Cache as ca$h can

W.J. Grootjans, M. Hochstenbach, J. Hurink,
W. Kern, M. Luczak, Q. Puite, J.A.C. Resing, F.C.R. Spieksma

Abstract

In this contribution several caching strategies for the World Wide Web are studied. Special attention is paid to the so-called proxy placement, i.e. placing of caches on carefully selected nodes in the network near to the end users. Using both a deterministic and a stochastic approach, algorithms are developed for calculating the allocation and sizes of caches with the aim to enhance the performance of the network. Under the restriction of fixed budget it is also indicated how both approaches can be combined.

Keywords

Cache assignment, World Wide Web, Allocation, Proxy placement.

1 Introduction

The World Wide Web (WWW) has experienced continuous, exponential growth since its inception in the beginning of the 90's. This has led to a considerable increase in the amount of traffic over the internet. As a consequence, Web users nowadays can experience large waiting times (latencies) due to network congestion and/or server overloading. Moreover, if current predictions concerning usage of the Web come true, this performance issue will become even more important in the near future.

A way to improve the performance of the Web is caching (as witnessed by for instance [2], [11]). Caching copies of popular objects closer to the user is an important way of improving the network's performance. Indeed, the two main potential benefits of caching are: reduction of latencies experienced by the users, and saving of bandwidth, due to a decrease of network traffic. In order to realize these potential benefits, at least two (related) problems have to be dealt with:

- how to operate a cache. There is a sizable literature devoted to caching strategies to improve Web performance; see, for instance, [1] and [9] where algorithms generalizing the well-known LRU algorithm are proposed.
- where to install cache. Different caching options are possible: on the one hand, objects can be stored at the user's browser, which gives the possibility to make use of the user's individual characteristics (client caching, see for instance [1] and [4]); on the other hand objects can be stored in the cache of the Web server (see [8]). In between these options, there is the option of using *proxies*, that is, to install specialized servers at various points in the network (first proposed by [6]; see also [3]). Typically, such an approach is attractive when an organization (like a company or a university) is responsible for (a part of) a network ([7]).

This paper deals with the latter subject called proxy placement in [10]: given a network with capacitated edges, external demand (request rates for objects and their sizes), costs for installing

a proxy and a budget, we develop a heuristic method to decide where to install proxy caches in the network and what the sizes of these caches should be. The heuristic attempts to minimize a function of the waiting times in the network, for instance, the average waiting time. We assume that only passive caching is used, i.e., features like pre-fetching and pre-loading are excluded.

The rest of this paper is organized as follows. Section 2 describes the problem and introduces some terminology. In Section 3 we propose an algorithm that suggests a proxy placement to minimize waiting times. Finally, Section 4 analyzes some stochastic aspects of the problem.

2 Problem description, notation and terminology

The input for our problem is as follows:

1. An *infrastructure* $\mathcal{T} = (\mathcal{V} \cup \{\infty\}, \mathcal{E})$ is a rooted tree (∞ standing for the root), such that the root has exactly one child. The root represents the outside world and the *inner vertices* (elements of $\mathcal{V}$) represent servers. The *edges* (elements of $\mathcal{E}$) are directed in direction of the root and represent connections between the servers. The relation "i' is a child of i" generates a partial order $\preccurlyeq$ ("descendant of") with top ∞ (i.e. $\forall i \in \mathcal{V} : i \preccurlyeq \infty$).

   Observe that the inner nodes are in 1-1-correspondence with the edges, each inner node being a child of another node via the corresponding edge. Let H denote the height of the tree and M the maximal number of children per node. Typical values for H and M are 5 and 100 respectively.

2. The files $1, \ldots, N$ that are requested at the servers have sizes $s_1, \ldots, s_N$, are located ouside the infrastructure, and can be achieved only via the root.

3. Let $\lambda_{i,j}$ denote the *frequency* of requests for file j at (inner) node i (in terms of number of requests per time unit). The following quantities are closely related to these frequencies: let

$$\lambda_i := \sum_{j=1}^{N} \lambda_{i,j}$$

   denote the *total frequency* of requests at node i. Then

$$p_{i,j} := \frac{\lambda_{i,j}}{\lambda_i}$$

   denotes the *relative frequency* of the requests for file j at server i. That is, for every i the function $j \mapsto p_{i,j}$ is a discrete distribution, $\sum_{j=1}^{N} p_{i,j} = 1$.

   The *demand* (data per time unit) generated by requests for file j at server i equals

$$\kappa_{i,j} := \lambda_{i,j} s_j.$$

   The *total demand* caused at server i is given by

$$\kappa_i := \sum_{j=1}^{N} \kappa_{i,j} = \lambda_i \sum_{j=1}^{N} p_{i,j} s_j.$$

4. Each edge $e \in \mathcal{E}$ has a *capacity* $c_e \geq 0$: the maximal flow (in terms of data per time unit) through the edge.

5. The costs to place a proxy in a node i are a linear function of the size of the cache y. If a denotes the fixed and b the variable costs, we can write the costs as

$$k(y) = \begin{cases} a + by & (y > 0) \\ 0 & (y = 0) \end{cases}$$

6. We have a total budget $B > 0$ to purchase proxies.

It is important to realize that given this input, any decision concerning the location, size and a local caching strategy for each proxy determines, in a unique fashion, flows in the network $\mathcal{T}$ (assuming that requests are served from proxies "up the tree" or from ∞ in case there are no proxies up the tree that contain the specific request). We will use variables x_e, $e \in \mathcal{E}$ to denote these flows. In particular, if no proxies at all are installed in $\mathcal{T}$, one can compute that the edge flows, denoted by x_e^0 in this case, are equal to $\sum_{i' \preccurlyeq i} \kappa_{i'}$ where i is the node directly under edge e. Let us now explicitly describe the assumptions that we use in our model:

- as mentioned above, requests are served by the closest proxy up the tree that contains the requested object. This assumption is reasonable in practice.
- in Section 3 we assume a *static* strategy as a local caching strategy, that is a set of files is chosen to remain in the cache permanently. Obviously, this is a crude simplification of reality, where LRU type of caching strategies are common. However, in this section we are primarily interested in proxy placement and their corresponding sizes; the specific local caching strategy is of minor importance in our setting which justifies this assumption. Afterwards, in Section 4 the results are analysed on the base of a LRU caching strategy.
- to be able to compute a waiting time for each edge $e \in \mathcal{E}$ (denoted by w_e), we rely on the following relation between waiting time w_e, (given) capacity c_e and flow x_e:

$$w_e = C \times \left(1 - \frac{x_e}{c_e}\right)^{-1} \qquad \text{for some constant } C.$$

- we assume that the objective function that we want to minimize, say h, can be expressed in terms of the waiting times w_e. For example, suppose one would like to minimize the largest waiting time experienced by some user in the network. This can be formulated as follows: let $\mathcal{P}$ denote the set of all paths from ∞ to any inner node, then the objective function is given by

$$h = \max_{p \in \mathcal{P}} \sum_{e \in p} w_e.$$

  Alternatively, the average waiting time in the tree over all possible paths can be formulated as

$$\frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \sum_{e \in p} w_e = \frac{1}{|\mathcal{P}|} \sum_{e \in \mathcal{E}} w_e \left|\{p \in \mathcal{P} \mid e \in p\}\right| = \frac{1}{|\mathcal{V}|} \sum_{e \in \mathcal{E}} n_e w_e,$$

  where n_e is the number of nodes in the subtree under e.

Concluding, a solution to our problem specifies for each node in the tree whether or not a proxy is installed, and, if so, it specifies its corresponding size. In addition, specific files are suggested to be stored in each proxy. All this is done while attempting to minimize (a function of) the waiting times.

3 The cache assignment

In this section we will consider the problem of determining the nodes in the tree where cache will be assigned and the files which will be stored in these caches. The goal is to achieve a comfortable situation for the users of the network. Due to the assumptions made in the previous section, the quality of the solutions depends on the waiting times and, therefore, on the loads in the edges.

Our heuristic algorithm to solve the problem consists of 2 major steps that are performed iteratively. In Step 1, we specify *upper bounds* u_e on the loads for each $e \in \mathcal{E}$. Next, in Step 2 we

(heuristically) decide whether a proxy placement exists such that $x_e \leq u_e$ for each $e \in \mathcal{E}$, and such that the total expenses remain within budget B. If the answer is yes, we update the upper bounds u_e in such a way that they correspond to a more comfortable situation and iterate, otherwise we either stop, or relax the current upper bounds. Subsection 3.1 deals with Step 1 and the updating of the upper bounds and Subsection 3.2 describes Step 2.

3.1 Step 1: computing and updating upper bounds u_e

The basic structure of the algorithm is visualized in Figure 1. It consists of an initialization of the bounds and a loop process in which the upper bounds are updated according to the the procedure which will be explained in the next section. In the following some remarks how these steps may be realized are given:

- To start, the algorithm needs an initial set of total edge flows for all edges. We propose two ways to find this set of flows: when the instance under consideration corresponds to an existing network, one can use the current situation as a starting point. More specifically, the current proxy placement and the current flows can be used as input for the algorithm. Another possible way to get an initial flow is as follows: specify a maximal waiting time for each edge $e \in \mathcal{E}$: w_e^*. Now $w_e \leq w_e^*$ is equivalent to

  $$x_e \leq \left(1 - \frac{1}{w_e^*}\right) c_e =: u_e.$$

  The corresponding flows can be used as input for the algorithm.

- The upper bounds are updated to the current total edge flows and a factor times the gradient of the objective function is subtracted. This means that the upper bound for each total edge flow is reduced proportional to the rate of descent of the objective function with respect to that total edge flow, which is symbolically denoted by $\nabla h(\mathbf{x})$. This gradient is evaluated for the current total edge flows. Notice that we rely here on the assumption that we are able to compute this gradient (cf. the choices of h mentioned in Section 2).

- The question whether or not an allocation with $\mathbf{x} \leq \mathbf{u}$ (for all components) exists, can be answered by the "inner loop" which yields either the answer *no* or the answer *yes* and a set of total edge flows $\mathbf{x}$.

  If the answer from Step 2 is negative then we go back a few steps in the algorithm and decrease the upper bounds less than we did initially, or we have found a solution that we consider satisfactory and stop.

  If the answer from Step 2 is positive then the new set of total edge flows becomes the set of upper bounds and the algorithm reiterates.

- The value of α_0 and the way α is decreased will have to be looked at using an implementation. At this time we cannot say anything sane on these matters.

3.2 Step 2: proxy placement for given upper bounds u_e

Given upper bounds u_e, we determine whether we can find flows $x_e \leq u_e$ by allocating a total amount of cache of cost $\leq B$. We divide this problem into three subproblems:

P1) Determine in which nodes a proxy is installed;

P2) Determine how much cache is installed in these nodes;

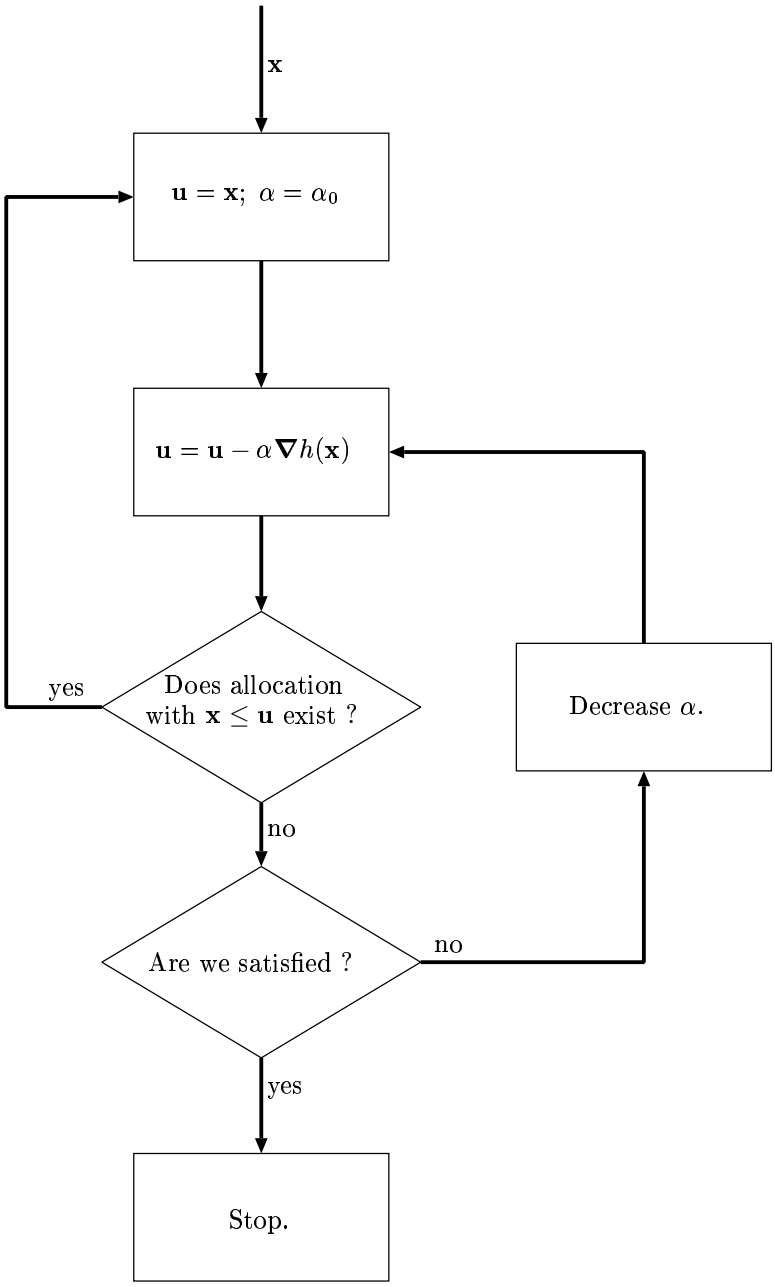

Figure 1: Flowchart of the updating of the bounds.

P3) Determine which files are stored in these proxies.

We solve these three subproblems by first considering P1, and next P2 and P3 simultaneously. Notice that Step 2 either outputs "yes" with an accompanying flow x or "no" meaning: no flow x with $x_e \leq u_e$ with expenses $\leq B$ is found.

3.2.1 Determine nodes in which a proxy is installed

In this subsection we describe an algorithm that determines in which nodes of $\mathcal{T}$ a proxy is installed. In fact, this algorithm determines the *minimal number* of proxies and possible locations for them that are necessary to achieve flows $x_e \leq u_e$ by a straightforward "bottom up" search in linear time. However, although our result is minimal in the mentioned sense, it may not be optimal concerning the complete problem.

To describe the algorithm, first we need some additional notation: If u_e is an upper bound we impose on the flow x_e then the overflow o_e with respect to x is defined as:

$$o_e = \begin{cases} x_e - u_e & (x_e > u_e) \\ 0 & (x_e \leq u_e) \end{cases}$$

If $o_e > 0$ we call e an *overflow edge* with respect to flow x.

The algorithm works as follows. Let e_0 be an overflow edge ($o_{e_0} > 0$) with respect to flow x^0 without any overflow edges below e_0. Then we place a proxy in the corresponding node i_0 and compute a new flow x, *assuming x_{e_0} vanishes: the total request of the corresponding subtree becomes zero ("complete caching").* Observe that the flow only changes in edges along the path from i_0 to ∞: for each of these edges e we set $x_e := x_e - x^0_{e_0}$. Next, we find a new overflow edge with respect to this updated flow and repeat until no overflow edges exist in $\mathcal{T}$.

Claim: *Assuming the budget constraint is not violated, this algorithm determines a minimal number of nodes where a proxy must be installed in order to be able to output "yes".*

Argument: Consider an overflow edge e which has no overflow edge below it. Since the flow in this edge has to be reduced to get a feasible solution, we have to place at least one proxy below e. The maximal reduction of the load in e by placing proxies below e is equal to the current load of e. However, since the placement of a proxy of arbitrary large size in the node corresponding to e leads to this reduction, it will be optimal to place the proxy there. These observations imply that for each overflow edge that has no overflow edge below it in $\mathcal{T}$, a proxy must be installed at the corresponding node. Moreover, it follows from our approach that the demand from nodes where a proxy is placed vanishes. The proof of the claim now follows by induction. □

Remarks:

- This algorithm determines the actual nodes where a proxy is located, which gives us a value of the fixed costs of the cache assignment. Indeed, if this sum of fixed costs exceeds budget B, we stop and output: no feasible solution found.
- Moreover, it gives lower bounds on the cache sizes. Indeed, putting file j of size s_j in the cache of node i_0 decreases x_{e_0} by

$$s_j \sum_{\text{relevant } i \preccurlyeq i_0} \lambda_{i,j},$$

  which is a decrease of $\widehat{\lambda}_{i_0,j} := \sum_{\text{relevant } i \preccurlyeq i_0} \lambda_{i,j}$ per unit cache. (Relevant $i \preccurlyeq i_0$ means: i below or equal to i_0 such that there is no proxy in between i and i_0.) Let τ_{i_0} be an enumeration of the files j in order of decreasing $\widehat{\lambda}_{i_0,j}$. Then the minimal needed cache size equals $\sum_{k=1}^{d-1} s_{\tau_{i_0}(k)}$, where d is minimal such that $\sum_{k=1}^{d} \widehat{\lambda}_{i_0,\tau_{i_0}(k)} s_{\tau_{i_0}(k)} > o_{e_0}$. (Without complete caching below i_0 more i may become relevant (probably depending on j), and one easily verifies the minimal needed cache size in node i_0 only increases.)

- The given algorithm may be modified by assuming that the placement of a cache at a node does not vanish the complete flow, but only a certain percentage x (this seems to be more realistic). In this case we will loose the 'minimality property' but it may be easier to solve Steps P2) and P3).

3.2.2 Determine the sizes of the proxies and the stored files

First, let us deal with the complexity of these subproblems.

Claim: Given the nodes where proxies are installed, computing the minimum total size necessary to achieve $x_e \leq u_e$ is NP-hard.

Argument: We will prove the claim by a reduction from the partitioning problem. Let n numbers $a_1, \ldots, a_n$ with $\sum_{i=1}^{n} a_i = 2b$ be given. The partition problem consist of answering the question whether or not a partition of $\{1, \ldots, n\}$ into two subsets S_1, S_2 with $\sum_{i \in S_1} a_i = \sum_{i \in S_2} a_i = b$ exists. We may reduce an instance of the partition problem to the following cache assignment problem: Given are n leaves $1, \ldots, n$ which are all connected to a source s and this source s is connected to the root. Leave i requests only for a file f_i of size a_i and source s has no request. For the edge (s, ∞) we define an upper bound $u = b$ and all other edges have upper bounds which are large enough. Furthermore, cache may only be installed at the source s. It is straightforward to see that it is possible to achieve a feasible solution with a total amount of cache equals b if and only if the partition problem has a feasible solution. □

Thus, problem P2) is unlikely to be solvable exactly by a polynomial time algorithm.

In view of the result above, we will solve P2) & P3) simultaneously by a greedy heuristic. Generally stated our heuristic works as follows. Put files in cache, one at a time, so as to maximize the relative total overflow reduction in each step. Proceed greedily until the overflow on each edge in $\mathcal{T}$ is reduced to 0. This implies a cache size allocation. If this allocation has costs exceeding budget B we return "no", otherwise we return "yes" with the corresponding flow x.

A more detailed description is as follows. Recall that an overflow edge is one with $o_e > 0$. The relative overflow reduction $r_{i_0,j}$ is the total reduction of overflow caused by putting one unit size of file j in the cache at node i_0. This depends on both the frequency $\widehat{\lambda}_{i_0,j}$ of requests on file j that arrive in node i_0, and the multiplicity $\mu_{i_0,j}$ counting the number of overflow edges above node i_0 up to the next proxy containing j. More precisely, putting file j (of size s_j) in cache at node i_0 reduces total overflow with $r_{i_0,j} = \mu_{i_0,j}\widehat{\lambda}_{i_0,j}$.

Thus, in each step of the algorithm, the current situation is given by:

- The set of files cached per proxy so far;
- The set of overflow edges;
- The frequencies $\widehat{\lambda}_{i_0,j}$ of file j at node i_0:

$$\widehat{\lambda}_{i_0,j} := \sum_{\text{relevant } i \preccurlyeq i_0} \lambda_{i,j},$$

  where for a fixed file j, the summation is over those nodes i below or equal to i_0 such that j is not cached in between i and i_0;

- The relative overflow reductions $r_{i_0,j}$ of putting file j in the proxy at node i_0:

$$r_{i_0,j} = \mu_{i_0,j}\widehat{\lambda}_{i_0,j}.$$

Now we can identify a node i and a file j for which $r_{i,j}$ is maximal, store this file j in the cache at node i and iterate.

Finally, the amount of cache in each proxy i is computed as

$$y_i = \sum_{j \text{ cached in } i} s_j.$$

Observe that only here the file sizes come in. Finally, our algorithm returns the total costs

$$\sum_{\text{proxies } i} k(y_i).$$

Of course, as the number N of files involved can be a fairly high number (depending upon the specific situation) this algorithm should be carefully implemented. Let us now suggest an efficient implementation of an updating step in the algorithm. Recall that $\mu_{i_0,j}$ is the number of overflow edges in between node i_0 and the first node above i_0 where j is cached. We therefore consider two files at node i_0 *equivalent* if the first node on the path from i_0 to the root, where these files are cached, is the same. For each i_0, we order equivalence classes into lists according to decreasing frequencies $\widehat{\lambda}_{i_0,j}$. Note that ($i_0$-)equivalent files j_1, j_2 have the same multiplicity: $\mu_{i_0,j_1} = \mu_{i_0,j_2}$.

After we put file j in proxy at node i_0, the update procedure now consists of:

- Finding the right equivalence class for file j in nodes below node i_0;
- updating frequencies of j at certain nodes above i_0;
- in case for some original overflow edge e the overflow reduces to 0, changing the multiplicity $\mu_{i,j}$ on those equivalence classes to whose multiplicity e contributed before.

To illustrate the updating, we now provide an example.

Example: *Suppose we have the following current situation in node E of the depicted infrastructure of Fig. 2.*

$\mu_{E,j}$	*contributing overflow edges*	*nearest proxy containing j*	*j (in order of decreasing $\widehat{\lambda}_{E,j}$)*
4	$A', (B'), C', D', E'$	∞	*permutation of* $[6], \ldots, [N]$, *say* [10] *(28)* ; [6] *(23)* ; ...
2	D', E'	C	[1] *(60)* ; [5] *(50)*
1	E'	D	[2] *(100)*
0	*none*	E	[4] *(0)* ; [3] *(0)*

Then

$$\begin{aligned} r_{E,[10]} &= 4 \cdot 28 = 112, \\ r_{E,[1]} &= 2 \cdot 60 = 120, \\ r_{E,[2]} &= 1 \cdot 100 = 100 \text{ and} \\ r_{E,[4]} &= 0 \cdot 0 = 0, \end{aligned}$$

so [1] *is a candidate to be put in stack at node E. However, there may be a proxy i_0 with even bigger maximal $r_{i_0,j}$.*

- *Suppose $i_0 = D$ has biggest maximal $r_{i_0,j}$, viz. for $j = [5]$(130) (with $\mu_{D,[5]} = 1$, whence $r_{D,[5]} = 130$). Putting* [5] *in the proxy at D has the following effect on the lists in node E (below $i_0 = D$), assuming — say — that A' and D' become overflow free:*

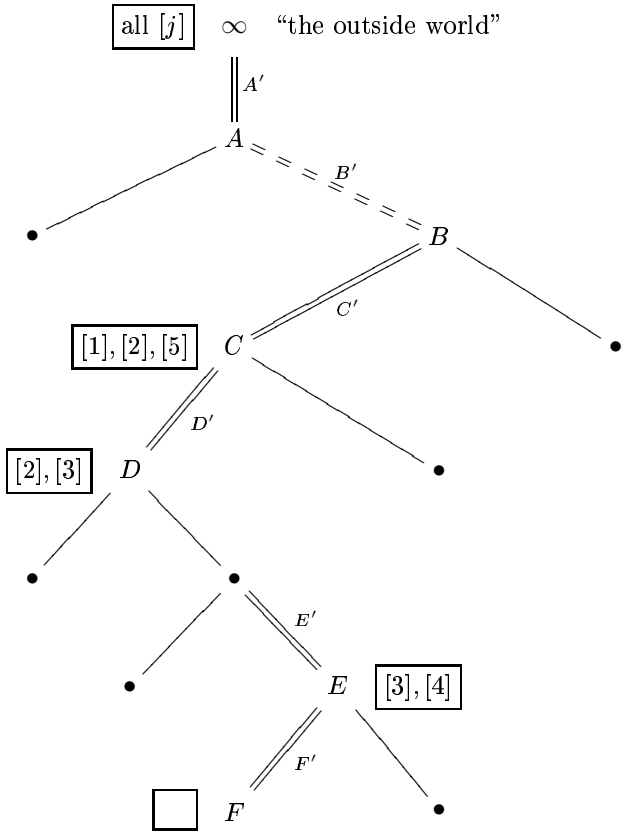

Figure 2: An infrastructure with original overflow edges A', B', C', D', E', F', current overflow edges A', C', D', E', F', proxies installed in C, D, E, F, currently containing the depictured files $[j]$.

$\mu_{E,j}$	*contributing overflow edges*	*nearest proxy containing* j	j *(in order of decreasing* $\widehat{\lambda}_{E,j}$*)*
2	$(A'), (B'), C', (D'), E'$	∞	[10] *(28)* ; [6] *(23)* ; ...
1	$(D'), E'$	C	[1] *(60)*
1	E'	D	[2] *(100)* ; [5] *(50)*
0	*none*	E	[4] *(0)* ; [3] *(0)*

- *Suppose* $i_0 = F$ *has biggest maximal* $r_{i_0,j}$*, viz. for* $j = [10]$*(25) (with* $\mu_{F,[10]} = 5$*, whence* $r_{F,[10]} = 125$*). Putting* $[10]$ *in the proxy at* F *has the following effect on the lists in node* E *(above* $i_0 = F$*), assuming — say — that* A' *and* D' *become overflow free:*

$\mu_{E,j}$	*contributing overflow edges*	*nearest proxy containing* j	j *(in order of decreasing* $\widehat{\lambda}_{E,j}$*)*
2	$(A'),(B'),C',(D'),E'$	∞	[6] *(23)* ; ... ; [10] *(3)* ; ...
1	$(D'),E'$	C	[1] *(60)* ; [5] *(50)*
1	E'	D	[2] *(100)*
0	*none*	E	[4] *(0)* ; [3] *(0)*

4 Stochastic Analysis

So far we analyzed the caching problem using deterministic methods. We tackled the question *where* to cache specific files. In practice, however, no local caching strategy would cache *specific* files. Instead, one often uses the so-called Least Recently Used (LRU) strategy, which operates as follows:

- whenever a request arrives for a file that is not in cache, this file is cached,
- whenever the total size of the files in cache exceeds the cache capacity, the least recently used files are dropped.

In the following subsections we present a stochastic analysis of the caching problem that takes into account the LRU strategy. By this analysis we will estimate the expected loads on the edges of the network that result from a given assignment of cache to the nodes. Thus, using these results we may get a better view on the quality of the solutions achieved by the methods presented in the previous section.

4.1 Problem Formulation

As in the deterministic analysis, we consider the web-caching problem on a infrastructure $T = (V, E)$ with node (vertex) set V and edge set E. We assume there is a fixed number N of files in the network that can be requested by users, and associated with each node of the tree is a *fixed* number of files that can be stored simultaneously in its cache. Note that, for convenience, we neglect the fact that different files can have different sizes.

We are given the following parameters:

- $\lambda_{i,j}$, the frequency of arrivals of external requests for file j at node i;
- λ_i, total frequency of external request arrivals at node i (thus $\sum_{j=1}^{N} \lambda_{i,j} = \lambda_i$);
- $p_{i,j} = \frac{\lambda_{i,j}}{\lambda_i}$, the probability that a request arriving at node i is for file j;
- M_i, the number of files that can be simultaneously stored at node i.

The aim is to determine the *expected* load x_e on edge e, for all $e \in E$ when the LRU strategy is used. This is done by calculations that start at the leaves of the tree and successively working our way up to the root of the tree. For each node i, we first compute the total arrival rate $\bar{\lambda}_{i,j}$ of requests for file j at i, which also includes the requests received from all descendants of i. From a Markov chain analysis we then obtain the total rate of requests leaving the node (which equals the expected load of the edge incident on it in the direction of the root). In the next subsections we shall illustrate the Markov chain analysis.

4.2 Markov chain analysis for a single node

In this subsection we analyze the cache at a single node. From now on, we suppress the index i. Thus requests for files arrive with rate λ and the probability that some request is for file j is p_j, independent of all other requests. After a new request of a file, which is not in the cache, arrives at a node, the file is placed in the cache and, simultaneously, the least recently used file is removed from the cache. The state of the cache can be described by the files contained in the cache and the order in which they have recently been used; thus the total number of different states is equal to $\binom{N}{M} M! = N(N-1)\dots(N-M+1)$. Clearly, this process is a discrete-time Markov chain. The stationary probability that file j is present in cache is denoted by $p_j^{(M)}$.

A closed form expression for the limiting distribution of the Markov chain can be obtained by using the fact that the LRU strategy for caching is closely related to the "move-to-the-front rule", which has extensively been studied in the context of selecting records from a computer file and taking out books from a library shelf. For example, in [5], Hendricks considers the following problem. A library contains N different books $B_1, \dots, B_N$ arranged on a single shelf, and regardless of the arrangement of the books the probability of selecting B_j is p_j, where p_j $(1 \le j \le N)$ are positive numbers such that $\sum_{j=1}^{N} p_j = 1$. Only one book is demanded each time and the book is returned as the next book is borrowed. Upon return, a book is placed at the end of the shelf nearest to the librarian's desk. What is the stationary distribution of the order of books on the shelf?

Let $\tau = (j_1, \dots, j_N)$ be a permutation of $\{1, \dots, N\}$. The Markov chain has $N!$ states, corresponding to the $N!$ different permutations of the books. The state $B(\tau)$ corresponds to the books being arranged in the order, from left to right, $B_{j_1}, \dots, B_{j_N}$ on the shelf. Let u be the equilibrium probability of state $B(\tau)$. Hendricks proved that

$$u = \prod_{n=1}^{N} (p_{j_n} / \sum_{r=n}^{N} p_{j_r}).$$

The shelf corresponds to the cache in our case, while books correspond to files. Hendricks' result can be applied if we only take into consideration which books are in the first M positions and how they are ordered. Let $\pi = (j_1, \dots, j_M)$ be an ordered M-tuple, whose entries are distinct members of the set $\{1, \dots, N\}$, and let v be the equilibrium probability of the corresponding state of our Markov chain. Then

$$v = \sum_{\sigma} \prod_{n=1}^{N} (p_{j_n} / \sum_{r=n}^{N} p_{j_r}),$$

where the summation is over all permutations $\sigma = (j_{M+1}, \dots, j_N)$ of the set $\{1, \dots, N\} \backslash \{j_1, \dots, j_M\}$.

Unfortunately, the above closed-form expression is not likely to be useful in practice, since calculating its value involves a summation of $(N-M)!$ terms. Typically $N - M = \Omega(N)$ (that is, $(N-M)/N$ is bounded below by a positive constant as $N, M \to \infty$), and thus $(N-M)!$ grows exponentially in N and M. This implies that evaluating the formula quickly becomes computationally infeasible. Therefore, we propose the following approximation for $p_j^{(M)}$:

$$p_j^{(M)} = 1 - (1 - p_j)^z, \tag{1}$$

where z solves the equation

$$M = N - \sum_{j=1}^{N} (1 - p_j)^z. \tag{2}$$

Equation (2), which determines z, follows from summing equations (1) over all j and using the fact that $\sum_j p_j^{(M)} = M$, as in equilibrium the cache should be full. Furthermore, the number z represents the expected number of steps required to pick M distinct files and hence $1 - (1 - p_j)^z$ can be interpreted as the probability that file j has been selected during the z steps required to form the current contents of the cache.

4.3 Expected loads on the edges of the tree

Finally, we show how to calculate the expected loads on all the edges of the tree. For each node i and each file j, let $\bar{p}_{i,j} = \bar{\lambda}_{i,j}/\bar{\lambda}_i$ be the probability that a given request at node i (either external or from one of the descendants of node i) is for file j. Clearly, if node i is a leaf of the tree then $\bar{p}_{i,j} = p_{i,j}$ for $j = 1, \ldots, N$. For each leaf i, we calculate $p_{i,j}^{(M)} = 1 - (1 - \bar{p}_{i,j})^{z_i}$, where z_i solves $M_i = N - \sum_{j=1}^{N}(1 - \bar{p}_{i,j})^{z_i}$. The rate of requests leaving leaf i equals $\lambda_i' = \bar{\lambda}_i \sum_{j=1}^{N} \bar{p}_{i,j}(1 - p_{i,j}^{(M)})$, and hence the rate of request arrivals for a node i at height 1 from the bottom is $\lambda_i + \sum_r \lambda_r'$, where the sum is over all nodes r that are descendants of node i. We proceed recursively in the manner described above to higher levels of the tree, until we reach the root. The rate of requests leaving a node equals the expected load on the edge leaving the node towards the root.

5 Conclusion

In this paper we have considered the problem of placing proxies in a network to get a better performance of the net. We have divided this problem into two subproblems: identify nodes where proxies will be placed and determine the size of the proxies. To make the problems easier to handle, first we have simplified the problems by neglecting the stochastic structure of the process resulting from the caching strategies used in practice. Based on the assumption that fixed files are placed in the proxies, we have developed algorithms to determine locations for and sizes of the proxies.

Since the estimated quality of the resulting proxy placement is calculated using the deterministic model, it may be not very realistic. To overcome this, in a second step we have presented a stochastic analysis for the commonly used LRU caching strategy to achieve a more realistic estimate of the quality of the solutions. This analysis may be combined with the deterministic algorithms in an iterative procedure: First, on the base of given bounds on the loads of the edges, calculate a solution which in the deterministic model achieves these bounds on the loads. Afterwards, analyze the solution using the stochastic method. Based on the outcome of this analysis, change the used bounds on the loads and iterate the procedure.

References

[1] C. Aggarwal, J.L. Wolf and P.S. Yu, Caching on the World Wide Web, IEEE Transactions on Knowledge and Data Engineering 11 (1999), 94 – 107.

[2] H. Braun and K.C. Claffy, Web traffic characterization: an assessment of the impact of caching documents from NCSA's web server, Computer Networks and ISDN Systems 28 (1995), 37 – 51.

[3] R. Cáceres, F. Douglis, A. Feldmann, G. Glass and M. Rabinovich, Web proxy caching: the devil is in the details, Performance Evaluation Review 26 (1998), 11 – 15.

[4] S.J. Caughey, D.B. Ingham and M.C. Little, Flexible open caching for the Web, Computer Networks and ISDN Systems 29 (1997), 1007 – 1017.

[5] W.J. Hendricks, The stationary distribution of an interesting Markov chain, Journal of Applied Probability 9 (1972), 231 – 233.

[6] A. Luotonen and K. Altis, World wide web proxies, Computer Networks and ISDN Systems 27 (1994), 147 – 154.

[7] C. Maltzahn, K.J. Richardson and D. Grunwald, Performance issues of enterprise level web proxies, Performance Evaluation Review 25 (1997), 13.

[8] E. Markatos, Main memory caching of web documents, Computer Networks and ISDN Systems 28 (1996), 893 – 905.

[9] J. Shim, P. Scheuermann and R. Vingralek, Proxy cache algorithms: design, implementation and performance, IEEE Transactions on Knowledge and Data Engineering 11 (1999), 549 – 562.

[10] J. Wang, A survey of web caching schemes for the internet, ACM Computer Communication Review 29 (1999), 36 – 46.

[11] C.E. Wills and M. Mikhailov, Towards a better understanding of Web resources and server responses for improved caching, Computer Networks 31 (1999), 1231 – 1243.