A New Protocol for Efficient Cooperative Transversal Web Caching

Jean-Marc Menaud, Valérie Issarny and Michel Banâtre

IRISA/INRIA Rennes Campus Universitaire de Beaulieu 35042 Rennes Cedex, FRANCE
{Jean-Marc.Menaud, Valerie.Issarny, Michel.Banatre}@irisa.fr

Abstract. The bandwidth demands on the World Wide Web continue to grow at an exponential rate. To address this problem, many research activities are focusing on the design of Web caches. Unfortunately, Web caches exhibit poor performance with a hit rate of about 30%. A solution to improve this rate, consists of groups of cooperating caches. In its most general form, a cooperative cache system includes protocols for hierarchical and transversal caching. Although such a system brings better performance, its drawback lies in the resulting network load due to the number of messages that need to be exchanged to locate an object. This paper introduces a new protocol for transversal cooperative caching, which significantly reduces the associated network load compared to that induced by existing systems. ...

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 [12]. 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 [1]. Furthermore, the use of large caches raises financial and technical problems. Other efforts have been focusing on the prefetching of data within caches but the resulting traffic overhead is too costly [7]. In that context, the most promising approach to cache effectiveness is to have a system of cooperating caches. Because that take advantage of 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 [11] from the Harvest project [2].

The hierarchical approach to cache cooperation is among the pioneering one and has been proposed by P. Danzig [3] 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. Then on a cache miss, a cache not only contacts its ancestor cache but also its siblings.

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 [11]. 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.

This paper introduces our protocol for transversal cooperative caching, which we intend to integrate within the Squid software through modification of Icp (*Internet Cache Protocol*) [11], hence using the support of Squid for hierarchical cooperative caching and base cache management. The next section describes the proposed protocol for transversal cooperative caching where we assume an identical bandwidth for all the network links between caches of the transversal system, and homogeneous machines for all the caches. Section 3 then gives an evaluation of our protocol through a comparison with the transversal cooperation protocol of Icp. It is shown that our objective of minimizing both the

cache workload and network load induced by transversal cooperation is met. Furthermore, our protocol allows to increase the effectiveness of a transversal cache system by a factor that is proportional to the square root of the number of caches in the system. Finally, Section 4 gives our conclusion.

2 Protocol for Efficient Transversal Cooperation

As previously stated, the workload caused by a transversal cache system results from the cooperation protocol that is used: a cache has to handle the requests issued by its siblings in addition to the ones received from its clients. In the Relais project [9], the way to minimize the number of messages exchanged among siblings is to have 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 as well as a non negligible network load since messages need to be issued to siblings each time a cache is updated asynchronously or not. 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. Due to our focus on transversal caching, the term *cache system* will be used to mean *transversal cache system* in the remainder.

2.1 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 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 1. The value of $\mathcal{D}(C_i)$ should then be such that:

$$\mathcal{A}(C_i) \bigcup \cup_{(\forall C_j \in \mathcal{D}(C_i), j \neq i)} \mathcal{A}(C_j) = \mathcal{S} \quad (1)$$

Thus, a cache C_i in the system can know the state of another cache C_x if C_i maintain locally C_x's list or if one of the caches contacted by C_i, noted C_j ($C_j \in \mathcal{D}(C_i)$), maintains C_x's list ($C_x \in \mathcal{A}(C_j)$). 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. Our objective is to find a solution that is a tradeoff between these two alternatives in terms of object retrieval latency, memory consumption, and message exchanges.

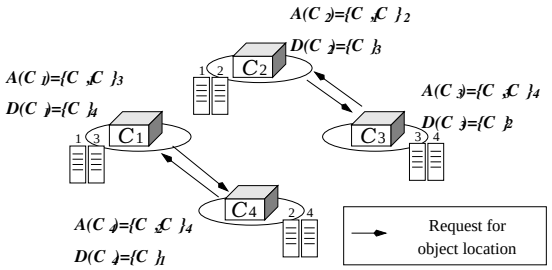

Fig. 1. Exploiting distributed knowledge of the system state

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 between caches (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's caches, each C_j then returns the list of caches that own the object with respect to the state 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 1 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 condi-

tion 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 (see appendix) 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.

2.2 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 $\mathcal{D}(C_i)$[1]. 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 on 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, C_i maintains the state of this cache (i.e. $C_j \in \mathcal{A}(C_i)$).

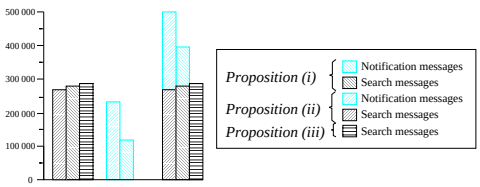

Fig. 2. Number of exchanged messages with different notification policies

[1] As will be shown in Subsection 2.2, 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 $\mathcal{D}(C_i)$.

Let us now examine removal of an object within a cache C_i of the system. Using simulation, achieved in the same framework as simulations detailed in Section 3.2, we have compared three approaches: (*i*) notification for all object 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 2 gives results obtained by our simulations. It shows that the third solution gives better performance in terms of the overall bandwidth use, which was thus preferred for our protocol.

Figure 3 gives an example of a system composed of 4 caches where the distribution of the system's state is done 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 the use of anticipation leads to have the following relation: $C_j \in \mathcal{D}(C_i) \Leftrightarrow C_i \in \mathcal{A}(C_j)$ for every C_i of $\mathcal{S}$.

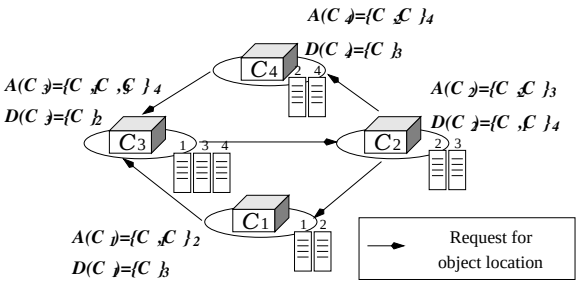

Fig. 3. Distributed knowledge of system state

Let us first address 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 that $K(i,j)$ holds if $\exists C_k | C_k \in \mathcal{D}(C_i)$ 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 $\mathcal{S}$. Given $|\mathcal{A}(C_i)|$ and 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:

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

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:

$$|\mathcal{D}(C_i)| = |\mathcal{A}(C_i)| - 1, \forall C_i \in \mathcal{S} \quad (3)$$

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 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.

2.3 The complete protocol

The cooperation protocol resulting from the use of a fair distributed knowledge of the system state is depicted in Figure 4. 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**).

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 2.1. 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

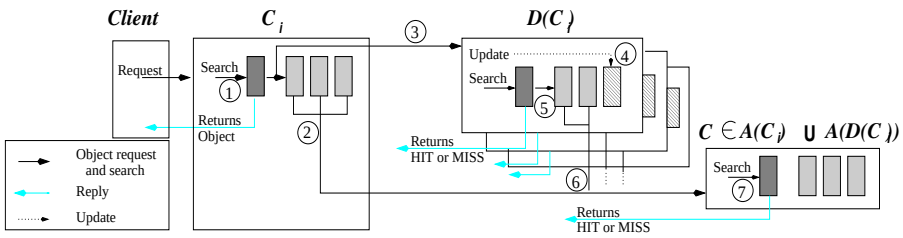

Fig. 4. The cooperation protocol

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 5 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 that 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.

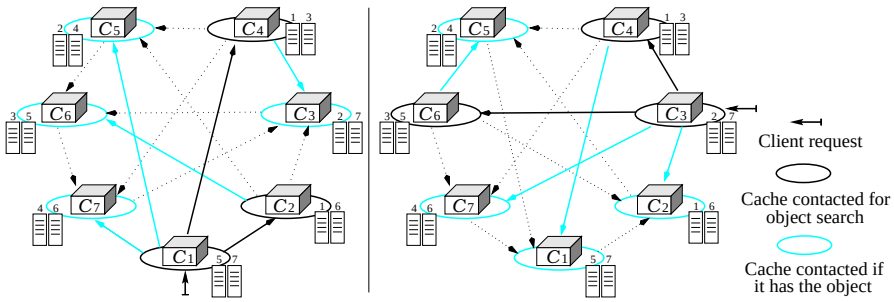

Fig. 5. An example with 7 caches

2.4 Discussion

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. 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. In the presence of a cache addition, the original cooperation architecture is given by the adjacency matrix of size $|\mathcal{S}|$. The reconfiguration computes a valid adjacency matrix (as defined in Subsection 2.2) 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 6 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.

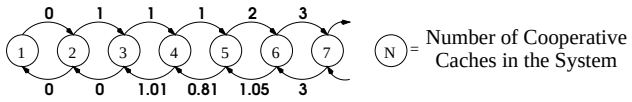

Fig. 6. Reconfiguration cost

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.

3 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. Our comparison falls in two steps: (*i*) comparison of the two approaches to transversal cooperation from the standpoint of algorithmic complexity, and (*ii*) comparison of the two approaches through simulation so as to show the benefit of our solution in terms of caching effectiveness.

3.1 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 $O(N\ Log(N))$ 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 § 2.2). Furthermore, the complexity for the workload is $O(\sqrt{N}\ Log(\sqrt{N}))$ since, on an object request, $\sqrt{N}$ caches get contacted and each of them performs a search within $\sqrt{N}$ lists for a cost equal to $O(\sqrt{N})$.

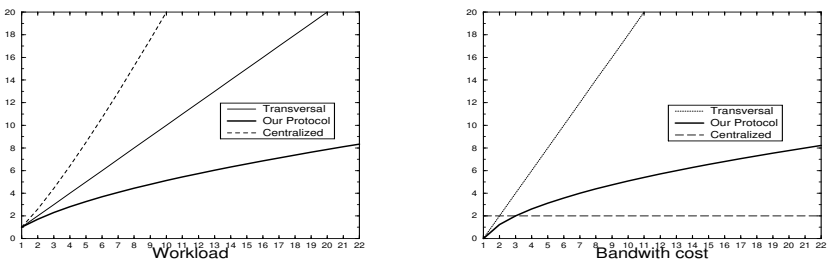

Fig. 7. System comparison in terms of complexity

In addition to the algorithmic complexity, we can evaluate the needs in memory size to have an optimal search time. The search time is optimal if all the lists maintained by a cache are in main memory (to avoid disk accesses). Thus, if we assume that each cache stores 400 000 objects, and that the length of an oject identifier (URL) is of about 50 bytes, the memory consumption per list is equal to 20 MB. Thus, 40MB per cache is required with 7 cooperative caches, 60MB with 13 caches, and 200MB with 111 caches. For comparison, a centralized server, like CRISP [5], requires for the mapping server 140 MB with 7 cooperative caches, 260MB with 13 caches and 2 220MB with 111 caches.

For illustration purpose, Figure 7 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.

3.2 Simulation

The proposed simulations were achieved using traces collected on a cache located at the Technical State University of Virginia [10]. These traces are representative of the Web accesses performed by 3 groups of University members, i.e. (*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. Furthermore, concerning the evaluation of our protocol, the period for the emission of cache content for cache update (see Section 2.2), 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: The efficiency of a single cache is measured in terms of its *Hit Rate* and its *Weighted Hit Rate*. The former gives the percentage of requested documents that are present in the cache, and the latter gives the percentage of bytes transferred from the cache to the client.

We have extended the above criteria, relating to a single cache, to a cache system. We have considered the two following criteria for evaluation: (*i*) the *Group Hit Rate* (Ghr) that gives the percentage of requested documents that are present in the cache system, (*ii*) the *Group Weighted Hit Rate* (Gwhr) that gives the percentage of bytes transferred from the cache system to the client.

Simulations gave identical results for both transversal systems in terms of GHR and GWHR. In the same way, we have evaluated the *Sibling Hit Rate* (resp. *Sibling Weighted Hit Rate*) that gives the proportion of objects (resp. bytes) obtained from a cache over the total number of objects (resp. bytes) obtained from the network (i.e. the remote server and the set of siblings). Similar results were observed for both transversal systems, as shown in table 1.

Table 1. Caching effectiveness for a fixed number of cooperating caches

Results for a fixed number of cooperating caches								
		Workload	HR	WHR	SHR	SWHR	GHR	GWHR
Mean per cache	ICP	142102	30.93%	18.92%	14.56%	10.61%	40.90%	27.37%
	Proposition	78087	30.93%	18.92%	14.56%	10.61%	40.90%	27.37%
	Gain	**-45.05%**	0%	0%	0%	0%	0%	0%
Bandwith consumption (in number of messages)								
ICP		1322925						
Proposition		553534						
Gain		**-58.15%**						

The main difference between the two cache systems lies in the bandwidth consumption and workload overhead. For identical results in terms of hit rates, the base transversal system consumes a bandwidth that is 138 % higher than the bandwidth consumed by a system integrating our cooperation protocol[2]. Thus it increases by 82 % the number of request treated by each of the cooperative 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 *Hit Rate* equals 30.93% in the absence of cooperation, 35.77% with transversal cooperation based on the ICP protocol, and 40.90% with transversal cooperation based on our protocol. Table 2 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 showed that the system based on our protocol is in the average 2.12 times more efficient than the other system in terms of *Hit Rate*, and is 2.49 times more efficient in terms of *Weighed Hit Rate*. 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%.

[2] 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.

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.

Table 2. Caching effectiveness for a fixed bandwith cost

Results for a fixed bandwith cost								
		Workload	HR	WHR	SHR	SWHR	GHR	GWHR
Mean per cache	ICP	76059	30.93%	18.92%	6.98%	4.97%	**35.77%**	**22.82%**
	Proposition	78087	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%**
Bandwith consumption (in number of messages)								
ICP		529170						
Proposition		553534						
Gain		**4.60%**						

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 13 or 21 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 using our prototype, which is currently under testing. In particular, we will then be able to evaluate and compare the fetching delay associated to our protocol and to the one of ICP.

4 Conclusion

This paper has introduced a new protocol for cooperation among caches composing a *transversal system* so as to reduce both the bandwidth load and the cache workload, which are general induced by such systems. The novelty of our protocol comes from a fair distribution of the whole system state among the cooperating caches, which significantly reduces the load caused by transverse cooperation. As a consequence, this enables to set up transversal systems composed of a larger number of caches compared to existing systems, hence leading to improved cache effectiveness.

Among related work, let us cite Gadde's proposal with CRISP [5], who addresses a solution to the ICP bandwidth overhead problem, by using a centralized mapping server, maintaining a list of all objects present in each system's cache. 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 (i.e. the central mapping server) which can easily becomes a bottleneck ; moreover, it needs a synchronous interaction between caches and the central mapping server for the maintenance of the object lists. Many solutions are proposed to address this problems [4] but

none of them guarantee to deliver a hit if the requested object is present in the system. There exist other various studies on the improvement of cooperative cache systems. However, we are not aware of a proposal similar to ours. Existing work should rather be considered as complementary to our effort. For instance, let us mention the work of R. Malpani [6] and D. Povey [8] whose contribution lies in reducing the memory consumption of caches by limiting object replication within the system.

We are currently working on the integration of our protocol within the Squid software. In addition, we are examining an enhancement of our protocol in order to cope with heterogeneous architectures of cache systems, including systems composed of 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.

References

1. 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 from http://ei.cs.vt.edu/~succeed/WWW4/WWW4.html – 1995.
2. 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 – 1996.
3. 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.
4. S. Gadde, J. Chase and M. Rabinovich - Directory Structures for Scalable Internet Caches - Technical Report CS-1997-18-Duke University, Durham, North Carolina – Novenber 1997.
5. S.Gadde, M.Rabinovich, and J. Chase - Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches. -The Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI), Available from http://www.cs.duke.edu/ari/cisi/crisp/ – May 1997.
6. 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.
7. G. C. Mogul - Hinted Caching in the Web - Proceedings of the 7th ACM SIGOPS European Workshop, Available from http://mosquitonet.stanford.edu/sigops96/papers/ – 1996.
8. D. Povey and J. Harrison - A Distributed Internet Cache - Proceedings of the 20th Australian Computer Science Conference, Available from http://www.psy.uq.edu.au/ ~dean/project/ – 1997.
9. The Relais Group. Relais : cooperative caches for the world wide web. Available from http ://www-sor.inria.fr/projects/relais - 1998
10. Trace Files of WWW Traffic Available from http://www.cs.vt.edu/ ~chitra/www.html.
11. D. Wessels. - Configuring Hierarchical Squid Caches - Available from http://squid.nlanr.net/ Squid/Hierarchy-Tutorial – 1997.

12. 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.

Appendix

In our system the number of messages sent for an object search depends on the list number maintained by each cache, noted $|\mathcal{A}(C)|$, and is egual to $\frac{|\mathcal{S}|-|\mathcal{A}(C)|}{|\mathcal{A}(C)|}$. The number of research mesages sent by a cache is inversely proportional to the number of lists maintained by a cache. In opposition, the number of updating messages is proportional to $|\mathcal{A}(C)|$ and equal to $|\mathcal{A}(C)|$. So, the total number $\mathcal{N}$ of exchanged messages for the object search and fetching is a function which depends on the $|\mathcal{A}(C)|$ and which is egual to $\mathcal{F}(\mathcal{N}) = \frac{|\mathcal{S}|-|\mathcal{A}(C)|}{|\mathcal{A}(C)|} + |\mathcal{A}(C)|$.

Finaly, by derivation, we find that the optimal number of list maintained by each cache for minimizing the number of messages exchanged[3], is egual to $|\mathcal{A}(C_i)| = \sqrt{|\mathcal{S}|}$.

[3] The minimal value for $\mathcal{F}(\mathcal{N})$ is obtain when $\mathcal{F}'(\mathcal{N}) = 0$