconfiguration method is computed by one of the cooperative caches, selected among alive caches. The proposed reconfiguration method subdivides into two algorithms that respectively deal with the addition and removal of a cache. The original cooperation architecture is given by the adjacency matrix of size $|\mathcal{S}|$. In the presence of a cache addition, the reconfiguration computes a valid adjacency matrix (as defined in Subsection 4.3) of size $(|\mathcal{S}|+1)$ such that the number of cache states to be exchanged is minimal. The removal of a cache is handled according to the same principle. A valid adjacency matrix of size $(|\mathcal{S}|-1)$ must be computed while minimizing the resulting network load. Figure 11 gives the average number of states (in the link's label) that are exchanged between caches to cope with the addition and removal of a cache, for a system composed of up to 7 caches. For instance, adding a cache to a system made of 4 caches requires to exchange a single cache state. The proposed evaluation shows that the bandwidth used for reconfiguration is negligible (if reconfigurations are rare), compared to the one used by the transversal system.

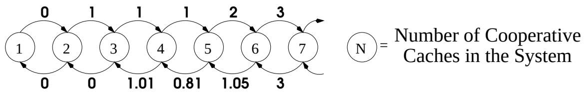

Figure 11: Reconfiguration cost

4.5 Evaluation

This section proposes an evaluation of a transversal system integrating our cooperation protocol. We base our evaluation on a comparison with the ICP transversal cooperation protocol [27]. For both protocols, the replacement algorithm run by caches is the LRU algorithm. Our comparison falls in two steps: (*i*) comparison from the standpoint of algorithmic complexity, and (*ii*) comparison through simulation so as to show the benefit of our solution in terms of caching effectiveness.

4.5.1 Algorithmic complexity

The ICP transversal cooperation protocol requires that a cache issues a request to all its siblings to get knowledge of the whole system state. Thus, the network load is $O(N)$ with N being the number of caches composing the system, *i.e.* $N = |\mathcal{S}|$. In the same way, the workload for each cache is $O(N)$ since each cache has to handle requests from its siblings, in addition to the requests received from its clients.

Let us consider a centralized system where one of the caches (say C_1) has a local knowledge of the whole system state for cooperation , *i.e.* $|\mathcal{A}(C_1)| = N = |\mathcal{S}|$. With such a system, the workload for C_1 is $O(N\ Log(N) + N)$ and the network load is O(1) (ignoring state update).

Finally, using our cooperation protocol, the network load is $O(\sqrt{N})$ since the number of requests that are issued to get a global knowledge is equal to $\sqrt{N - \frac{3}{4}} - \frac{1}{2}$ given the value set for $\mathcal{D}(C_i)$ (see Subsection 4.3). Furthermore, the complexity for the workload is $O(\sqrt{N}\ Log(\sqrt{N}))$ since, upon an object request, $\sqrt{N}$ caches get contacted and each of them performs a search within $\sqrt{N}$ lists for a cost equal to $O(\sqrt{N})$.

In addition to the algorithmic complexity, we can evaluate the needs in memory size of the systems, for an optimal search time. The search time is optimal if all the lists maintained by a cache are in main memory (hence, avoiding disk accesses). Let us assume that each cache stores 400,000 objects, and that the length of an object identifier (URL) is 50 bytes, the memory consumption per list is equal to 20 MBytes. Thus, 40 MBytes per cache is required with 7 cooperative caches, 60 MBytes with 13 caches, and 200 MBytes with 111 caches. As a comparison, a centralized server, like CRISP [12], requires for the mapping server 140 MBytes with 7 cooperative caches, 260 MBytes with 13 caches and 2,220 MBytes with 111 caches.

For illustration purpose, Figure 12 gives the bandwidth (resp. workload) cost for each cooperation protocol, for systems whose number of caches ranges from 1 to 21. It can be noticed that for a system composed of 21 caches, 40 messages are exchanged to get knowledge of the whole system state when using the ICP cooperation protocol while only 8 messages are required with our protocol. In general, the efficiency of our solution is proportional to the square root of the number of caches composing the transversal system. Thus, our protocol significantly decreases the bandwidth and workload induced by transversal cooperation. In order to further validate our solution, we have run simulations so as to study the behavior of our system in a real environment.

4.5.2 Simulation

The proposed simulations were achieved using traces collected on a cache located at the Technical State University of Virginia [25]. These traces are representative of the WEB accesses performed

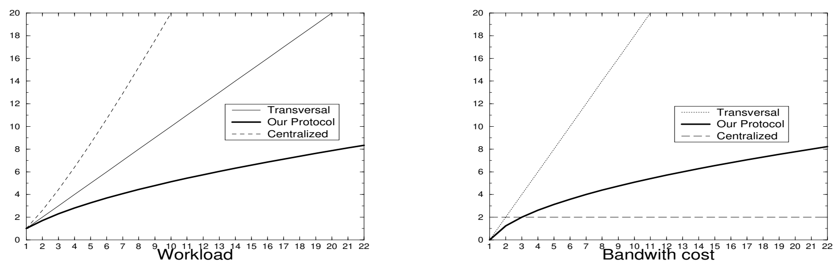

Figure 12: System comparison in terms of complexity

by the 3 following groups of University members: (*i*) students, (*ii*) professors, and (*iii*) students using a WEB browser during sessions of a class on multimedia. For our evaluation, we have used the traces corresponding to the accesses performed by the members of the first group. The number of computers used by this group were equal to 31, and traces were collected between April and October 1995, which gives traces over a 185-days period. During this period, 188,674 requests were sent by users and the total size of the objects that were transferred was equal to 2.26 GBytes.

In order to evaluate the efficiency of our proposal, we have subdivided the traces of the single cache that were available into 6 sets of traces. This subdivision was done according to the IP number of client machines, and each set of traces was assigned to a virtual cache. Each virtual cache was further assigned a disk capacity equal to 10% of the capacity required to avoid cache replacement, giving a cache size of 30 MBytes. Let us remark that the number of cooperating caches is set to 6 caches due to the traces that we have been using. We are about to run experiment with the UCB traces that we used for evaluation of the replacement algorithms (see Subsection 3.4) so as to run simulations for 32 cooperating caches. Furthermore, this experiment will use our prototype that is under testing.

Still concerning the evaluation of our protocol, the period for the emission of cache content for cache update (see Subsection 4.3), was set to a week (precisely, each Sunday at midnight). Finally, the simulations were run on a 100 MHz PC PENTIUM running the NT 4.0 operating system.

Two kinds of simulations were run: (*i*) one aims at comparing two transversal systems using respectively the protocol of ICP and ours, for a system composed of 6 caches, (*ii*) the other aims at comparing the two aforementioned cache systems from the standpoint of caching effectiveness, *i.e.* evaluating the hit rate of the two systems for the same bandwidth cost.

Evaluation for a fixed number of cooperating caches: We have extended to a cache system, the criteria used for the evaluation of the effectiveness of a single cache (see Subsection 2.3). We have considered the following additional criteria for evaluation:

- The *Sibling Hit Rate* (SHR) that gives the percentage of objects sought by a cache that are present in the cache system.
- The *Sibling Weighted Hit Rate* (SWHR) that gives the percentage of bytes obtained by a cache from the cache system over the total number of bytes obtained from the network (*i.e.* the remote server and the set of siblings).
- The *Group Hit Rate* (GHR) that gives the percentage of requested objects that are present in the cache system,
- The *Group Weighted Hit Rate* (GWHR) that gives the percentage of bytes that are directly transferred from the cache system to the client (*i.e.* without contacting the server).

As show in Figure 13, simulations gave identical results for both transversal systems in terms of all the effectiveness criteria. The main difference between the two cache systems lies in the bandwidth consumption and workload overhead. For identical results in terms of HR, the base transversal system consumes a bandwidth that is 138 % higher than the bandwidth consumed by a system integrating our cooperation protocol[3]. Thus, it increases by 82 % the number of requests treated by each of the cooperative caches.

		Workload	HR	WHR	SHR	SWHR	GHR	GWHR
Mean per cache	ICP	22700	30.93%	18.92%	14.56%	10.61%	40.90%	27.37%
	Our protocol	12474	30.93%	18.92%	14.56%	10.61%	40.90%	27.37%
	Gain	**-45.05%**	0%	0%	0%	0%	0%	0%

Bandwidth consumption (in number of messages)	
ICP	211330
Our protocol	88424
Gain	**-58.15%**

Figure 13: Caching effectiveness for a fixed number of cooperating caches

Evaluation for a fixed bandwidth cost: Let us now evaluate our system from the standpoint of caching effectiveness for a fixed bandwidth cost. This cost is given in terms of request messages exchanged among caches, which determines the maximal number of cooperating caches. If we set this cost to 4 messages, our protocol enables to have a single transversal system composed of 6 caches while the ICP protocol leads to have 2 transversal systems made of 3 caches. Simulation results showed that the efficiency of the cache system in terms of HR equals 30.93% in the absence of cooperation, 35.77% with transversal cooperation based on the ICP protocol,

[3]This follows from the fact that a cache miss leads to send 5 request messages in the base system while only 2 are sent with our system.

		Workload	HR	WHR	SHR	SWHR	GHR	GWHR
Mean per cache	ICP	12150	30.93%	18.92%	6.98%	4.97%	**35.77%**	**22.82%**
	Our protocol	12474	30.93%	18.92%	14.56%	10.61%	**40.90%**	**27.37%**
	Gain	**2.66%**	0%	0%	112.44%	149.73%	**14.22%**	**21.41%**
Bandwidth consumption (in number of messages)								
ICP		84532						
Our protocol		88424						
Gain		**4.60%**						

Figure 14: Caching effectiveness for a fixed bandwidth cost

and 40.90% with transversal cooperation based on our protocol. Figure 14 further shows the complete results of the mean caching effectiveness for the two systems.

In general, our simulation results about the efficiency of the cache systems show that the system based on our protocol is in the average 2.12 times more efficient than the other system in terms of HR, and is 2.49 times more efficient in terms of WHR. Depending on the number of caches composing the system and the associated system configuration, the improvement gained by our cache system ranges from 82% to 132%. Let us notice that to get the proposed results, our system actually increases the bandwidth cost of at most 4%, which is due to the absence of update of the distributed system state on object removal.

The efficiency of our system is a direct consequence of the number of caches composing the transversal system. Even better results could be exhibited with a system composed of more caches. Unfortunately, the traces we have been using to run our experiment were not suited to run the evaluation of such transversal systems. Further experiment will thus be run with our prototype, which is currently under testing, based on the UCB traces. In particular, we will then be able to evaluate and compare the fetching delay associated to our protocol and to the one of ICP.

We are currently working on the enhancement of the cooperation protocol so as to cope with heterogeneous architectures of cooperating caches, including caches having different capacities. Another concern is to dynamically compute the cooperation graph underlying the proposed protocol. In particular, this would allow to couple caches that frequently share common objects.

5 Conclusion

WEB caching is an appealing low cost technique for improving the WEB latency. Caching effectiveness then depends upon the replacement algorithm that is used for cache management, which has to keep in the cache the objects that will be re-accessed. However, the effectiveness of a single cache is bounded by the percentage of objects that are actually re-accessed. One way to increase this percentage is either to have a larger storage capacity for caches or to have a set of caches cooperate. The latter solution is the most cost-effective, which has given rise to cooperation protocols for the WEB. This paper has presented novel solutions for increasing caching effectiveness at the level of both a single cache and cooperating caches. We have introduced two novel replacement algorithms that performs better than existing ones from the standpoint of the resulting hit rate (HR), weighed hit rate (WHR), and response time for client requests (TIME). We further have proposed a new protocol for transversal cooperation among caches, which induces low network load, and low storage and processing overhead for caches, compared to alternative proposals. We have presented a performance evaluation of our solutions using simulation. We are currently testing a prototype, which integrates our replacement algorithms and cooperation protocol within the Squid software. We will then run experiment using this prototype so as to further assess the benefits of our solutions.

Our future work relates to devising a generic cache management strategy for WEB caches so as to adapt the strategy to the cache behavior. In particular, we believe that the effectiveness criterion that matters for a cache depends upon the type of the cache. If we consider a client cache, the most important criterion is TIME since the client is primarily interested in having a requested document in a short time interval. Let us now examine a network cache. From the standpoint of the provider, what matters is to save network bandwidth and to make documents available to clients in a timely manner. Thus, WHR and TIME should be optimized. Finally, regarding a server cache, it must minimize the accesses to the database server while responding in a timely manner. Hence, a server cache must optimize HR and TIME. Another factor of adaptation for cache management relates to caches that serve as both a server and a network cache. For such a cache, the management strategy must thus adapt to requests depending on whether they relate to network or server caching. We are currently designing a generic algorithm for WEB cache management, enabling adaptation of the strategy according to the aforementioned criteria.

References

[1] G. Abdulla, M. Abrams, and E. A. Fox. Scaling the World Wide Web. IEEE. 1996.

[2] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching Proxies: Limitations and Potentials. *Proceedings of the 4th International World-Wide Web Conference*, Available at http://ei.cs.vt.edu/~succeed/WWW4/WWW4.html. 1995.

[3] M. Arlitt and C. Williamson. Web Server Workload Characterization : The Search for Invariants. *ACM SIGMETRICS Conference*. May 1996.

[4] A. Bestavros. Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic, and Service Time for Distributed Information Systems. *Proceedings of the 1996 International Conference on Data Dissemination*. 1996.

[5] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), pp.422-426. July 1970.

[6] J.-C. Bolot, S.M. Lamblot, and A. Simoniai. Design of efficient schemes for the World Wide Web. *Proceedings of ITC 15.* June 1997.

[7] A. Chantkuthod, P. B. Danzig, C.Neerdaels, M. F. Schwartz and K. J. Worrell. *A Hierarchical Internet Object Cache.* Technical Report 95-611-University of Southern California, Boulder, California, USA. 1996.

[8] B. Charpiot, J-M. Menaud, V. Issarny, and M. Banatre. Quality of Service and Electronic Newspaper: The Etel Solution. *Chapter in Broadcast book.*

[9] P. Danzig, R. S. Hall and M. F. Schwartz. A Case for Caching File Object Inside Internetworks. *Proceedings of ACM Sigcomm'93*, pp.239-248. 1993.

[10] L. Fan, P. Cao , J. Almeida, and A. Broder. Summary Cache : A Scalable Wide-Area Web Cache Sharing Protocol. *Computer Communication Review, a publication of ACM SIGCOMM*, 28(4). October 1998.

[11] S. Gadde, J. Chase and M. Rabinovich. *Directory Structures for Scalable Internet Caches.* Technical Report CS-1997-18-Duke University, Durham, North Carolina, USA. November 1997.

[12] S.Gadde, M.Rabinovich, and J. Chase. Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches. *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Available at http://www.cs.duke.edu/ari/cisi/crisp/. May 1997.

[13] S. Gribble and E. Brewer. *UCB home IP HTTP traces.* Available at http://www.cs.-berkeley/g̃ribble/traces/index.html. June 1997.

[14] E. D. Katz, M. Butler et R. McGrath - A Stable HTTP Server : The NCSA prototype. *Computer Networks and ISDN Systems*, 27, pp.155-164. 1994.

[15] T. Kwan, R. McGrath and D. Reed. NCSAs World Wide Web Server : Design and Performance. *IEEE Computer*, pp.68-74. 1995.

[16] R. Malpani, J. Lorch, and D. Berger. Making World Wide Web Caching Servers Cooperate. *Proceedings of the 4th International World-Wide Web Conference*, pp.107-117. 1995.

[17] J-M. Menaud, V. Issarny, and M. Banatre. A New Protocol for Efficient Transversal Web Caching. *Proceedings of DISC'98 (Formerly WDAG) – 12th International Symposium on Distributed Computing.* 1998. *To appear.*

[18] A. Menezes, P. Van Oorschot, and S. Vanstone. Handbook of Applied Cryptography. *CRC Press.* 1997.

[19] J. C. Mogul. Hinted Caching in the Web. *Proceedings of the 7th ACM SIGOPS European Workshop*, Available at http://mosquitonet.stanford.edu/sigops96/papers/. 1996.

[20] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Newsletter.* 1996.

[21] G. Pierre and M. Makpangou. Saperlipopette!: A Distributed Web Caching Systems Evaluation Tool. *Proceedings of Middleware'98*. September 1998. To appear.

[22] D. Povey and J. Harrison. A Distributed Internet Cache. *Proceedings of the 20th Australian Computer Science Conference*, Available at http://www.psy.uq.edu.au/ ~dean/project/. 1997.

[23] The Relais Group. *Relais: Cooperative Caches for the World Wide Web*. Available at http ://www-sor.inria.fr/projects/relais. 1998.

[24] D. Steere and M. Satyanarayanan. Using Dynamic Sets to Overcome High I/O Latencies during Search. *Proceedings of the 5th Workshop on Hot Operating System Topics*. 1995.

[25] *Trace Files of WWW Traffic*. Available at http://www.cs.vt.edu/ ~chitra/www.html.

[26] D. Wessels. Configuring Hierarchical Squid Caches. Available at http://squid.nlanr.net/ Squid/Hierarchy-Tutorial. 1997.

[27] D. Wessels and K. Claffy. *Internet Cache Protocol (ICP), version 2*. National Laboratory for Applied Network Research/UCSD, Available at http://www.sztaki.hu/mirrors/squid/-Squid/rfc2186.txt. September 1997

[28] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World Wide Web documents. *Proceedings of ACM Sigcomm'96*, pp.293-305. 1996.

[29] R. Wooster and Abrams M. *Caching that Estimates Page Load Delay*. Network Research Group, Virginia Tech Blackburg, Available at http ://www.cs.vt.edu/ chitra/docs/www6r. 1996.