Available online at www.sciencedirect.com
SCIENCE DIRECT
Computer Communications 28 (2005) 640–653
computer communications
www.elsevier.com/locate/comcom

Optimizing content distribution through adaptive distributed caching

Peter Backx*, Thijs Lambrecht, Bart Dhoedt, Filip De Turck, Piet Demeester

Department of Information Technology (INTEC), Ghent University-IMEC, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium

Received 5 August 2004; accepted 5 August 2004
Available online 11 September 2004

Abstract

Web caching is a widely used technique to decrease Internet traffic and server load, as well as to reduce client-perceived response times. Most currently deployed caches are almost isolated entities, with very limited cooperation between them. In this paper, we use the power of active networking to implement a much more fine-grained cooperation. Active networking allows to intercept and monitor passing request packets and to place, on demand, a cache at virtually every node in the network, giving a broad range of possible caching locations and offering considerable content placement opportunities. A scalable and fast heuristic is proposed for inter-cache cooperation. The performance of the heuristic is evaluated through simulations, implemented on a test network and shown to be optimal for tree topologies.

Keywords: Active networks; Web caching; Cooperative caching; Content distribution

1. Introduction

The goal of web caching is threefold: to decrease Internet traffic, to reduce the load on servers and to decrease the latency as perceived by the users. Substantial improvements are already visible with the current widespread use of proxy caches. However, caches currently are mostly individual entities that either do not cooperate or cooperate in a very basic fashion. Traditionally, caches can only monitor traffic at a single point in the network and cannot optimize traffic with a more general network-wide scope. For this reason cooperative web caching was introduced, in which caches are connected together and exchange information, usually about the pages they are caching. Although, cooperative web caching might not always be suitable (Ref. [1] shows that traditional cooperative proxy caching is only beneficial at a small to medium-sized city scale), we propose a finer level of cooperation, with reduced network overhead and lower computational complexity than many other cooperative cache architectures [2–5].

With our Adaptive Distributed Cache (ADC) architecture we aim at providing a solution that can quickly and automatically adapt to new situations and request patterns that can scale to whatever size of cooperation and network is needed and that can be easily and incrementally deployed in existing networks. In contrast to most conventional schemes where caches are located at the edges of networks, we use the flexibility of active networks [6] and propose to deploy caches throughout the network. Active networks enable protocols to automatically install code and state on the routers inside the network, enabling them to host caching functionality. Caches inside the network have been studied and a number of advantages have been shown [7–9]. Section 2 presents a much more detailed overview of related work.

Active networks provide us with several benefits. First, instead of statically installing caches at nodes inside the network caches can be installed on demand when a first request to cache data at that point is made. Secondly they allow to intercept and monitor packets (in this case HTTP request packets) without additional changes to existing network protocols. Third, the active approach allows for a lightweight multicast protocol that is used for coordination between caches without the need to rely on IP multicast (which is not deployed at large scale) or unicast (which incurs substantial additional network overhead). Furthermore, the software, such as the caching heuristic, can be updated on the fly.

* Corresponding author. Tel.: +32 9 264 9991; fax: +32 9 264 9960.
E-mail address: peter.backx@intec.ugent.be (P. Backx).

0140-3664/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2004.08.007

Content delivery companies can deploy the ADC architecture in order to enhance the performance of their content delivery networks by reducing overall network traffic, used cache space and load on servers.

The remainder of this paper is structured as follows: Section 3 investigates the caching problem and gives an Integer Linear Programming (ILP) formulation that can be used to calculate the optimal solution in principle. This approach, although yielding optimal solutions, is only applicable to relatively small networks due to its computational complexity. Section 4 details a heuristic enabling our adaptive and distributed caching approach built on the concept of pushing web pages from the server into the network towards the optimal caching location. The performance both in terms of network cost as well as concerning incurred network delay are investigated. Section 5 addresses the performance of an implementation of this heuristic by comparing experimental results to simulations and to optimal solutions calculated by solving the ILP problem. Furthermore, the influence of parameters specific to web traffic on the performance is analyzed.

2. Related work

Although considerable work has been done in the area of designing cooperative caching architectures, these efforts are focused on cooperating edge caches, usually with at most two or three layers of hierarchy. The Internet Cache Protocol (ICP) [10] is one of the earliest protocols that enables cooperation between web caches. On a cache miss, an ICP query is sent to all caches at the same level in the hierarchy. The cache then waits for a hit from one of these sibling caches or until all siblings have reported a miss. In the latter case the request is sent up the hierarchy, either to a parent cache or to a server. The disadvantage of this protocol lies in both the additional network load that is generated on every cache miss and the additional delay that is introduced when waiting for a reply from sibling caches.

Cache Digests [2], implemented in the Squid web proxy [3], addresses these issues. On regular intervals caches exchange a summary of their contents and in this way a cache can know what pages are available in the other caches. While this approach partly solves the drawbacks of ICP, the main disadvantage here lies in the fact that a cache does not necessary have an up-to-date view of the contents of its sibling caches. Therefore, a cache could falsely assume a sibling has a page, while actually this page has already been removed. Also the way in which the cache digests are built (using Bloom filters) can cause false hits [4].

Both Cache Array Routing Protocol (CARP) [11] and Web Caching with Consistent Hashing [5] use hash values of requests to select a cache responsible for caching the requested page. Because there is a direct mapping between a request and a cache location, a request can be immediately routed to the correct cache (although that cache might still not have that page available). Furthermore pages will only be cached once, freeing up extra cache space. It is very important that the hash function is carefully designed such that the requests are evenly distributed over all caches. It is also necessary to make sure that when a cache is added as few pages as possible will map to a different cache (preferably only the pages that will be handled by the new cache should change mapping).

Cisco's Web Cache Coordination Protocol (WCCP) [12] also uses hash values, but at a lower layer. A router intercepts HTTP requests and uses the IP address of the destination to calculate a hash value which is then used to decide which cache will handle the request. The advantage is that caches themselves are offloaded because the hash calculations are now spread over the routers. However, since routers have only access to the destination IP address all pages on the same server will be cached by the same cache.

A number of studies [1,13,14] are available on the performance of these cooperative cache architectures. However, since caches are deployed at the edge, the cost of retrieving a document from a cache different from the local cache is almost as high as getting it from the server, both in terms of network usage and network delay (in the assumption that the network is not congested). Consequently, the conclusion of these studies is that cooperation between edge caches has very little influence on the client perceived latency. However, the main rationale for adopting cooperative caching lies in the fact that network traffic is distributed between caches and server, avoiding server hot spots (and associated delays and congestion).

Recent research [15,16] on hierarchical cooperative cache systems has shown advantages when caches are placed en-route between client and server. The cache hierarchy and routing is constructed such that, a request that is sent higher up in the cache hierarchy is also sent closer towards the server. This means the penalty of a cache miss (both in network usage and additional latency) will be a lot smaller compared to the traditional architectures discussed above. Zhang et al. [17,18] use the same idea, but also add adaptability to the architecture. The layers and relations in the hierarchy are automatically adapted to the network situation. Their approach resembles the techniques that will be discussed in this paper, however, the usage of multicast groups greatly complicates the protocols necessary to make this cache architecture work. [8] takes this another step further and places caches inside the network at routing nodes. The ability to cache at virtually any node in the network reduces the latency considerable even when caches are relatively small, however, the influence on the network usage is not investigated. [7,9] also place caches inside the network, but only a limited number of caches is placed at optimal locations. Both conclude that caching inside the network can produce a substantial network traffic reduction even compared to traditional proxy caches at the edge,

but no solution is offered how caches can be dynamically relocated (when optimal locations change due to changing request patterns).

In Ref. [19], Iyer et al. discuss distributed cooperative caching at a different layer. Instead of cooperation over the Internet, cooperation over a local LAN is proposed. In fact the architecture could replace the local proxy cache with smaller caches at every users PC. They showed a performance equal to a large centralized proxy cache as soon as individual cache sizes are around 100 MB, with the added benefit that their system is much more fault tolerant then a centralized cache.

A number of papers have already used active networking as a means to implement their proposed web caching architecture, such as the above mentioned [8]. The main benefit of active networking is that any node on the network can be used as a cache because it can monitor and process packets and has storage space available for caching. Ref. [21] uses active networks to route packets towards correct caches. This has the advantage that there is no need for storage space on the actual active network, but the disadvantage that packets incur additional delays when they are routed of the shortest path from client to server. Ref. [20] is one of the most recent papers to investigate the applicability of Active Networking to web caching, but only tries to adapt the existing two layer cooperative caching architectures.

3. The caching problem: optimal solution

Fig. 1 shows a general overview of the one server caching problem. A server contains a number of pages (p_1,…,p_n) that can be downloaded by the clients. Each client requests a number of these pages with a certain frequency. They connect to the server through a network of routers and/or active nodes. The problem at hand is to minimize the cost for delivering the requested pages to the clients. If no caches are used, pages are sent from server to client via the shortest path and the cost is the network cost associated with using the network links to transmit the pages to the appropriate clients. However, if an active network is considered, all the nodes inside the network are active and can be used as caches to store copies of one or more of the pages on the server. Storing pages inside the network will reduce the network cost because pages are closer to the clients. However, cached pages need to be reloaded from the server to the cache whenever they are changed on the server. This incurs an extra cost, which we call the *refresh cost*.

The goal now is to determine whether and where it is useful to cache a certain page. This optimization can be based on a range of criteria. Most important are the latency perceived by clients and the load on server. Reduced latency for the clients will give users quicker access to the information they need, while reduced server load will allow a server to handle more clients with the same hardware and network connection.

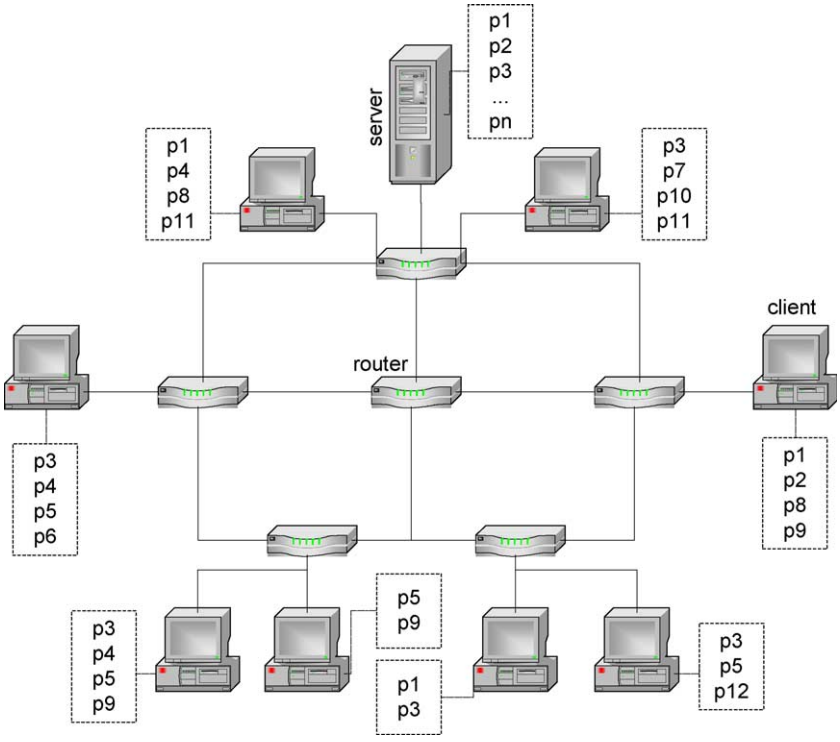

Fig. 1. The caching problem: a server offers a number of pages for download. The pages offered are shown in the box connected to the server (p_1,…,p_n). The clients download some of the pages (the boxes next to the clients show the requested pages) through a network of routers/active nodes. These can be used as caches in order to minimize network usage and client-experienced latency.

In our work we focus on optimizing the network traffic, because this is the fundamental driving factor behind both these criteria. With reduced network traffic and caches in the network, the pages will be cached closer to the user and experience less congestion, which results in shorter delays for the clients. Globally minimizing network traffic will also reduce the traffic to and from the servers and thus the server load.

3.1. Formal problem formulation

3.1.1. Notations

Given a network G with a topology consisting of a set of nodes V of size $|V|$, connected by a set of edges E of size $|E|$. With every edge $e \in E$ corresponds a cost per unit of bandwidth $c(e)$. For this paper we presume that bandwidth is not the restricting factor or, in other words, the capacity of the edges is unlimited. This also means that caches can load pages along the shortest path to the server. The cost of the shortest path from the server to node n is $K(n)$. With every node $n \in V$ corresponds a capacity $v(n)$ which indicates the size of the cache in node n. One of the nodes is selected to be the server node s. It holds a set of pages P of size $|P|$. Every page $p \in P$ is characterized by its size $s(p)$ and its refresh rate $r(p)$. Finally we have a set of client nodes D of size $|D|$. With every client $d \in D$ and each page $p \in P$ corresponds a request rate $f(d,p)$ (which may be zero if the client does not request that page).

All variables are binary:

- $z(n,p)$ is 1 iff node n is used for caching page p.
- $h(d,p,e)$ is 1 iff edge e is used to send page p to client d.
- $z'(n,d,p)$ is 1 iff node n is used for caching page p needed by client d. In other words, client d gets page p from the cache n. For every (client, page)-pair there is only one cache location (see constraint (2)).

There are $|V|\cdot|P|$ z-variables, $|D|\cdot|P|\cdot|E|$ h-variables and $|V|\cdot|D|\cdot|P|$ z'-variables.

3.1.2. The objective function

To optimize the network traffic, two factors should be considered: first the *request cost*, i.e. the network traffic generated by actually sending the file to a requesting client either from a cache or from the server and secondly the refresh cost, which is the network traffic caused by updating files in the caches that have been changed on the server.

The objective function that needs to be minimized shows these two components:

$$\sum_{e\in E}\sum_{p\in P}\sum_{d\in D} c(e)s(p)f(p,d)h(d,p,e) + \sum_{n\in V}\sum_{p\in P} z(n,p)s(p)r(p)K(n) \quad (1)$$

The first term is the request cost: a request for page p by client d will cause network traffic to increase by an amount of $c(e)s(p)$ on every edge e between the client and the cache (or server) that serves the request for that client and that page. The second term is the refresh cost. As discussed previously the shortest path (with a cost of $K(n)$ per unit of bandwidth) can be used for transferring pages from server to caches.

The variables are subject to a number of constraints: the server node s is certainly a cache and must contain all pages:

$$z(s,p) = 1, \quad \forall p \in P \quad (2)$$

For every page p requested by client d (i.e. $f(d,p) \neq 0$), there has to be exactly one node that caches the page for that client (this node can also be the server):

$$\sum_{n\in V} z'(n,d,p) = [f(d,p) \neq 0 ? 1 : 0], \quad \forall d \in D, \quad \forall p \in P \quad (3)$$

If a page is cached in a node for one or more clients (i.e. $z'(n,d,p) = 1$ for at least one page p requested by a client d at cache n) we need to set up a cache in that node.

$$z'(n,d,p) - z(n,p) \leq 0, \quad \forall d \in D, \quad \forall p \in P, \quad \forall n \in V \quad (4)$$

There should be a path from client to cache or server for every page that a client requests. Thus if an outgoing edge of a node n is used for a page p for client d, also an incoming edge should be used for that page and client, unless this is the source node (i.e. either the server or a cache).

$$\sum_{e\in \mathrm{out}(n)} h(d,p,e) - \sum_{e\in \mathrm{in}(n)} h(d,p,e) = z'(n,d,p), \quad \forall d \in D, \quad \forall p \in P, \quad \forall n \in V\backslash\{d\} \quad (5)$$

Requested pages arrive at the client but are not forwarded further.

$$\sum_{e\in \mathrm{in}(d)} h(d,p,e) + z'(d,d,p) = [f(d,p) \neq 0 ? 1 : 0], \quad \forall p \in P, \quad \forall d \in D \quad (6)$$

$$\sum_{e\in \mathrm{out}(d)} h(d,p,e) = 0, \quad \forall p \in P, \quad \forall d \in D \quad (7)$$

The capacity of the nodes is restricted:

$$\sum_{p\in P} z(n,p)s(p) \leq v(n), \quad \forall n \in V \quad (8)$$

3.2. Solution through integer linear programming

Each of the above constraints can be written in a formal way as follows:

$$\mathbf{L}_i \leq \mathbf{C}_i \cdot \mathbf{x} \leq \mathbf{U}_i, \quad i = 1, \ldots, 7 \quad (9)$$

where $\mathbf{x}$ represents the one-dimensional vector of all concatenated decision and auxiliary variables of the problem.

$\mathbf{C}_i$ is a two-dimensional matrix with as many columns as there are z variables and as many rows as there are constraints, stated by formula (2)–(8) above. $\mathbf{C}_i$ is typically a sparse matrix. $\mathbf{L}_i$ and $\mathbf{U}_i$ are, respectively, the lower and upper bound one-dimensional matrices for the ith constraint. An ILP optimization routine has been developed, which first calculates the numerical values of the seven constraint matrices $\mathbf{C}_i$ ($i=1,\ldots,7$) for the particular problem instance, together with the coefficients of the variables in the objective function (1). The optimal values for the decision variables $z(n,p)$, $h(d,p,e)$ and $z'(n,d,p)$ are then calculated, using a Branch and Bound based ILP solution approach [22]. Note that the auxiliary variables need to be calculated as well during the optimization process. From the found values of the decision variables, the optimal caching location for the considered problem instance can be easily deduced.

While this formulation enables us to calculate the optimal solution, actually finding a solution does have a number of disadvantages:

- All information is needed in advance: not only the refresh rates of the pages at the server but also the request rate of the users. Since this is impractical, this optimal solution cannot be used in a real situation (on could try and estimate request rates). It does, however, give an indication of 'where the caches should have been placed'. That is why results obtained by ILP are used as benchmark for the heuristic (see Section 4).
- A lot of processing power, time and memory is needed because of the large constraint matrices, coefficients and variables.
- All necessary data needs to be gathered at a central point, thus becoming a single point of failure.

An advantage of this approach is that it does not only look for caching locations along the shortest path from client to server (unlike the heuristic that is discussed in Section 4). It can also use caches that are not on the route from client to server. Although this allows for additional optimization, it does complicate protocol design, if one were to use this approach on a real network.

4. Practical solution to the caching problem through a heuristic

In view of the practical problems associated with using an ILP based solution, a heuristic was developed to solve the problem. Section 4.1 looks at a solution for one very specific case, which is then expanded into a general distributed heuristic in Section 4.2. Finally the incurred delay by using en-route caching is analyzed. This is an important point because every additional en-route cache adds an additional delay on a cache miss.

4.1. Optimal caching location in a regular tree

Fig. 2 shows the simplified network we are considering in this section. Two simplifications are made. First the degree of each node is constant: every cache is connected to a parent cache or the server and to exactly E child caches or clients. While the distance between clients and server is L hops. Secondly, every client requests a page with the same request rate λ. These symmetric assumptions have the consequence that if it is decided to cache a specific page at a certain cache in the tree, it should be cached at all the caches at that level in the tree. Requests are always forwarded along

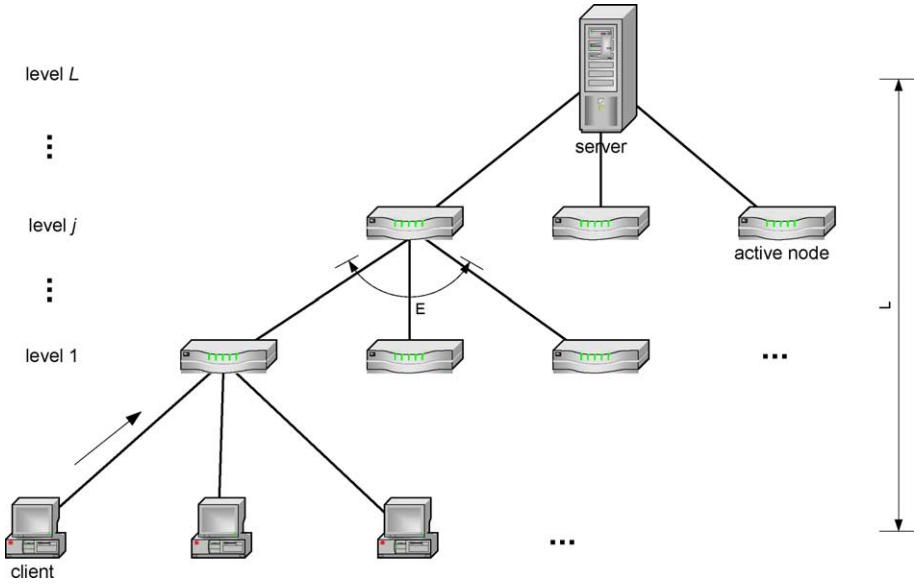

Fig. 2. A distribution tree with a constant out degree of E and height L. Every cache at level j is connected to exactly one cache at level $j+1$ (or to the server for those caches at level $L-1$) and to exactly E caches at level $j-1$ (or to E clients if the cache is at level 1). Clients request pages from the server with an average request rate λ.

the shortest path up the tree and sibling caches (caches at the same distance from the server) are not queried to keep overhead low.

The request cost for a page cached at level j and requested λ times per second by each client is given by:

$$B_{\text{req}} = jE^L\lambda \tag{10}$$

In Eq. (10), the cost for using a link is considered to be the same over the entire network and equal to 1, implying that a single request for a page will cost j (in general the cost for using a link is not necessarily exactly 1 and can be chosen at will, however, this only multiplies the equations by a constant factor and obviously does not affect the conclusions).

The refresh cost for a page cached at level j which needs to be refreshed μ times per second is dependent upon the number of location that it is cached at, which in turn depends upon the level at which the page is cached.

$$B_{\text{refr}} = (L-j)E^{(L-j)}\mu \tag{11}$$

The total network cost B is given by the sum of these two and an example of this is shown in Fig. 3. Clearly there is an optimal caching level at which the total network cost is minimal. By calculating the differential we can obtain an expression for the optimal caching level j_0:

$$\left.\frac{\Delta B}{\Delta j}\right|_{j=j_0} = E^L\lambda - E^{L-j_0}\mu[L-j_0-(L-j_0+1)E] = 0 \tag{12}$$

Implying:

$$\frac{\lambda}{\mu} = \frac{(L-j_0)(E-1)+E}{E^{j_0}} \tag{13}$$

We denote the request rate seen by a single cache at level j as λ_j, and hence:

$$\lambda_j = E^j\lambda \tag{14}$$

Therefore,

$$\frac{\lambda_{j_0}}{\mu} = (L-j_0)(E-1)+E \tag{15}$$

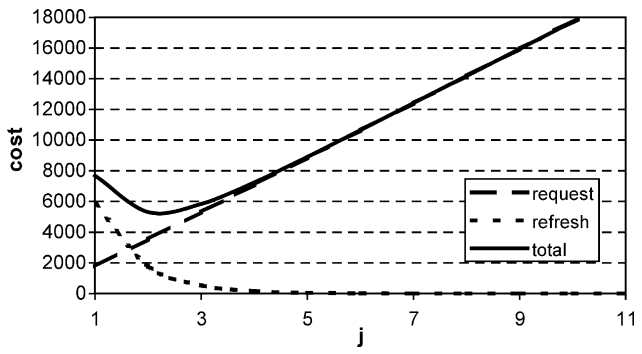

Fig. 3. The costs involved with caching a page at level j: request cost, refresh cost and total cost for $L=11$, $E=3$ and $\lambda/\mu=1$.

This expression defines a threshold value for caching pages, based on the local request rate λ_j and refresh rate μ.

4.2. Threshold push heuristic for en-route caching

The Threshold Push Heuristic is inspired by the results obtained in the previous section. Every cache is actually a transparent en-route cache, meaning that it can monitor the requests passing by and build a database of request rates for all passing pages. If the page is cached in this cache, it will send the reply to the client and not forward the request. The heuristic itself works on one very basic principle: If the total request rate for a page exceeds a given threshold, the cache will send a request to all adjacent caches to start caching this particular page thereby reducing the load on this cache.

Because transparent en-route caching is used we only have to consider the distribution tree from server to clients. While the network might have cyclic paths and cross-connections, the heuristic only works along the distribution tree, which is made up of the shortest paths from the server to all the clients. In principle no data is sent on any of the links that are not part of the distribution tree, however, in practice it is not always possible to know exactly what links are part of what distribution tree (it is, of course, different for every server). Such knowledge can be built up by caches by monitoring on which links requests have been received. This does mean that in the initial phase some superfluous message might be exchanged, however, this has no long-term effect on performance. Fig. 4 shows a small part of this tree. It is important to note that this distribution tree is not a regular tree such as the one in the previous Section 4.1. When minimizing network traffic, the influence of two parameters should be taken into account: the rate at which clients request pages and the refresh rate, which is the number of times a page has to be reloaded from the server, because it was changed.

Fig. 4 illustrates the threshold push heuristic: a cache at level j will have to decide if it is cheaper (in terms of network cost) to push the pages towards level $j-1$. This can be done be comparing the network costs B of caching at level j and $j-1$:

$$B_j = d_c(j)\lambda_j + d(j)\mu, \quad B_{j-1} = d_c(j-1)\lambda_j + Ed(j-1)\mu \tag{16}$$

With $d(j)$ the distance between the cache at level j and the server and $d_c(j)$ the average distance between that cache and the clients and E the number of caches connected to the cache at level $j+1$. These two equations can be written down for every page on the server.

A page should be pushed if it is cheaper to cache it at a lower level, thus whenever the following inequality holds:

$$(E-1)d(j)+E < \frac{\lambda_j}{\mu} \tag{17}$$

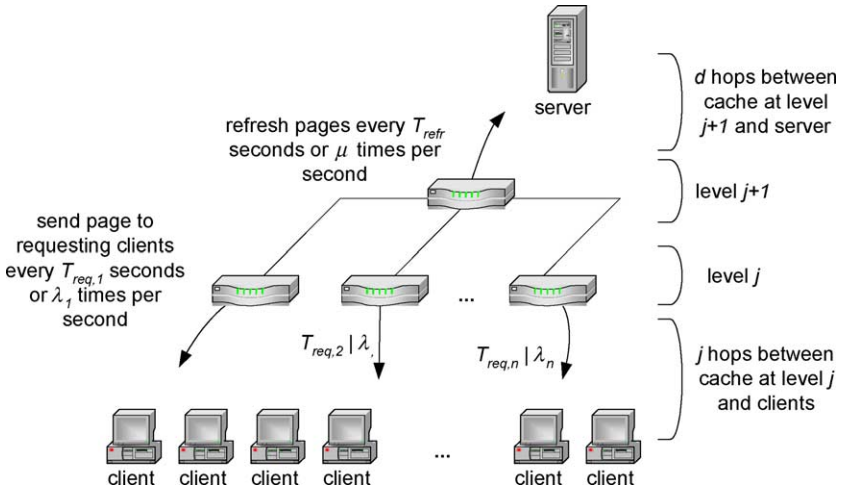

Fig. 4. Threshold push heuristic. The node decides whether or not to push a page further into the network based on the refresh and request frequencies.

(since in a tree topology $d_c(j-1)=d_c(j)-1$ and $d(j-1)=d(j)+1$)

By using this mechanism, the pages will gradually move to the aggregation point where there is a balance between the price to fulfill the requests being made and the cost of refreshing the page.

It is now easy to conclude that indeed the general threshold defined in Eq. (17) pushes the pages towards the optimal caching location for the specific regular distribution tree that we considered in Section 4.1. The push condition can be simplified because $d(j)=L-j$:

$$(E-1)(L-j)+E<\frac{\lambda_j}{\mu} \tag{18}$$

If a page is cached at level j this means that the caches at level $j+1$ pushes the page, while those at level j do not:

$$\begin{cases}(E-1)(L-j-1)+E<\dfrac{\lambda_j}{\mu}\\ (E-1)(L-j)+E\geq\dfrac{\lambda_j}{\mu}\end{cases}\Leftrightarrow(E-1)(L-j-1)+E<\frac{\lambda_j}{\mu}\leq(E-1)(L-j)+E\,((A)<(B)\leq(C)) \tag{19}$$

Clearly the formula for the optimal caching location in Section 4.1 holds for these conditions. Fig. 5 shows the different quantities that we have discussed in this section.

4.3. Delay reduction

The caching architecture and the active network that we use will incur extra delays in every node in the network. In order for our caching architecture to be viable it should, at the very least, not be slower then the usual proxy cache. That is why this section analyzes the speedup. The speedup is the ratio of the time T_{proxy} to get a page without cooperative caching (but with a single proxy cache) and the time $T_{\text{cooperative}}$ to fetch a page with cooperative caching (such as our adaptive distributed caching architecture):

$$S=\frac{T_{\text{proxy}}}{T_{\text{cooperative}}} \tag{20}$$

T_{proxy} is the average time to get a page when only local proxy caching is used. This value only depends on the hit ratio of the proxy cache and the time it takes to get the request:

$$T_{\text{proxy}}=h_0T_{\text{local}}+(1-h_0)T_s \tag{21}$$

with

- h_0 the hit ratio of the local proxy cache
- T_{local} the average time to get a requested page from the local proxy cache
- T_s the average time to get the page from the server.

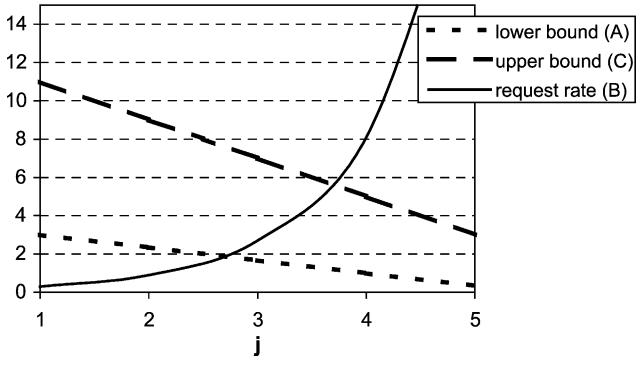

Fig. 5. THP bounds and request rate. If the ratio between request rate and refresh rate exceeds the upper bound the cache at level j will push the page (such is the case for caches at levels 4 and 5 in the figure). If it is bellow the lower bound the page should not have been pushed (such as for caches at levels 1 and 2). The values used to create the above figure are: $E=3$, $L=5$ and $\lambda/\mu=0.1$.

To calculate $T_{\text{cooperative}}$ we make the following assumption:

The time to get a page from a cache increases linear by the hopcount, and fetching a page from a cache i hops away will take:

$$T_i = T_{\text{local}} + iT_{\text{step}} \tag{22}$$

with T_{step} the average increase in response time between caches one hop away.

Getting a page from the server, L hops away, with our architecture will take $T_{\text{local}}+LT_{\text{step}}$, which is most likely higher than T_s, since extra processing needs to be done at every node.

Our architecture first searches the local proxy cache and then forwards the request along the shortest path towards the server, checking each cache at every hop. If the requested web page is found in a cache it is sent to the client. With the above assumption in mind $T_{\text{cooperative}}$ is:

$$T_{\text{cooperative}} = \left(\sum_{i=0}^{L-1} h_i m_i (T_{\text{local}} + iT_{\text{step}})\right) + m_L(T_{\text{local}} + LT_{\text{step}}) \tag{23}$$

with

- h_i the hit rate of the cache i hops from the local cache.
- $m_0 = 1$ and $m_i = \prod_{j=0\cdots(i-1)}(1-h_j)$
- L the number of hops between the local cache and the server.

The speedup is now given by:

$$S = \frac{h_0\left(\frac{T_{\text{local}}}{T_s} - 1\right) + 1}{\left(\sum_{i=0}^{L-1} h_i m_i \left(\frac{T_{\text{local}}}{T_s} + iT_{\text{step}}\right)\right) + m_L\left(\frac{T_{\text{local}}}{T_s} + L\frac{T_{\text{step}}}{T_s}\right)} \tag{24}$$

We now introduce another assumption:

The cache hitrate h is the same for all caches. This is a rather strict assumption. The literature is not clear on this point. Ref. [16] (among others) predict and measure a substantial higher hit ratio for caches higher up in the hierarchy, while others, such as Ref. [13], have the exact opposite results. While caches higher up in the hierarchy will not receive requests for popular pages that are already cached lower in the hierarchy, they do benefit from the fact that they receive an aggregated stream of requests and thus server a larger population with more requests and possibly more pages that are beneficial for caching. Our simulations show that this assumption does not always hold in practice. It is dependent on the exact request and refresh rate distributions, the individual cache sizes and the public interest (or document sharing). Especially the latter is important in this regard and this is investigated in Section 5.5.2.

The speedup can now be further simplified: $m_i = (1-h)^i$ and $h_i = h$

$$S = \frac{h\left(\frac{T_{\text{local}}}{T_s} - 1\right) + 1}{\left(\sum_{i=0}^{L-1} h(1-h)^i \left(\frac{T_{\text{local}}}{T_s} + i\frac{T_{\text{step}}}{T_s}\right)\right) + (1-h)^L\left(\frac{T_{\text{local}}}{T_s} + L\frac{T_{\text{step}}}{T_s}\right)} \tag{25}$$

We now use the following estimates:

- $T_{\text{local}}/T_s = 1/20$. This value is based on Ref. [1], which estimates the last byte latency of server response at 1.9 s and the latency of a proxy cache hit at 10 ms. Thus, our estimate is rather pessimistic and the overall performance of our caching architecture will most likely be better than the results shown below.
- $L = 19$. Monitoring measurements performed in the framework of this study on a test infrastructure of 346 web servers have shown an average hopcount of 19.4.

Fig. 6 demonstrates the speedup for different values of T_{step}/T_s. At $T_{\text{step}}/T_s = 0.2$, this means that it will take almost four times longer to get a page from the server when our cooperative caching architecture is used than when only a single proxy cache is used:

$$T_{\text{s,caches}} = \frac{T_s}{20} + \frac{19T_s}{5} = \frac{77}{20}T_s = 3.85T_s$$

Although, in reality, the increase in response time will very likely not be this high, it does demonstrate the potential of our architecture, which outperforms a basic proxy cache as soon as the cache hit ratio exceeds 0.2. When the cache hit ratio gets very high the advantage of the caching architecture is lost, because almost all pages will be cached at the local proxy cache.

Fig. 7 shows the maximum achievable speedup for different values of T_{step}/T_s and the hitrate to achieve this

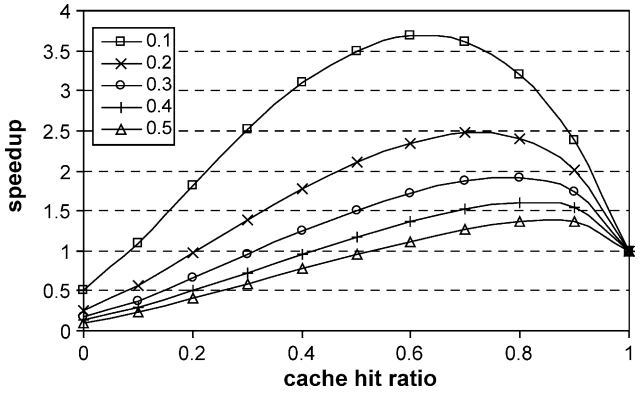

Fig. 6. Speedup versus cache hit ratio for values of T_{step}/T_s ranging between 0.1 and 0.5.

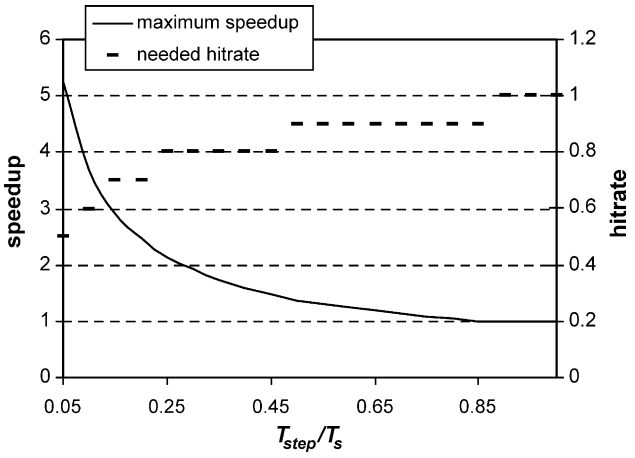

Fig. 7. Maximum achievable speedup and hitrate needed at the caches to achieve that speedup.

maximal speedup. At high values (and thus long additional delays in every node) the achievable speedup is very small and the hitrate needed is high.

While we have made a number of assumptions and estimates, we can conclude from this section that it is possible to suffer a substantial additional delay on every hop and still, on average, deliver the requested pages quicker.

5. Simulations and experimental results

5.1. Protocol design

A protocol implementing the THP heuristic was designed to evaluate the performance on an actual test network. Packets on an active network are usually called capsules, because they not only contain data but also code. The protocol itself consists of three types of capsules:

- The most basic capsule is the *data capsule* that transports web pages from one node to another; there is no need for active functionality here. In the actual implementation data capsules are only used to set up a TCP connection between the client and the data source (either the server or a cache), in order to reliably transfer the page.
- The *request capsule* contains the largest part of the protocol code: it is sent from the requesting client towards the server along the shortest path. The capsule will update the request rate at each node that has an active cache and it checks the cache database if the page is available.
- The *push capsule* is broadcasted by a cache to its neighbors whenever the request rate of a page has reached the threshold. The broadcasting can either be done through some existing network support or can be implemented as an active service.

The refresh rate can also be estimated from monitoring data, but it is easier and more convenient to use the HTTP header fields that are provided by the server, such as the 'cache-control' fields, which is what the implementation uses.

This protocol was simulated in the JavaSim simulation environment [24] and implemented using the ANTS execution environment [26]. Both use an identical protocol that is empowered by the functionality offered by the active nodes:

- Most importantly is of course the memory and/or disc storage space to store monitoring data and cached pages.
- A minimal amount of processing power is required for lookup operations and for calculating the request rate.
- Services that exploit this functionality can be installed dynamically and on demand. This greatly simplifies the deployment of the proposed service as the other alternative would be to statically install the caching service on every node that is part of the caching network.
- In principle, transparent (for the user) interception of HTTP get packets becomes possible. However, in practice, this is not as straightforward. This is discussed in more detail in Section 5.2.
- Push capsules are broadcasted using a lightweight broadcast protocol that can be implemented very easily using active networks. Relying on existing solutions would make this a lot harder: IP multicast is not deployed at large scale and unicasting to all recipients generates a lot of network traffic overhead.

5.2. Implementation details

While the simulation can assume that the entire network is active, in practice this is not feasible. To circumvent the routers that are not active the implementation uses an overlay network. Fig. 8 shows how this works: the TCP-network transports HTTP requests and HTTP data replies while the active network is overlaid on top of the existing network to add the flexibility required by the adaptive cache protocol. Proxies ensure the interaction between those two networks, translating between the HTTP protocol and our own active cache protocol, based on ANTS capsules. Other active network architecture offer support for intercepting packets and processing them. For instance, AMnet is, in this regard, a better choice. However, this framework has only recently been developed and was not available at the time.

However, the use of proxies is a solution that considerably simplifies development of the protocol, because once the HTTP request packets are converted to request capsules on the network, not all routers have to be active and not all active nodes need to have a cache installed for the cache architecture to work. An active request capsule will be forwarded to a proxy close to the server, where it is translated back into a HTTP request. Once on the active network, a request can be monitored and intercepted by caches. It should be noted that finding a proxy 'close' to

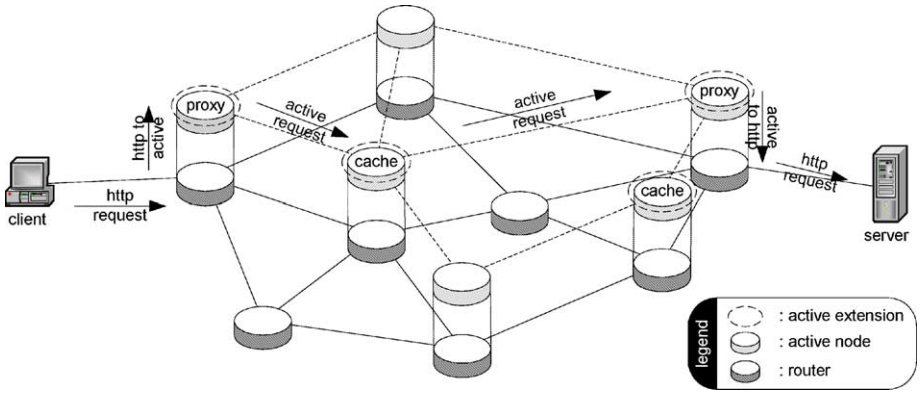

Fig. 8. Active network overlaid on an IP network. Also shown is the route a request takes from client to server.

the server is not an easy task. The easiest solution, which is used in this paper, is to install an active node on every server that wants to use the ADC network. Active requests can than be forwarded towards the server where the proxy will intercept them and convert the packets back to regular HTTP requests. However, more complicated solutions are also possible such as using a protocol that calculates distances between proxies and servers and selects the closest one. Such a solution requires a more complicated protocol, because proxies that translate HTTP requests into active packets need to send the active request towards the correct proxy.

Fig. 9 gives an overview of an active node. It has the basic functionality of a router (the IP forwarding block) but it will intercept any packets that are labeled active and forward them to the correct execution environment in which the code that the packets carry is executed. For the cache implementation we use the ANTS 1.31 execution environment [26], which is a general-purpose execution environment with support for active extensions. These are chunks of code that can be preloaded on the node to avoid excessive loading times The API offered by the execution environment to its active packets is then enriched by these extensions. Both caches and proxies are implemented as active extensions.

Active nodes enter the cache network by downloading and executing the proper extension. Once installed, the cache will automatically start monitoring every passing request for stored pages. A first push request will trigger the installation of the active caching extension. If no extensions are available it is also possible to make direct use of the soft state cache that is available in practically any execution environment. However, the extension offers an easier and more performing solution because there is no need to send the full code with the request capsules. Because caches are deployed dynamically whenever they are needed there is no lengthy manual setup procedure. Caches do not need to know the exact place of their neighboring nodes and whether they already have an active extension installed because push packets can be broadcasted over the active network. Active nodes will forward those push packets towards all their neighbors until they reach an active node that is either willing and able to join the cache network or where a proxy extension is installed. In the former case, a cache extension will be installed (if necessary) and the node will start caching the requested page. In the latter, the push packet will be discarded.

Storage space has become cheap so we have not considered the impact of limited cache sizes until now. If cache space becomes an issue pages can be removed by a least frequently used (LFU) cache replacement policy. This way, pages that have become unpopular and for which the request rate has become too low are not pushed back towards the server, but are simply purged from the cache. If such a page becomes popular again it will have to pushed back from the server towards the cache. While this generates a certain overhead, the old page would probably be out-of-date anyway, so keeping it in some cache is most likely a waste of resources.

While one might argue that the proposed solution could be implemented by upgrading routers this is actually not

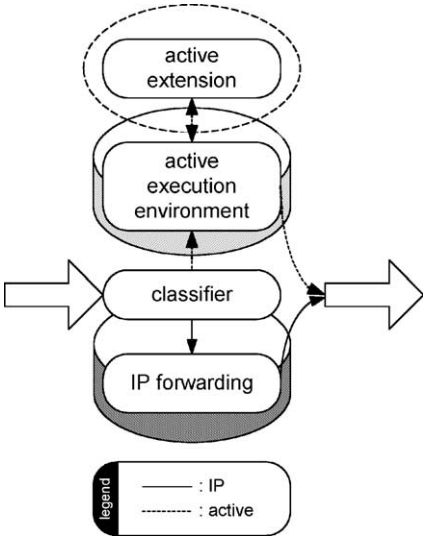

Fig. 9. Active node. Incoming packets are classified. Non-active IP-packets are forwarded as usual, while active packets are executed in the correct execution environment.

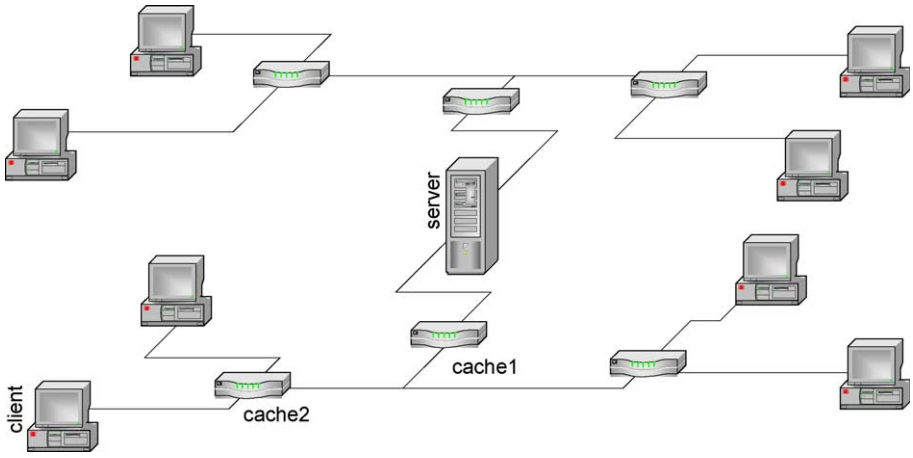

Fig. 10. Basic test network. Eight clients connect through six caches to a server.

a very scalable and practical technique. Exactly here the active networking approach shines by allowing us to dynamically deploy more caches on the network when needed.

Furthermore, as previously mentioned, active networks give us several more advantages:

- Intercepting of packets, which is vital for monitoring requests. ANTS does give us a disadvantage here because this is not provided in the framework, hence the need for proxies. In this regard, the net-centric approach of, for instance, AMnet 2.0 [27] is better suited. However, AMnet 2.0 was not available at the time of this research.
- Broadcast. Again it could be argued that most routers already have multicast functionality, however, this is very rarely enabled (especially across multiple service providers domains). Active Networks pretty much gives us multicasting for free, especially the very basic needs we have (only one hop towards the neighbors).

5.3. Testing configurations

For the tests two networks are used:

- *Network A* is shown in Fig. 10. It is a small test network and is used for testing and comparing the implementation and the ILP solution on an actual testbed.
- *Network B* is a much larger network consisting of 1000 nodes. It was generated using the generalized linear preference model [28]. This model generates a graph closely resembling the power law and characteristic path length exhibited in the actual Internet topology [23]. The network is structured as a tree, so there are no cyclical paths. There are 660 leaf nodes, which are used as clients. Because of its size, this network could only be analyzed through simulations.

5.4. Heuristic evaluation

Fig. 11 compares for network *A* both the implementation and simulation of the threshold push heuristic with the optimal solution provided by solving ILP problem. The server has 26 pages, each of size 1 kB, that are served to each of the eight clients. The actual page size does not influence the performance of the heuristic, but does influence the performance of the network and cache database components. Every page remains unchanged for a random period between 4 and 20 min, after which it becomes stale and needs to be refreshed. Every client has a random subset of these 26 pages from which it regularly requests a random page. The implementation was tested and benchmarked using Web PolyGraph [29]. The request sequence it generated were used by the CPLEX ILP solver and the JavaSim simulation to compare the performance.

We also compared the cost on a small irregular network. The network consists of 15 nodes, of which nine are clients, five caches and one is a server. The average distance between client and server is 2.9 hops.

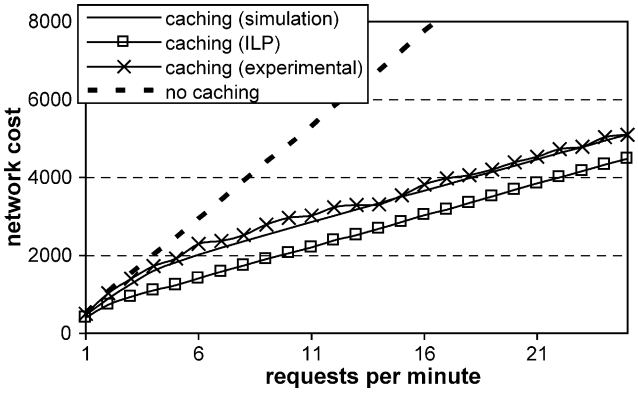

Fig. 11. Network cost comparison between the ILP solution (gray), the implemented and simulated THP heuristic (black) and a network without caches (dashed).

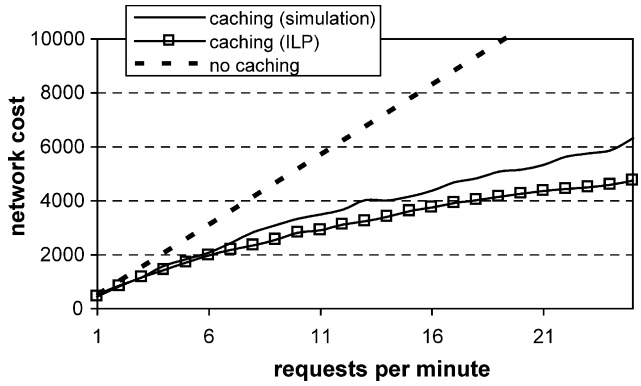

Fig. 12. Network cost comparison on an irregular 15 node network.

Figs. 11 and 12 show a typical result of a 20 min experiment in which we increased the number of requests by clients from 1 to 25/min. The total network cost (1 unit/kB) is shown as function of the request rate. Compared to the case when no caching is used the heuristic gives a typical bandwidth cost reduction of 50% while it is typically 20% more expensive in terms of network cost when compared to the optimal ILP solution. The latter difference is caused by the fact that the heuristic estimates the request rate based on the current observations, while the ILP solution can correctly calculate the request rate because it is executed after all measurements have been done.

While Fig. 11 depicts the steady state solution, Fig. 13 shows the evolution of cache loads as a function of time.

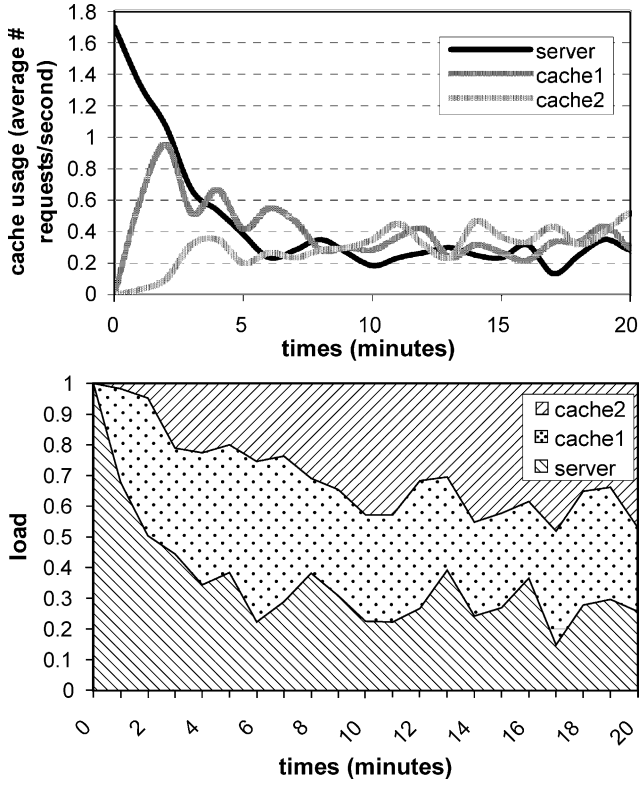

Fig. 13. Dynamic evolution of the load on server and caches: absolute (*top*) and relative (*bottom*).

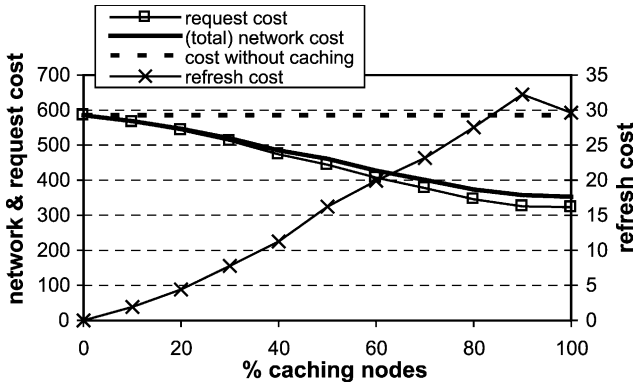

Fig. 14. Influence of the number of caches on the performance of the caching architecture.

The top figure shows the absolute values for a server offering 100 pages with a 4-min refresh period and clients randomly requesting 20 pages per minute. Again *Network A* was used, with cache 1 located on the level closest to the server and cache 2 closest to the clients. By pushing excessive load from the server into the network and thus caching popular files load balancing is achieved. The bottom figure shows this even clearer in relative values. At first the server has to handle all the requests while later in the experiment it only handles about 25%.

The simulations presented in Fig. 14 investigate the behavior of the caching architecture on a more realistic network and use the 1000 nodes Internet-alike *Network B*. In any realistic situation the number of caches will be limited for various reasons, such as the availability of sufficient CPU and memory resources at the nodes. Until now we have assumed that all nodes in the network are candidate locations for caches. In this simulation the number of caches was varied from 0 to 100% of all routing nodes. Caches were chosen randomly and one server offered 500 pages that were requested according to a Zipf-like distribution with α-value of 0.75 (which is a value commonly found in web traffic [25], also see the experiment in Section 5.5.1). Fig. 14 shows the average results for 20 simulations. Although the performance decreases with fewer caches, there still is a substantial gain in network cost even for few deployed caches.

Throughout this paper the size of caches is not considered to be of any major importance and it is assumed that cache sizes can hold all pages that need to be cached at that location or caches can safely discard old content without impacting the network cost. This simulation, using the same parameters as the previous simulation, investigates whether this assumption is valid. A standard LFU cache replacement policy was implemented and the size of the caches was varied. Fig. 15 shows the average result for 20 simulations. As soon as the caches can hold about 20% of the available pages there is not much performance improvement. At cache sizes smaller than 5% of the available page set the interaction between the LFU

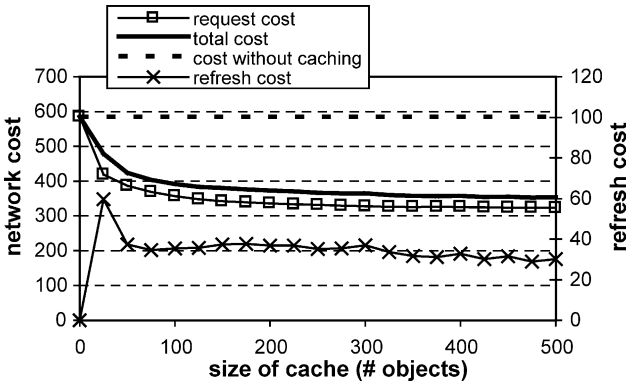

Fig. 15. The influence of cache size on the network cost. The cache size is measured in the number of constant length pages it can hold.

replacement policy and THP heuristic tends to become somewhat unstable and pages are cached and removed from the cache too frequently, which accounts for the increased refresh cost.

5.5. Influence of typical web parameters

In this section the influence of two major parameters in web traffic is investigated. Both the parameter of the Zipf-like distribution and the amount of public interest directly influence the distribution of requests for web pages and thus it is important to see their effect on our caching architecture.

5.5.1. Zipf-like distribution

As has been demonstrated in multiple studies ([1,25]), the requests for pages follow a Zipf-like distribution, in which the number of requests for the ith most popular page is proportional to $1/i^{\alpha}$ for some constant α, which usually is situated between 0.6 and 0.8 [18].

To investigate the influence of α on our cache architecture we set up a server with 100 pages, each with a 4 min refresh time and let the clients request 20 pages/min, and the pages are chosen according to a Zipf-like distribution. Fig. 16 is the result of this experiment for network A. Clearly, the performance of our architecture is relatively constant, no matter the exact α value.

The dashed line is an indication of the cache cost, representing the total amount of cached pages during

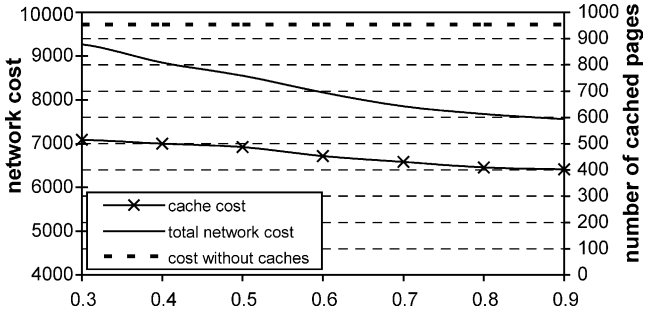

Fig. 16. Influence of the Zipf parameter α on the performance of the cache architecture.

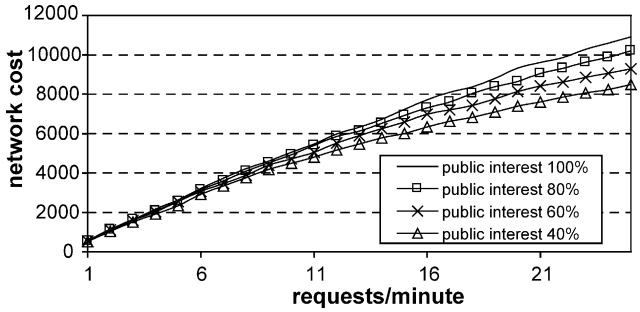

Fig. 17. The performance of the cache architecture for different values of public interest.

the experiment. With low α values more pages need to be cached to obtain a gain in network cost, which is what is to be expected.

5.5.2. Public interest

One major objection against cooperative caches is that the requested pages are too distinct between clients [1]. This phenomenon is commonly referred to as the public interest of a client or page sharing between clients. This parameter represents the percentage of pages requested by more than one client. Clients with 50% public interest will request 50% of their pages out of a private set that are not requested by other clients, while the other 50% will also be requested by other clients (and possibly by all clients).

For an experiment on network A, the number of available pages was kept at 100. Somewhat counterintuitive results are shown in Fig. 17: the performance actually increases with decreasing public interest.

This is explained by the fact that with increasing public interest and a fixed number of total pages on the one server, the actual number of different pages that clients will request decreases, which allows our caching architecture and heuristic to function more optimal.

The 100 pages offered by the server can be split into a public set and a number of private sets. Every client has its own private set (hence the name) and it will request pages from both the public set and his private set. Table 1 shows the sizes of these sets (all private sets are equally large because of the symmetric nature of our test setup). For a small public interest ratio clients will only request from a small set of pages, however, their total request rates will remain the same. So for small public interest request rates by individual clients for a small number of pages will be

Table 1
Public interest influence on working set size

Public interest (%)	Size of public set	Size of private set	Total working set per client
100	100	0	100
80	33.3	8.3	41.6
60	15.8	10.5	26.3
40	7.7	11.5	19.2
20	3	12.1	15.2

high. These conditions are very well suited for our heuristic because the caches closest to the clients will be able to 'specialize' in the pages that their clients request.

6. Conclusion

A new caching architecture was presented as well as an implementation based on active networking technology. By allowing the network to host cache functionality a vast number of cache locations becomes available and it is possible to store content close to its optimal location for given user behavior and content characteristics. A heuristic was presented to find these optimal locations and through simulation and experimentation it was shown that this architecture works in a wide variety of situations and reductions in network traffic of up to 30% were achieved. The architecture continues to behave under typical web traffic parameters that other cooperative caching architectures do not handle very well, such as a high diversity in requested pages between clients.

Acknowledgements

The Measurement Factory and in particular Alex Rousskov are gratefully acknowledged for the Web Polygraph benchmark and the free support provided. Part of this work was carried out within the framework of the project CoDiNet sponsored by the Flemish Institute for the promotion of Scientific and Technological Research in the Industry (IWT) and by the Ghent University GOA-project 'Programmable and Active Networks' (PAN).

References

[1] A. Wolman, G.M. Voelker, N. Sharma, N. Cardwell, A. Karlin, H.M. Levy, On the scale and performance of cooperative web proxy caching, Operating Systems Review 34 (5) (1999) 16–31.
[2] L. Fan, P. Cao, J. Almeide, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol, in: Proceedings of SIGCOMM 1998.
[3] Squid web proxy cache, At URL: http://www.squid-cache.org.
[4] A. Rousskov, D. Wessels, Cache digests, Computer Networks and ISDN Systems 30, 1998.
[5] D. Karger, T. Leighton, D. Lewin, A. Sherman, Web caching with consistent hashing, in: Proceedings of the Eighth International World Wide Web Conference, Toronto, Canada, May 1999.
[6] K. Psounis, Active networks: applications, security, safety and architectures, IEEE Communications Surveys First Quarter (1999).
[7] P. Krishnan, D. Raz, Y. Shavitt, The cache location problem, IEEE/ACM Transactions on Networking October (2000).
[8] S. Bhattacharjee, K.L. Calvert, E.W. Zegura, Self-organizing wide-area network caches, Technical Report GIT-CC-97/31, 1997.
[9] P.B. Danzig, R.S. Hall, M.F. Schwartz, A case for caching file objects inside internetworks, CU-CS-642-93, March 1993.
[10] D. Wessels, K. Claffy, Internet cache protocol (ICP), version 2, At URL: http://www.ietf.org/rfc/rfc2186.txt.
[11] K. Ross, Hash-routing for collections of shared web caches, IEEE Network 11, 37–44, 1997.
[12] M. Cieslak, D. Foster, G. Tiwana, R. Wilson, Web cache coordination protocol v2.0. At URL: http://www.web-cache.com/writings/internet-drafts/draft-wilson-wrec-wccp-v2-00.txt.
[13] S.G. Dykes, K.A. Robbins, A viability analysis of cooperative proxy caching, in: Proceedings of IEEE Infocom, Anchorage, Alaska, April 2001.
[14] C. Lindemann, O.P. Waldhorst, Evaluating cooperative web caching protocols for emerging network technologies, in: Proceedings of Workshop on Caching, Coherence and Consistency (WC3′01), Sorrento, Italy, June 2001.
[15] H. Che, Z. Wang, Y. Tung, Analysis and design of hierarchical web caching systems, in: Proceedings of IEEE Infocom 2001, Anchorage, Alaska, April 2001.
[16] P. Rodriguez, C. Spanner, E.W. Biersack, Analysis of web caching architectures: hierarchical and distributed caching, IEEE/ACM Transactions on Networking August (2001).
[17] L. Zhang, S. Floyd, V. Jacobson, Adaptive web caching, in: Proceedings of the NLANR Web Cache Workshop, Boulder, Colorado, USA, June 1997.
[18] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, V. Jacobson, Adaptive web caching: towards a new global caching architecture, in: Proceedings of the Third International Web Caching Workshop, Manchester, England, June 1998.
[19] S. Iyer, A. Rowstron, P. Druschel, Squirrel: a decentralized peer-to-peer web cache, in: Proceedings of IEEE Infocom 2001, Anchorage, Alaska, April 2001.
[20] L. Lefèvre, J. Pierson, S. Guebli, Deployment of collaborative web caching with active networks, in: Proceedings of the International Workshop on Active Networks (IWAN) 2003, Kyoto, Japan, December 2003.
[21] U. Legedza, J. Guttag, Using network-level support to improve cache routing, in: Proceedings of the Third International WWW Caching Workshop, Manchester, England, June 1998.
[22] G.L. Nemhauser, A.L. Wolsey, Integer and Combinatorial Optimization, Wiley, New York, 1988.
[23] S. Jin, A. Bestavros, Small-world internet topologies, Boston University Technical Report BUCS-TR-2002–2004, January 2002.
[24] H. Tyan, The JavaSim simulation environment, At URL: http://www.javasim.org.
[25] L. Breslau, P. Cao, L. Fan, G. Philips, S. Shenker, Web caching and zipf-like distributions: evidence and implications, in: Proceedings of IEEE Infocom, 1999.
[26] D.J. Wetherall, Service introduction in an active network, PhD Thesis at the MIT, February 1999.
[27] T. Fuhrmann, T. Harbaum, M. Schöller, M. Zitterbart, AMnet 2.0: an improved architecture for programmable networks, in: Proceedings of the International Workshop on Active Networks (IWAN) 2002, Zürich, Switzerland, December 2002.
[28] T. Bu, D. Towsley, On distinguishing between power-law internet topology generators, in: Proceedings of IEEE Infocom 2002, New York, US, 2002.
[29] Web Polygraph, At URL: http://www.web-polygraph.org.