On the Optimization of Storage Capacity Allocation for Content Distribution*

Nikolaos Laoutaris, Vassilios Zissimopoulos, Ioannis Stavrakakis

December 18, 2003

Abstract

The addition of storage capacity in network nodes for the caching or replication of popular data objects results in reduced end-user delay, reduced network traffic, and improved scalability. The problem of allocating an available storage budget to the nodes of a hierarchical content distribution system is formulated; optimal algorithms, as well as fast/efficient heuristics, are developed for its solution. An innovative aspect of the presented approach is that it combines all relevant subproblems, concerning node locations, node sizes, and object placement, and solves them jointly in a single optimization step. The developed algorithms may be utilized in content distribution networks that employ either replication or caching/replacement. In addition to reducing the average fetch distance for the requested content, they also cater to load balancing and workload constraints on a given node. Strictly hierarchical, as well as hierarchical with peering, request routing models are considered.

1 Introduction

Recent efforts to improve the service that is offered to the ever increasing internet population strive to supplement the traditional bandwidth-centric internet with a rather non-traditional network resource – storage. This trend is evident in a number of new technologies and service paradigms. End-users employ local caches at their Web browsers or participate in peer-to-peer networks to share storage and content with other users. Organizations install dedicated proxy servers to cache popular documents that are shared among their local populations. Clusters of networks cooperate by allowing their proxy servers to communicate, forming large hierarchical caches such as the NLANR cache. Similar steps are taken regarding content provision/distribution. Popular web-sites employ reverse proxies for load balancing purposes or maintain multiple mirrors in geographically dispersed locations. Content Distribution Networks (CDNs) operate large numbers of servers that "push" the content of their subscribed clients close to the end-users. In all aforementioned cases, storage capacity (or memory) is employed to bring valuable information in close proximity to the end-users. The benefits of this tactic are quite diverse: end-users experience smaller delays, the load imposed on the network and on web-servers is reduced, the scalability of the entire content provisioning/distribution chain in the internet is improved.

*This work and its dissemination efforts have been supported in part by the IST Program of the European Union under contract IST-2001-32686 (Broadway). The authors are with the Department of Informatics and Telecommunications, University of Athens, 15784 Athens, Greece (email: laoutaris@di.uoa.gr; vassilis@di.uoa.gr; istavrak@di.uoa.gr).

In most cases the engagement of the memory resource has been done in an ad hoc manner. Organizations select the location and capacity of their dedicated proxy servers with little or no coordination with other organizations, even in the case that their proxies connect to the same hierarchical cache. Similarly, web-servers are replicated based on geographical criteria alone (e.g., US, Europe, Asia site) or on crude statistics (total number of requests coming from a certain network/region). The uncoordinated deployment of memory can seriously impair its effectiveness.

This paper attempts to answer the question of how to allocate a given storage capacity budget to the nodes of a generic hierarchical content distribution system. Such a system can materialize as any one of the following: a hierarchical cache comprising cooperating proxies from different organizations; a content distribution network offering hosting services or leasing storage to others that may implement hosting services on top of it; a dedicated electronic media system that has a hierarchical structure (e.g., video on demand distribution). The dimensioning of web caches and content distribution nodes has received a rather limited attention as compared to other related issues such as replacement policies [1, 2], proxy placement algorithms [3, 4, 5, 6] and request redirection mechanisms [7]. This observation seems to have been made by other researchers too in very recent publications [8]. In fact, the only published paper on the dimensioning of web proxies that we are aware of is due to Kelly and Reeves [9] whereas the majority of related works in the field have disregarded storage dimensioning issues by assuming the existence of infinite storage capacity [10, 11].

The limited attention paid to this problem probably owes to the fact that the rapidly decreasing cost of storage combined with the small size of typical web objects (html pages, images), make *infinitely large caches* for web objects realizable in practice, thus potentially obliterating the need for storage allocation algorithms. Although we support that storage allocation algorithms are marginally useful when considering typical web objects – which have a median size of just 4KB – we feel that recent changes in the internet traffic mix prompt for the development of such algorithms. A recent large scale characterization of http traffic from Saroiu et al. [12] has shown that more than 75% of internet traffic is generated by P2P applications that employ the http protocol, such as KaZaa and Gnutella. The median object size of these P2P systems is 4MB which represents a thousand-fold increase over the 4KB median size of typical web objects. Furthermore, the access to these objects is highly repetitive and skewed towards the most popular ones thus making them highly amenable to caching as demonstrated by the authors of [12].

Similar is the case with stored audio and video files distributed by either standard web servers or dedicated video on demand (VoD) servers. Such object are usually short videos (advertisements or news) and amount to around 1 MB [13]. Although a 1997 study [14] found that such traffic represents a rather limited percentage of the total aggregate traffic in the internet, the percentage was shown to have been increased significantly by a follow-up measurement study just a few years later [15]. This trend seems to have persisted, as recent studies (2001) have shown that stored video and audio files have by now become almost pervasive [13].

Such objects can exhaust the capacity of a cache or a CDN node, even under a low price of storage. Probably is just a matter of time before caching systems move on to accommodate P2P traffic as well as video traffic (either over http or dedicated protocols used by VoD servers) to reap the same benefits as with web objects, for the new class of applications that dominate the traffic. This has already been a reality in the case of CDNs and stored video files. Given the traffic /

storage requirements of such content it is highly doubtful that the "infinitely large cache" will be realizable, thus storage allocation algorithms such as the ones developed here might prove useful. Finally, another application of storage capacity allocation algorithms would be a wireless caching system, possibly employing some hierarchical structure (e.g., employing cluster head nodes which are given increased capacity as compared to ordinary nodes), where storage capacity would be a scarce resource even under typical web content.

2 Our approach towards storage capacity allocation

The current work addresses the problem of allocating a storage resource differently than previous attempts, taking into consideration related resource allocation subproblems that affect it. Previous attempts have broken the problem of designing a content distribution network into a number of subproblems consisted of: (1) deciding where to install proxies (and possibly their number too); (2) deciding how much storage capacity to allocate to each installed proxy; (3) deciding on which objects to place in each proxy. Solving each one of the problems independently (by assuming a given solution for the others) is bound to lead to a suboptimal solution, due to the dependencies among them. For instance, a different storage allocation may be obtained by assuming different object placement policies and vice versa.

The dependencies among the subproblems are not neglected under the current approach and, thus, an optimal solution for all the subproblems is concurrently derived, guaranteeing optimal overall performance. The current approach considers memory as a fluid commodity and decides how much of the "fluid" to allocate to each node of the system. When placing a fluid unit it also decides on its "kind", i.e., on the identity of the object that it will hold, thus combining the resource allocation decisions (where to place storage and how much) with the object placement decisions (more on this issue in Sect. 3.1).

Treating storage capacity as a fluid that can be provisioned using a very small granule serves both as a paradigm for the optimization of existing systems (e.g. to re-organize more effectively the allocation of storage in a hierarchical cache) but hopefully will be the approach to be followed in developing future systems. Indeed, if the provisioning of memory continues to materialize as it has in the recent past, then in the very near future memory pools (CDN nodes, or local proxy servers) will be in place in most systems that constitute the internet [16]. Building adaptive overlay content distribution systems on top of the underlaying memory pools can become a significant alternative to the static provisioning of memory as materialized with the current replication schemes that employ very large granules of memory (e.g., entire mirror site). Should memory pools exist and be marketed, content creators (or intermediaries) can build distribution systems on them by leasing storage capacity dynamically, as dictated by their needs. The main advantage of such a scenario is that memory will be utilized more efficiently, and at a finer granularity, as each potential user or application will be able to use it on-demand and release it when no longer necessary making it available to other users that may request and pay for it (protocols and e-currencies for such resource trade paradigms have been proposed recently [17]). Re-organizing the memory is not possible with the current instalment of dedicated mirrors and proxies in fixed locations and with fixed capacities. It is believed that the ability to reorganize the existing resources will be central to future intelligent information systems (see IBM's *autonomic computing* initiative [18]). In such environments, the proposed algorithms would be useful to regulate the utilization of storage in each memory pool, and do so fast enough that they may execute as frequently as required to address sudden changes of demand (transient “hot spot” demand).

The current work makes the following distinct contributions towards the realization of the above mentioned objectives:

- Introduces the idea of provisioning memory using a very small granule as an alternative to/extension of known paradigms (mirror placement, proxy placement) and formulates an optimal solution to this problem. The derived solution provides for a joint optimization of storage capacity allocation and object placement and can be exploited in systems that perform replication, as well as in those that perform caching.

- Develops fast efficient heuristic algorithms that approximate closely the optimal performance but can execute very fast, as required by self organizing systems (and as opposed to planning/dimentioning processes that can employ slow algorithms). Moreover these algorithms may be executed incrementally, thus obliterating the need for re-optimization from scratch.

- Supplements the algorithms for the optimization of storage capacity allocation with the ability to take into consideration load balancing and load constraints on the maximum demand that may be serviced by a node. Load balancing is an important issue in its own right [19, 20] and is jointly addressed here. The presented techniques may be used to avoid overloading some nodes while at the same time underutilizing others, as done by the well known “filtering” effect [20] in hierarchical caches. The filtering effect is addressed by allocating the storage, as well as the most popular objects, more evenly to the nodes of the hierarchy.

- Models and studies the effect of request peering, i.e., of the ability to forward request to peer nodes at the same level of the hierarchy, as opposed to pure hierarchical systems that only forward requests to upstream ancestor nodes.

The work focuses on hierarchical topologies. There are several reasons for this: (1) many information distribution systems have an inherent hierarchical structure owing to administrative and/or scalability reasons (examples include hierarchical web caching [21], hierarchical data storage in Grid computing [22], hierarchical peer-to-peer networks [23]); (2) although the internet is not a perfect tree as it contains multiple routes and cycles, parts of it are trees (due to the actual physical structure, or as a consequence of routing rules) and what’s more, overlay networks on top of it have no reason not to take the form of a tree if this is called for; (3) it is known that once good algorithms exist for a tree they may be applied appropriately to handle general graph topologies [24]. Figure 1 gives an example of how can a hierarchical content distribution overlay network be built upon a flat network of clients by using proximity/administration based clustering. Client nodes are depicted with solid black squares while storage nodes of the overlay content distribution infrastructure are depicted with empty squares. The left figure of Fig. 1 shows a possible proximity/administration-based clustering on the physical topology, and the right one the resulting three-level content distribution overlay (institution, ISP, and national levels). Such an hierarchical overlay may benefit by the algorithms developed here, despite the fact that the underlaying physical topology is not a tree.

The remainder of the article is organized as follows. Section 3 formally defines the storage capacity allocation problem and presents exact and approximate solutions for it. The same section contains a number of numerical results and a discussion concerning implementation issues. In

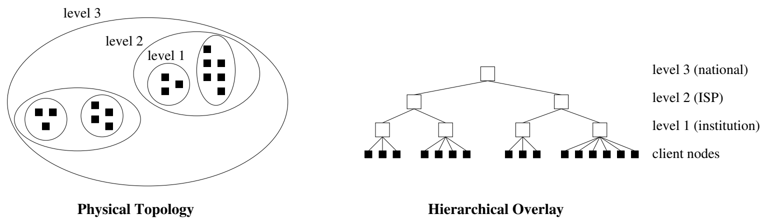

Figure 1: A three level hierarchical content distribution overlay built upon of a set of client nodes using proximity/administration based clustering.

Section 4 the algorithms (exact and approximate) are modified in order to be able to handle load balancing and request peering. A second set of numerical results, pertaining to the modified algorithms, is included. Related work in the field is presented in Sect. 5. Finally, Sect. 6 concludes the article.

3 The Storage Capacity Allocation Problem

3.1 Problem statement

The storage capacity allocation problem is defined here as that of the distribution of an available storage capacity budget to the nodes of a hierarchical content distribution system, given known access costs and client demand patterns. Since users request specific objects and not abstract storage units, a solution to the storage capacity allocation problem must include a solution to a related object placement problem. The coupling of the two makes this a challenging problem. Related works have overcome this difficulty by either provisioning entire web-server replicas [4], or by assuming that the hit rate of each installed proxy is a priori known [25]. In the first case each node holds all the content thus no object placement problem needs to be solved, nor there is cooperation between nodes (as in the case that nodes with potentially different content cooperate to handle local misses). This reduces to the well studied k-median problem in a tree graph [26, 27] for which exact solutions exist. The second case (known hit ratio) also leads to a k-median problem where only the locations for the proxies are required; the object set of each proxy is implicitly modeled by the known hit ratio. This solution, however, is an approximate one since the exact hit ratio of a proxy depends on several parameters including the actual content of the proxy, the demand, the existence of other proxies in the path to the clients and, thus, the use of known "typical" values for the hit ratio may introduce a significant error factor.

The algorithms proposed here allocate storage units that may contain any object from a set of distinct objects (thus this is a multi-commodity problem as opposed to the single commodity k-median) and use objective functions that are representative of the exact content of each node. Additionally, it is not assumed that a node can hold the entire set of available objects; in fact, this set need not contain objects from a single web-server, but it potentially includes object from different web-servers outside the hierarchy. As compared to works that study the object placement problem [28] where each proxy has a known capacity, the current approach adds *an additional degree of freedom* by performing the dimensioning of the proxies along with the object placement. Note that this may lead to a significant improvement in performance because even an optimal object placement policy will perform poorly if poor storage capacity allocation decisions have preceded (e.g., a large amount of storage has been allocated to proxies that receive only a few requests, whereas proxies that service a lot of requests have been allocated a limited storage capacity).

The input to the problem consists of the following: a set of N distinct unit-sized objects[1], $\mathcal{O}$; an available storage capacity budget of S storage units; a set of m clients, $\mathcal{J}$, each client j having a distinct request rate λ_j and a distinct object demand distribution $p_j : \mathcal{O} \to [0,1]$; a tree graph T with a node set of n nodes, $\mathcal{V}$, and a distance function $d_{j,v} : \mathcal{J} \times \mathcal{V} \to R^+$ associated with the jth leaf node and node v; this distance captures the cost paid when client j retrieves an object from node v. In the remainder of the work it is assumed that $N > S$, i.e., the total number of objects exceeds the available storage budget[2]. Each client is co-located with a leaf node and represents a local user population (with size proportional to λ_j). A client issues a request for an object and this request must be serviced by either an ancestor node that holds the requested object or by the origin server. In any case, a client always receives a given object from the same unique node. The storage capacity allocation problem amounts to identifying a set $\mathcal{A} \subseteq \mathbb{A}$ with no more than S elements (node-object pairs) (v,k), $v \in \mathcal{V}$, $k \in \mathcal{O}$; $\mathbb{A}$ is the set that contains all node-object pairs. $\mathcal{A}$ must be chosen so as to minimize the following expression of cost:

$$\min_{\mathcal{A}\subseteq\mathbb{A}:|\mathcal{A}|\leq S} \sum_{j\in\mathcal{J}} \lambda_j \sum_{k\in\mathcal{O}} p_j(k) \cdot d_{j,v}^{min} \quad \text{where} \quad d_{j,v}^{min} = \min\{d_{j,os}, d_{j,v}\} : v \in ancestors(j), (v,k) \in \mathcal{A} \tag{1}$$

where $d_{j,os}$ is the distance between the jth client (co-located with the jth leaf node) and the origin server, while $d_{j,v}$ is the distance between the jth leaf node and an ancestor node v. The function in (1) captures the mean cost per unit time. It is a non-linear cost function since it involves the non-linear operator $\min\{\}$ in the definition of $d_{j,v}^{min}$. Also, note this cost models only "read" operations from clients. Adding "write" (update) operations from content creators is possible but as stated in [29] the frequency of writes is negligible compared to the frequency of reads and, thus, it does not seriously affect the placement decisions.

The output of the storage capacity allocation problem prescribes which objects to place in each node so as to achieve a minimal cost (in terms of fetch distance) subject to a capacity constraint. This solution can be implemented directly in a real world content distribution system that performs replication of content. Notice that the exact specification of objects for a node also produces the storage capacity that must be allocated to this node. Thus, an alternative strategy is to disregard the exact object placement plan and just use the derived per-node capacity allocation in order to dimension the nodes of a hierarchical cache that operates under a dynamic caching/replacement algorithm (e.g., LRU, LFU and their variants). Recently there has been concern that current hierarchical caches are not appropriately dimensioned [20] (e.g., too much storage has been given to underutilized upper level caches). Thus, the produced results can be utilized by systems that employ replication as well as by those that employ caching. Replication and caching offer different advantages and are considered to be complementary technologies not rivals [29].

[1]The unit-size assumption is merely to simplify the presentation. The proposed heuristic algorithms can be transformed to handle non-unit sized objects.

[2]The case $N \leq S$ can be handled by the same algorithms. Only some minor technical changes are required, mostly in the proof of Proposition 1.

3.2 Integer linear programming formulation of an optimal solution

In this section the storage capacity allocation problem is modeled as an integer linear programming (ILP) problem. Let $X_{j,v}(k)$ denote a binary integer variable which is equal to one if client j gets object k from node v where v is an ancestor of client j (including the co-located jth leaf node, excluding the origin server), and zero otherwise. Also let $\delta_v(k)$ denote a binary integer variable which is equal to one if object k is placed at the ancestor node v, and zero otherwise. The two types of variables are related as follows:

$$\delta_v(k) = \begin{cases} 1 & \text{if} \quad \sum_{j \in leaves(v)} X_{j,v}(k) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Equation (2) expresses the obvious requirement that an object must be placed at a node if some clients are to access it from that node. The following ILP gives an optimal solution to the storage capacity allocation problem.

Maximize:

$$z = \sum_{j \in \mathcal{J}} \lambda_j \sum_{k \in \mathcal{O}} p_j(k) \cdot \sum_{v \in ancestors(j)} (d_{j,os} - d_{j,v}) \cdot X_{j,v}(k) \tag{3}$$

Subject to:

$$\sum_{v \in ancestors(j)} X_{j,v}(k) \leq 1 \qquad j \in \mathcal{J}, k \in \mathcal{O} \tag{4}$$

$$\sum_{j \in leaves(v)} X_{j,v}(k) \leq U \cdot \delta_v(k) \qquad v \in \mathcal{V}, k \in \mathcal{O}, U \geq |\mathcal{J}| \tag{5}$$

$$\sum_{v \in \mathcal{V}} \sum_{k \in \mathcal{O}} \delta_v(k) \leq S \tag{6}$$

$$X_{j,v}(k), \delta_v(k) \quad \text{binary decision variables} \quad v \in \mathcal{V}, j \in \mathcal{J}, k \in \mathcal{O}$$

The maximization objective operates on the cumulative gain that is incurred from the placement of the S storage units on the tree. This gain captures the difference in cost between accessing a cached copy of an object from some ancestor node v (excluding the origin server) and accessing the original object at the origin server (see the parenthesis in (3)). The maximization of (3) is equivalent to the minimization of (1). The chosen definition of $X_{j,v}(k)$ allows for the use of a linear (gain) function for the exact modeling of the non-linear (cost) function (1). Notice that only the $X_{j,v}(k)$'s contribute to the objective function and the $\delta_v(k)$'s do not.

Constraint (4) enforces the requirement that a client fetches a given object always from the same ancestor node. Constraint (6) guarantees that no more than S storage units will be allocated (exactly S in an optimal solution). The adoption of the decision variables $\delta_v(k)$ – in addition to $X_{j,v}(k)$ – is to facilitate the formulation of this constraint. Notice that using only $X_{j,v}(k)$ and requiring that the sum of all $X_{j,v}(k)$ be S could lead to the allocation of less than S storage units since the placement of a single copy of an object, which consumes exactly one storage unit, permits multiple clients to share the same copy, thus allowing for several variables $X_{j,v}(k)$ to assume the value one. The introduction of the auxiliary decision variables $\delta_v(k)$ allows for the correct formulation of the storage capacity constraint but also produces a non-linearity in the form of the conditional relationship (2) that relates the two types of variables. Constraint (5) guarantees that an object be placed at a node if some clients are to access it from there. U is an upper bound on the number of clients that may be fetching an object from the same node; the smallest sufficient value for U is m, the total number of clients. Constraint (5) combined with the objective function (3) enforce that the conditional relationship (2) between $X_{j,v}(k)$ and $\delta_v(k)$ holds under an optimal solution. This permits the formulation of a non-linear relationship using only linear functions. The validity of this linearization technique is established in the following proposition.

Proposition 1 *Let (X^*, δ^*) be an optimal solution to the ILP (3), (4), (6), (5). Then $\delta_v(k) = 0$ if and only if $\sum_{j \in leaves(v)} X_{j,v}(k) = 0$ for all $X_{j,v}(k)$ and $\delta_v(k)$ in the optimal solution (X^*, δ^*).*

Proof: If $\delta_v(k) = 0$ then constraint (5) guarantees that $\sum_{j \in leavesv} X_{j,v}(k) = 0$ since all $X_{j,v}(k)$ are non-negative numbers. For the proof of the inverse assume that for a given node v and object k the optimal solution (X^*, δ^*) has $\sum_{j \in leaves(v)} X_{j,v}(k) = 0$ and $\delta_v(k) = 1$. Now it is easy to see that by removing k (flipping $\delta_v(k)$ from 1 to 0) and placing instead one of the objects that reside only at the origin server (such an object k' will exist since it has been assumed that $N > S$) a new solution $(X^\dagger, \delta^\dagger)$ may be constructed that incurs a higher value of z. This holds true because flipping $\delta_v(k')$ from 0 to 1 allows all $X_{j,v}(k')$, $j \in leaves(v)$ to flip from 0 to 1 thus increasing the value of z. This contradicts the initial assumption that (X^*, δ^*) is optimal. The proof for the inverse can also be given for $N \leq S$. Instead of fetching an (unreplicated) object from the server, an object from an ancestor node (higher in the hierarchy) can be replicated again at v (lower) thus improving the solution. □

In the sequel, the above mentioned ILP will only be employed for the purpose of obtaining a bound on the performance of an optimal storage capacity allocation. Such a bound is derived by considering the LP-relaxation of the ILP (removing the requirement that the decision variables assume integer values in the solution) which can be derived rapidly by a linear programming solver.

3.3 Complexity of an optimal solution

The ILP formulation of Sect. 3.2 is generally NP-hard thus cannot be used for practical purposes. One of our recent findings is that the aforementioned storage capacity allocation problem can be modeled as a special type of problem that belongs to a family of multicommodity resource allocation problems that generalize the single-commodity k-median problem [26, 27]. We have shown that these generalized problems can be solved in polynomial time for the case of tree graphs [30] and have described a two-step algorithm that reaches an optimal solution to the multi-commodity resource allocation problem (including the one presented here) in $O(\max\{n^4 N, n^2 N^2\})$. The solution is sought by first solving a series of k-medians for the different objects using known dynamic programming techniques [31, 32] and then combining the solutions of the various k-medians in order to solve a packing problem that is a modified version of the 0/1 Knapsack problem [33], using again dynamic programming.

From a theoretical point of view, the $O(\max\{n^4 N, n^2 N^2\})$ result is rather attractive as it involves small powers of the input. In practice, however, such a result might prove too difficult to apply. The main complication owes to the N^2 term indicating a quadratic dependence of complexity on the number of distinct objects N. In real applications N may assume very large values (as it represents the number of distinct web pages or P2P files), typically much larger than the values assumed by n (number of nodes in the content distribution infrastructure). Thus, the common case will be a complexity of $O(n^2N^2)$ with n ranging from tens to hundreds and rarely few thousands (large CDNs of Akamai's size) and N in the order of multiple thousands to millions (depending on the exact application). Quadratic complexities on such large inputs are generally treated as too complicated for practical purposes [28] in systems that need to re-allocate storage frequently. Such solutions are less possible to apply. Additionally, the aforementioned optimal algorithm cannot jointly consider load balancing (Sect. 4.1) and request peering (Sect. 4.2). For these cases no exact polynomial algorithm is known.

For all these reasons, this work is primarily focused on the development of efficient approximate algorithms. We employ natural greedy heuristics to address the basic problem and its variations (considering load balancing and request peering). The developed algorithms are easy to implement, incur a complexity that is in the worst case linearly dependent on N, provide for a close approximation of the optimal, and lend themselves to incremental use under common circumstances involving time varying input (such need arising when considering dynamic adaptive systems).

3.4 The Greedy heuristic

The Greedy heuristic begins with an empty hierarchy and enters a loop placing one object in each iteration, thus, in exactly S iterations all the storage capacity has been allocated. Objects are placed in a greedy fashion according to the gain that is produced with each placement and past placement decisions are not subject to future placement decisions in subsequent iterations. The gain of an object at a certain node depends on the location of the node, the popularity of the object, the aggregate request rate for this object from all clients on the leaves of the subtree rooted at the selected node, and on prior iterations that have placed the same object lower in the subtree. In the first iteration the algorithm selects an node-object pair (v_1, k_1) that yields a maximal gain and places k_1 at v_1. Subsequent decisions place an object k in node v such that the $gain_v(k)$ is maximal among all (v, k) pairs that have not been selected yet; $gain_v(k)$ is given by:

$$gain_v(k) = \sum_{\substack{j \in leaves(v) \\ k \notin path(j,v)}} (d_{j,os} - d_{j,v}) \cdot p_j(k) \cdot \lambda_j \tag{7}$$

Greedy is presented in detail in Table 1. Lines 1-6 describe the initialization of the algorithm. For each node v the gain of placing object k in v is computed and these values are inserted in a max-heap [34] data structure $g(v, \cdot)$ (n max-heaps, one for its node v). In the S iterations of the algorithm the following two steps are executed: (1) (v^*, k^*), the node-object pair that produces the maximum gain among the node-object pairs that have not been placed, $\mathcal{P}$, is selected, removed from $\mathcal{P}$, and the max-heap $g(v^*, \cdot)$ is re-organized (lines 9-10); (2) for each ancestor u of v^* that does not hold k^* the (potential) gain incurred if k^* is selected for u at a later iteration is updated and the corresponding max-heap is re-organized (lines 11-14). The update of the potential gain $g(u, k^*)$ is necessary because the clients belonging to the subtree below v^* will not be fetching k^* from u but from v^* or its subtree, thus effectively reducing its previous potential gain at u. A max-heap needs to be re-organized when one of its elements changes value in order to be able to guarantee that the maximum value is always at the root of the heap.

The initial creation of the n max-heaps can be done in $O(nN)$ time (each max-heap containing N values). The first step of each iteration requires that the highest value in all n max-heaps be selected. Finding the largest value in a max-heap requires $o(1)$ time thus the largest value in all

```
for each $v \in \mathcal{V}$
   for each $k \in \mathcal{O}$
       $g(v,k) = gain_v(k)$
       insert $g(v,k)$ in max-heap $g(v,\cdot)$
   end for
end for
$i = 1, \mathcal{P} = \{(v,k) : v \in \mathcal{V}, k \in \mathcal{O}\}$
while $i \leq S$
   select $(v^*,k^*) \in \mathcal{P} : g(v^*,k^*) \geq g(v,k) \forall (v,k) \in \mathcal{P}$
   $\mathcal{P} = \mathcal{P} - \{(v^*,k^*)\}$, re-organize max-heap $g(v^*,\cdot)$
   for each ancestor $u$ of $v^*$ not caching $k^*$
       $g(u,k^*) = gain_u(k^*)$
       re-organize max-heap $g(u,\cdot)$
   end for
   $i = i + 1$
end while
```

Table 1: The Greedy algorithm.

n max-heaps can be identified in $O(n)$ by a simple linear search or in $O(\log n)$ if an additional max-heap is maintained, containing the highest value from each of the n max-heaps $g(v,\cdot)$ (the latter might be unnecessary since n is typically rather small). The second step in the worst case requires the update of $L-1$ ancestors that do not cache k^*. The potential value of k^* for these nodes must be reduced and the corresponding max-heap must be re-organized in order to maintain the heap property; this can be done in $O(\log N)$ for a heap with N objects. As a result the S iterations of the algorithm require $O(S \cdot (n + L \log N))$ time, which simplifies to $O(S \cdot n \log N)$ by noting that L is at most n. Thus the overall complexity of Greedy (initialization + iterations) is $O(\max\{nN, Sn \log N\})$ which is linear in either N or S.

A salient feature of Greedy is that it can be executed incrementally, i.e., if the available storage budget changes from S to S' (e.g., because more storage has become available) and the user access patterns have not changed significantly then no re-optimization from scratch is required; it suffices to continue Greedy from its last iteration and add (or remove) $|S' - S|$ objects. This can present a significant advantage when the algorithm must be executed frequently for different S.

3.5 The improved Greedy heuristic (iGreedy)

In the previous Greedy algorithm one can make the following simple observation. If an object is placed at all children of a node u it is meaningless to also store it in u since no request will reach it there. This situation leads to the "waste" of storage units in "stale" objects. The Greedy algorithm often introduces stale objects as a result of its greedy mode of operation; an object is at some point placed at the father u while at that time not all children store it but with subsequent iterations it is also placed at all children thus rendering stale the copy at the father. This situation is not an occasional one but it is repeated quite frequently, resulting in wasting a substantial amount of the storage budget. The situation may be resolved by executing an additional check when placing

```
10.1:  allpeers=1
10.2:  for each v ∈ peers(v*)
10.3:     if k* not cached in v
10.4:        allpeers = 0
10.5:        break
10.6:     end if
10.7:  end for
10.8:  if k* cached in u=father(v*) AND allpeers==1
10.9:     remove k* from u and set i = i − 1
10.10: end if
```

Table 2: Additional step of the iGreedy algorithm. It is executed between lines 10 and 11 of the basic Greedy algorithm.

an object k^* at a node v^*. The improved algorithm checks all peer nodes of v^* (at the same level, belonging to the same father) and if it finds that all store k^* then it also check whether their father u also stores it. In such a case it removes it from u freeing one storage unit. The resulting algorithm is called improved Greedy (iGreedy) and outperforms the basic Greedy algorithm. The additional step of iGreedy is given in Table 2 and is executed between lines 10 and 11 of the basic Greedy algorithm.

iGreedy performs slightly more processing as compared to Greedy due to the following two additional actions: (1) in each iteration a maximum of Q peers need to be examined against k^*, Q denoting the maximum node degree of the tree; (2) each eviction of a stale object increases the number of iterations by one by freeing one storage unit which will have to be allocated in a subsequent iteration. Searching the Q peers does not affect the asymptotic per-iteration complexity of Greedy which is $O(n \log N)$. The increase in the number of iteration has a somewhat larger impact on the required processing. The following proposition establishes an exact upper bound on the number of iterations performed by iGreedy.

Proposition 2 *The maximum number of iterations performed by iGreedy cannot exceed* $T(S) = 2 \cdot S - 1$.

Proof: Note that in the worst case an additional iteration may be introduced after at least $q+1$ new objects have been placed, q denoting the minimum node degree in the hierarchy. This corresponds to the case that an object is placed at the node with the minimal outdegree and at all its children, thus, freeing the stale object at the father which in turn increases the number of iterations by one. Let $T(S)$ denote the maximum number of iterations that may be performed when iGreedy starts with S available storage units. Using the previous observation it is easy to see that: $T(S) = q+1+T(S-(q+1)+1) = q+1+T(S-q)$. In the worst case that $q = 1$ the recursive relationship becomes: $T(S) = 2 + T(S-1)$, with $T(1) = 1$, which leads to $T(S) = 2 \cdot S - 1$. $\square$

Thus in the worst case iGreedy will perform $2 \cdot S - 1$ iterations, with each iteration incurring the same complexity as with the basic Greedy. This means that the asymptotic complexity of iGreedy is identical to that of Greedy.

3.6 Numerical results under iGreedy

In this section the presented numerical results attempt to accomplish the following: (1) demonstrate the effectiveness of iGreedy in approximating the optimal performance; (2) present possible applications of the developed algorithms. When not stated otherwise, the clients are assumed to be sharing a common Zipf-like demand distribution p_j over $\mathcal{O}$ with a typical skewness parameter $a = 0.9$ and equal request rates $\lambda_j = 1, \forall j \in \mathcal{J}$. A Zipf-like distribution is a power-law dictating that the ith most popular object is requested with a probability C/i^a, where $C = (\sum_{j=1}^{N} \frac{1}{j^a})^{-1}$. The skewness parameter a captures the degree of concentration of requests; values approaching 1 mean that few distinct objects receive the vast majority of requests, while small values indicate progressively uniform popularity. The Zipf-like distribution is generally recognized as a good model for characterizing the popularity of various types of measured workloads, such as web objects [35] and multimedia clips [13]. The popularity of P2P [12] and CDN content has also been shown to be quite skewed towards the most popular documents, thus approaching a Zipf-like behavior. Recently, evidence of Zipf-like behavior has been observed in the distribution of Gnutella queries [36].

As far as the topology of the experiments is concerned, regular Q-ary trees are used in all examples. Regular Q-ary trees are commonly used for the derivation of numerical results for algorithms operating on trees [37, 10]; it has also be seen that numerical results from regular tree topologies are in good accordance with experimental results from actual internet tree topologies [38]. The entire set of parameters (demand and topology) for each experiment is indicated in the title of the corresponding graph. For the purpose of the numerical study we let the distance function $d_{j,v}$ capture the number of hops between client j (co-located with the jth leaf thus $d_{j,j} = 0$) and node v. The distance of the origin server is $d_{j,os} = L$ for an L level hierarchy. Notice that the hop count metric is merely used to simplify the presentation of the numerical results. It is by no means necessary in order to apply the algorithms (which allow for arbitrary weights to be associated with each link).

3.6.1 Quality of the approximation

Figure 2 shows the average cost per request for Greedy and iGreedy (expressed in number of hops to reach an object). The performance of the heuristic algorithms is plotted against the bound of the corresponding optimal performance obtained from the LP-relaxation of the ILP of Sect.3.2. The x-axis indicates the number of available storage units in the hierarchy (S) with each storage unit being able to host a single object (e.g., an mp3 file or a video file). From the graph it may be seen that iGreedy is no more than 2% away from the optimal while Greedy may deviate as much as 25% in the presented results. The performance gap between the two owes to the waste of a significant amount of storage in stale objects under Greedy.

The asymptotic performance of the best performing iGreedy – although not rigorously proved here – is claimed to be at a constant distance away from the optimal under typical workloads as S and N approach infinity. This can be intuitively argued by noting that the value of objects that belong to the tail of the popularity distribution tends to be insignificant and, thus, from a point onwards adding more capacity does not reduce the cost. This in effect means that suboptimal placement decisions involving invaluable objects – which are cached only when there is abundant storage capacity – cannot increase the performance gap created by the suboptimal decisions in the placement of the initial most valuable objects.

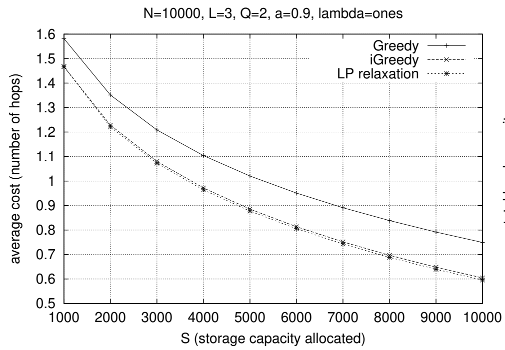

Figure 2: The average cost of Greedy, iGreedy and LP-relaxation.

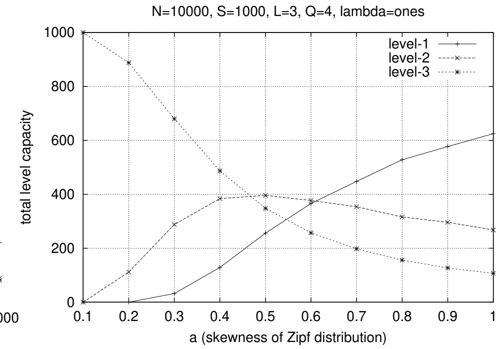

Figure 3: The effect of skewness of popularity on the per-level allocation of storage under iGreedy.

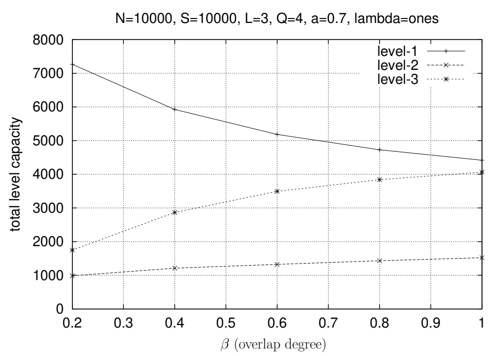

Figure 4: The effect of non-homogeneous demand on the per-level allocation of storage under iGreedy.

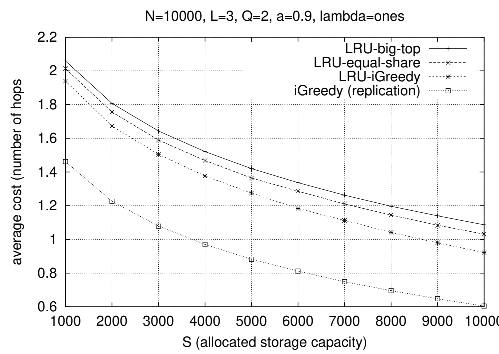

Figure 5: Simulated performance of LRU under iGreedy, equal-share and big-top storage allocation and equal request rates $\lambda_j = 1$.

3.6.2 The effects of skewness and non-homogeneous demand

The following two figures focus on the vertical allocation of storage under iGreedy, attempting to provide new insights into the effect of user access patterns in the dimensioning of hierarchical distribution systems. Figure 3 shows the effect of the skewness of the demand distribution on the per-level allocation of storage. Highly skewed distributions (the skewness parameter a approaching 1) lead to the allocation of more storage to the lower levels while less skewed distributions lead to the allocation of more storage to the higher levels. This effect is explained as follows. Under a highly skewed distribution a small number of popular objects attracts the majority of requests and, thus, these objects are intensively replicated at the lower levels, leading to the allocation of most of the storage to the lower levels. When the distribution tends to be "flat" it is better to limit the number of replicas per object and instead increase the number of distinct objects that can be replicated. This is achieved by sharing objects, i.e., by placing them higher in the hierarchy that leads to the allocation of more storage to the higher levels. An equal request rate ($\lambda_j = 1, \forall j$) for all clients leads to an equal allocation of storage in each node of the same level. This horizontal symmetry is disturbed when unequal client request rates are employed (such results not shown at this point).

Figure 4 illustrates the effect of the degree of homogeneity in the access patterns of different clients. Two clients are non-homogeneous if they employ different demand distributions. In the presented results each client j references N objects; βN objects are common to all clients while the remaining $(1-\beta)N$ are only referenced by client j. A Zipf-like distribution is created for each client by randomly choosing an object from its reference set and assigning it the next higher value from a Zipf-like distribution and then repeating the same action until all objects have been assigned probabilities. The parameter β will be referred to as the *overlap degree*; values of β approaching 1 mean that most objects are common to all clients (although each client may request a common object with a potentially different probability) while small values of β mean that each client references a potentially different set of objects. Figure 4 shows that the upper levels of a hierarchical system are assigned more storage when there is a substantial amount of overlap in client reference patterns. Otherwise most of the storage goes to the lower levels. This behavior is explained as follows. Storage is effectively utilized at the upper levels when each placed object receives an aggregate request stream from several clients. Such an aggregation may only exist when a substantial amount of objects are common to all clients; otherwise it is better to allocate all the storage to the lower levels – thus sacrificing the (ineffective) aggregation effect – and instead reduce the distance between clients and objects.

3.6.3 Using iGreedy with caching/replacement

The aforementioned results presented examples of how iGreedy would be useful in dimensioning a distribution system that operates using replication of content. For such systems iGreedy prescribes the amount of storage to be allocated to each node as well as the exact set of objects to be replicated. There is a second interesting use of iGreedy however. As discussed in Sect. 3.1 one can choose to retain only the node dimensioning from iGreedy, disregarding the object placement plan, and use it to dimension a system that operates under request driven caching/replacement (e.g., a hierarchical cache). Indeed, most caching/replacement strategies try to approximate the result of an optimal replication plan, but achieve that on-line, without a priori knowledge of demand patterns. Thus, using an optimized node dimensioning that originates from replication can also benefit the online caching/replacement systems.

It is interesting to note that there have been some recent works that question the appropriateness of the dimensioning of current operational hierarchical caches (e.g., noting that higher level caches have been unnecessarily overdimensioned [20]). Our results tend to agree with these concerns, indicating that an optimized storage allocation produces a gain as compared to two empirical storage allocations plans: (1) equal-share, where each node gets an equal share of storage, i.e., approximately $\lceil S/n \rceil$ units; (2) big-top, where nodes tend to get bigger at the higher levels (as often done in practice, and questioned in [20]). Under big-top, the S units are allocated as follows: each leaf node takes Y units, each second level node takes $2 \cdot Y$ units and, in general, each l level node takes $l \cdot Y$ units. These empirical rules might sound over-simplistic but it seems that they have been employed in practice (the aforementioned concern about overdimensioning higher level nodes points to that). In any case, we are not aware of any other storage allocation method that can fit the described problem statement thus be used for comparison.[3]

To examine the potential gain for caching/replacement systems it will be assumed that they operate under the least recently used (LRU) replacement algorithm. Figure 5 demonstrates the

[3] The work of Kelly and Reeves [9] considers tradeoffs between communication costs and storage ownership costs thus differing from the current setting where the amount of available storage is fixed. Also [9] assumes that all clients generate identical request streams (homogeneous using our terminology).

λ_1	λ_2	λ_3	λ_4	LRU_iGreedy avg. hit dist.	LRU_equal-share avg. hit dist.	LRU_big-top avg. hit dist.
1	1	2	7	0.63	1.06	1.11
9	1	6	1	0.63	1.07	1.10
4	1	8	6	0.81	1.05	1.10
4	2	3	3	0.88	1.04	1.11

Table 3: Simulated performance of LRU-iGreedy, LRU-equal-share and LRU-big-top under unequal client request rates λ_j ($N = 10000$, $S = 10000$, $L = 3$, $Q = 2$, $a = 0.9$).

server name	Urbana-Champaign
type	NLANR root
trace date	19/02/2003 - 24 hours
# requests	815194
# distinct objects	279375

Table 4: Properties of the NLANR trace used in the simulations of Sect. 3.6.4.

performance of LRU in a hierarchical cache that has been dimensioned under iGreedy and the two empirical rules. The performance of static replication under iGreedy is also simulated and plotted. Unlike all previous analytical results that refer to object replication, the results of Fig. 5, refer to request-driven caching using LRU. They are produced using a simulation model with synthetically generated i.i.d. requests following a Zipf-like distribution (see caption and title of Fig. 5 for parameters). The results show that when LRU operates on a hierarchy whose storage allocation has been optimized using iGreedy, it achieves a 10% reduction of cost over equal-share and a 15% reduction over big-top. These percentages become much larger under unequal client request rates λ_j. Table 3 presents some examples with λ_j's taken from a uniform distribution in $[1, 10]$. A reduction as large as 40% over equal-share, and 42% over big-top, can be achieved by the allocation under iGreedy. The performance gap widens with the degree of asymmetry in request rates. In effect, this performance gap may grow arbitrarily under various demand distributions and request rates (e.g., when there is small overlap in demand distributions and one of the clients has a dominantly larger request rate).

Comparing the performance of LRU (irrespectively of capacity allocation method) to that under static replication with iGreedy shows that replication – which has a priori knowledge of object popularities – achieves a significantly better performance than the online algorithms that, however, do not need object popularities and can adapt automatically to changes of demand patterns.

3.6.4 A trace-driven simulation study

One issue that has not been discussed so far is that of the temporal correlation that usually characterizes actual measured workloads. The use of iGreedy under replication leads to a system that is *immune to temporal correlations*. To see this, consider any request sequence and permute it arbitrarily. The hit ratio (or average hit distance) will be unaffected since a replication strategy does not permit nodes to change their content in response to the received input. Thus, the quality of the replication provided by iGreedy depends only on the quality of the estimation of the request frequencies of the individual objects, and is unaffected by temporal correlation.

Temporal correlation becomes relevant to some extent when iGreedy is limited to storage dimensioning and the network operates under caching/replacement. As the dimensioning procedure is immune to temporal characteristics, it is only the replacement algorithm that is affected in the usual manner. In the case of LRU, a positively correlated request stream will lead to improved performance as compared to a request stream composed of independent requests. Thus, the quality of the dimensioning – and the gain as compared to an empirical rule – remains, and it is only a matter of the replacement algorithm to best handle the correlation and produce a further gain.

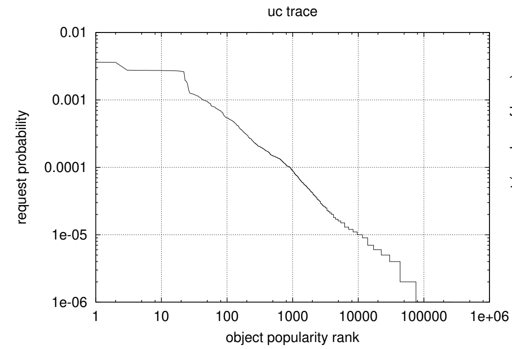

Figure 6: Popularity profile (request probability against object rank) for the trace of Table 4.

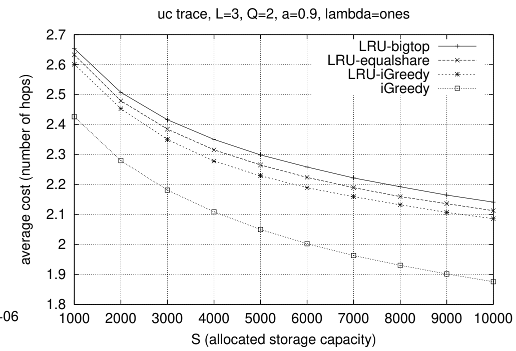

Figure 7: Average hit distance under various storage allocation schemes for the trace of Table 4.

In order to substantiate the aforementioned observations we have conducted a simulation experiment evaluating the performance of LRU-iGreedy, LRU-equal-share, LRU-big-top and iGreedy (with replication) under a real workload that captures some of the characteristics (notably a degree of temporal correlation) not present in a synthetically generated stream of i.i.d. requests. Due to the lack of publicly available P2P workload traces, we employed a trace from a NLANR root proxy server (Urbana-Champaign, abbreviated uc); the properties of the trace are summarized in Table 4. The motivation for using this trace as an approximation of an actual P2P trace is: (1) the trace captures properties not present in the synthetical workloads; (2) the popularity of objects in this trace approximates satisfactorily the popularity of P2P objects that – according to some very recent measurements studies – shows some deviation from the Zipf-like model (thus, our numerical evaluation is enriched as wells as captures this new artifact).

Figure 6 depicts a log-log plot of the request probability against the object rank (according to popularity) for the objects accessed in the trace. A visual inspection suffices to show that this distribution partially deviates from the Zipf-like one, that appears as a straight line in log-log scale. The divergence is towards the most popular objects (initial part), that appear to be more uniformly requested here (observe the initial plateau on the plot) as compared to a Zipf-like distribution. A very recent measurement study (Oct. 2003) from a research group in the University of Washington [39] (the same group that has produced some of the previously cited measurement studies [12, 13]) has reported that such a plateau appears consistently in all their measured P2P traces. It has been attributed to the fact that unlike web objects, P2P objects are accessed *at most once* by each client (a web user constantly downloads the same popular web pages (e.g., Google) but rarely downloads more than once the same mp3 file). In the case of the NLANR trace the plateau is due to a different reason, probably owing to the fact that the corresponding proxy server is high in the NLANR hierarchy (root server), thus requests reaching it have been filtered at lower levels. This means that many requests for popular objects have been serviced at the lower levels, thus making the request distribution at the root more uniform in its initial part. We take advantage of this coincidental fact in order to approximate a P2P trace.

Figure 7 shows the average hit distance in a hierarchy that operates under the optimized storage capacity allocation and the empirical rules. Both LRU replacement (first three curves), as well as replication (last curve), are depicted. The results are identical to the ones depicted in Fig. 5 that have been produced using synthetical workload (the discussion on the achieved gain applying here also). They suggest once more – this time using a real workload – that an optimized storage capacity allocation can result in a reduced average hit distance under both LRU replacement and replication. We have obtained similar results under several other traces (from different NLANR servers and/or time periods) and different topologies, which we do not report here due to length considerations.

3.7 Discussion and implementation

As discussed in the introduction, two possible applications of the presented storage allocation algorithms are: (1) to re-organize current hierarchical caches; (2) to drive the operation of future content distribution systems that dynamically lease memory. The case of (1) is an offline optimization of a resource allocation problem. The required input (demand distributions and rates) can be extracted from proxy log files. An interesting question is whether a useful solution can be derived using historical data. Krishnan et al. [3] have answered this question for the related "cache location problem". They showed that their solutions were fairly stable across time and estimation deficiencies, despite that their optimization was carried-out based on input from a past "snapshot" of the workload (including some estimation error). The stability of the derived solution owes much to the fact that user access patterns seem to be fairly stable. In fact, a recently published measurement study [40] reported that the top 10% most popular documents on one day make up for about 80% of all requests for at least the following week. This allows for the possibility of using recent historical data as input in making replication decisions.

The case of (2) is somewhat more challenging from the perspective of implementation. The main difficulty is due to the need for online estimation of demand distributions (the aforementioned p_j's). There are various methods for doing that efficiently, i.e., without keeping track of all requested objects but only a fraction of the most popular ones [41]. However, owing to the aforementioned stability of demand distributions, it might not be necessary to keep re-estimating them frequently but only occasionally. Should demand distributions change slowly, then it will be the request rates (λ_j's) that will trigger changes in storage capacity allocation plans. Notice that under stable demand distributions as the ones reported in [40], changes in the request rates will require re-optimization, since the allocation of storage is sensitive to the request rates. Such changes can occur as users activate and deactivate at the various access points. This also allows for accommodating the case of mobile users that show up and access the resources through the leaf access points affecting the request rate. From the perspective of implementation complexity, request rates may be obtained easily by maintaining a counter in each node; a task much easier as compared to the frequent estimation of request distributions.

The fact that iGreedy's complexity is only linearly dependent on the input of the problem facilitates the frequent re-optimization, necessary for systems that re-organize to cope with changes in request intensity (possible re-optimization intervals ranging from few minutes to some hours). Since the optimization algorithm is centralized, it should be executed at some node with administrative duties that overlooks the operation of the CDN. For that purpose, estimates of the demand distributions should be communicated to the administrative node by all leaf nodes, which are the access points to the hierarchical distribution infrastructure.

In order to reduce the bandwidth overhead from the reporting of the demand distributions compact representations should be used to provide succinct "summaries" of the actual demand at each leaf. Leafs can use simple techniques like sending a numerical value encoded with a few bytes

(2-8) representing an estimate of the request probability for each one of the N distinct objects. Alternatively, more advanced techniques could be employed to reduce the bandwidth overhead of transmitting the relevant information. For example, Bloom filters [42], such as those used in "Summary Cache" [2], could be employed to succinctly represent the set of distinct objects requested at a given leaf and this information be accompanied by approximate representation of the corresponding demand distribution (e.g., by exploiting its spectral representation).

Additionally, due to the aforementioned stability of demand distributions, it is possible to report back the required information relatively infrequently (retransmissions every one or several days would suffice), thus further limiting the bandwidth overhead. On the other hand, request rates may be reported as frequently as needed since their overhead is negligible (only a single number from each leaf must be sent). The frequent reporting of request rates (intensity) is however crucial in adapting to changes of demand due to activation/deactivation of clients and mobile users.

Summing it up, it is possible to re-organize effectively the allocation of storage with the aforementioned architecture, so as to accommodate possible changes in the request intensity at the access points of the infrastructure, owing to the volatility of users.

4 Variations for load balancing and request peering

In the following two sections the iGreedy heuristic is modified to cater to load balancing and request peering. An additional section presents some relevant numerical results. Load balancing is a commonly sought ability in distributed systems and so is the case here where the system should avoid overutilizing some nodes while underutilizing others. Request peering is employed by hierarchical systems with positive effects in the user-perceived performance.

4.1 Load balancing: Optimal and the iGreedyLB heuristic

Let W_v denote the maximum number of requests per unit time that may be serviced by node v – this metric captures the maximum *load* that can be assigned to a node and will be used for *load balancing* and *congestion avoidance* at the caching nodes. The ILP model of Sect. 3.2 can be modified to respect the maximum load of each node with the addition of the following constraint.

$$\sum_{j \in leaves(v)} \sum_{k \in \mathcal{O}} \lambda_j \cdot p_j(k) \cdot X_{j,v}(k) \leq W_v \qquad v \in \mathcal{V} \tag{8}$$

Similarly, the iGreedy algorithm of Sect. 3.5 can be augmented to handle load constraints with the addition of few extra steps. In brief, the algorithm maintains a load counter for each node and selects node-objects pairs in decreasing order with respect to gain. Contrary to the basic iGreedy, the load balancing version of the algorithm, iGreedyLB, considers only feasible node-object pairs when placing objects, i.e., it selects objects that do not violate the load limit of their node if they are selected for placement. The new algorithm is based on iGreedy and Greedy. Table 5 gives the additional lines of pseudocode for iGreedyLB. Lines 5.1, 9.1 and 13.1-13.4 are new and should be added to the pseudocode of Table 1 after lines 5, 9 and 13, while lines 7 and 10 are substitutes for the corresponding lines of Table 1. Line 5.1 initializes to zero the load counter for each node. Line 7 creates the set $\mathcal{P}$ which comprises the node-object pairs which are eligible for placement; it requires that the new load imposed by a node-object pair does not violate the maximum load of

5.1: $load_v = 0 \quad \forall v \in \mathcal{V}$
7: $i = 1, \mathcal{P} = \{(v, k) : load_v + newload(v, k) \le W_v, v \in \mathcal{V}, k \in \mathcal{O}\}$
9.1: $load_{v*} = load_{v^*} + newload(v^*, k^*)$
10: $\mathcal{P} = \mathcal{P} - \{(v^*, k^*)\} - \{(v^*, k) \in \mathcal{P} : load_{v^*} + newload(v^*, k) > W_{v^*}\}$, re-organize $g(v^*, \cdot)$
13.1: find $u \in \mathcal{V}$ the closest ancestor of v^* that caches k^*
13.2: $load_u = load_u - newload(v^*, k^*)$
13.3: $\mathcal{P} = \mathcal{P} + \{(u, k) : (u, k) \notin \mathcal{P}, k$ not cached in $u, load_u + newload(u, k) \le W_u\}$
13.4: re-organize max-heap $g(u, \cdot)$

Table 5: Additional steps for the iGreedyLB algorithm.

the node. The function $newload(v, k)$ is defined as follows: $newload(v, k) = \sum_{\substack{j \in leaves(v) \\ k \notin path(j,v)}} \lambda_j \cdot p_j(k)$. Line 9.1 updates the load of node v^* once object k^* has been chosen for placement at v^*. Line 10 updates $\mathcal{P}$ by removing (v^*, k^*) and all the node-object pairs (v^*, k) that become infeasible due to the increment in the load of v^*; this necessitates the update of max-heap $g(v^*, \cdot)$. Line 13.1 identifies u, the closest ancestor of v^* that caches k^* (excluding the origin server). The load of this node is reduced because the leaves of v^* do not any longer fetch k^* from u but from v^* (Line 13.2). Finally, due to the reduction of $load_u$ some objects that were not cached at u and were previously infeasible due to the load constraint now become feasible and thus are added to $\mathcal{P}$ (Line 13.3) and the corresponding max-heap is re-organized (Line 13.4).

Contrary to load-unaware practices, the load balancing ILP and the iGreedyLB heuristic not only allocate the storage capacity more evenly among the nodes of the hierarchy, but also place some popular objects at higher level nodes only, as opposed to the common intuition that popular objects must be placed at the lower levels of the hierarchy. This must be done in order to preserve the load constraints and to distributed the load more evenly in the hierarchy. Such a tactic results in a worse performance in terms of the average distance or bandwidth consumption (such is the case with the hop-count distance metric that is considered in the numerical results of Sect. 3.6). However, this may not be the case for delay metrics. Placing memory and some popular objects higher in the hierarchy admittedly increases the propagation delay of accessing them but this increment is expected to be small compared to the delay reduction that is incurred by accessing non-congested proxy servers. The end-system delay part dominates the propagation delay part in the overall delay budgets of congested end-systems such as those that arise when load balancing is not considered.

4.2 Modeling request peering: Optimal and the iGreedyP heuristic

Some replication and caching [21] schemes permit peer nodes – i.e., nodes of the same level that have the same father – to cooperate. A node that receives a request from a downward node, and cannot service it locally from its own cache, forwards it to its upstream node as well as to its peer nodes at the same level. In some occasions this may lead to a significant reduction in delay as a peer node, which is usually "closer" than upstream nodes, might quickly return the requested object. In general peering is meaningful when fast direct links exist between peer nodes and the delay on those links is smaller than the delay on the links that lead to higher level nodes. In such an environment, request peering leads to a higher utilization of each cached object, as this object services a larger population, which in turn reduces the number of replicas that are necessary for this object. Reducing the number of replicas of each object allows for the caching of a larger number of *distinct* objects (a larger initial part of the tail of the object popularity distribution fits in the hierarchy) thus resulting in better performance in terms of cost (bandwidth/delay). This section describes how to integrate request peering in the study of the storage capacity allocation problem.

The ILP formulation of Sect. 3.2 can be augmented to handle request peering. A larger number of variables $X_{j,v}(k)$ is required because – due to peering – client j may receive object k from a larger set of nodes including not only the ancestors v, but their peers too. Also, the objective function (3) must change to:

$$z = \sum_{j\in\mathcal{J}} \lambda_j \sum_{k\in\mathcal{O}} p_j(k) \sum_{v\in ancestors(j)} \sum_{u\in peers(v)} (d_{j,os} - d_{j,v} - peercost \cdot I_{\{u\neq v\}}) X_{j,u}(k) \tag{9}$$

$peers(v)$ denotes the set of nodes that are peers to v, including v itself. $peercost$ is a fixed[4] cost that is paid when transmitting an object between any two peers; this cost is typically smaller than the cost of accessing an ancestor at a higher level. I is an indicator function; it returns 1 when the expression in brackets becomes true and 0 otherwise. Also constraint (5) must change to:

$$\sum_{u\in peers(v)} \sum_{j\in leaves(u)} X_{j,v}(k) \leq U \cdot \delta_v(k) \qquad v \in \mathcal{V}, k \in \mathcal{O} \tag{10}$$

Finally, constraint (4) must be re-written as:

$$\sum_{v\in ancestors(j)} \sum_{u\in peers(v)} X_{j,u}(k) \leq 1 \qquad j \in \mathcal{J}, k \in \mathcal{O} \tag{11}$$

Similarly the iGreedy algorithm of Sect. 3.5 can be converted to handle request peering; the resulting algorithm will be referred to as iGreedyP. The introduction of request peering has two effects on the basic iGreedy algorithm. First, the gain function (7) must be updated as follows:

$$gain_v(k) = \begin{cases} \displaystyle\sum_{u\in peers(v)} \sum_{\substack{j\in leaves(u)\\ k\notin peers(w)\\ w\in path(j,u)}} (d_{os,j} - d_{j,u} - peercost) \cdot p_j(k) \cdot \lambda_j & \text{, if } k \notin u, \forall u \in peers(v) \\ \displaystyle\sum_{\substack{j\in leaves(v)\\ k\notin peers(w)\\ w\in path(j,v)}} peercost \cdot p_j(k) \cdot \lambda_j & \text{, otherwise} \end{cases} \tag{12}$$

The first expression in (12) corresponds to the case that no peer of v caches k thus placing k at v will attract requests from the descendants of v as well as from the descendants, j, of all peer nodes u, that do not fetch k from any node or peer in the path from j to u. The second line corresponds to the case that k is already cached at some peer(s) u thus the gain of also caching it at v will be equal to the extra distance *peercost* that will be "spared" (not have to be crossed) if an additional copy of k is placed at v (otherwise the descendants of v would have to fetch k from a peer u that caches it, incurring an additional cost *peercost*). The second modification refers to the removal of "stale" objects from higher level nodes. In iGreedyP it suffices to have one copy of k at any child u of a node v to remove k from v if already placed there. This is motivated by the fact that due to request peering a single copy of k at any peer can service all other peers too, thus, immediately rendering stale the copy of k at the father (without requiring that all children cache k as is the case with iGreedy).

Table 6 contains the pseudocode for iGreedyP. Lines 1-7 initialize the algorithm by computing the initial gains for each node-object pair and storing them in n max-heaps, one for each node. Each

[4]The communication cost between peers could depend on the exact two peers that communicate but since such a cost is typically much smaller than the cost of going to a higher level, it is modeled by a constant in order to simplify the presentation.

1: **for each** $v \in \mathcal{V}$
2:   **for each** $k \in \mathcal{O}$
3:     $g(v,k) = gain_v(k)$
4:     insert $g(v,k)$ in max-heap $g(v,\cdot)$
5:   **end for**
6: **end for**
7: $i = 1, \mathcal{P} = \{(v,k) : v \in \mathcal{V}, k \in \mathcal{O}\}$
8: **while** $i \leq S$
9:   select $(v^*,k^*) \in \mathcal{P} : g(v^*,k^*) \geq g(v,k) \forall (v,k) \in \mathcal{P}$
10:   $\mathcal{P} = \mathcal{P} - \{(v^*,k^*)\}$, re-organize max-heap $g(v^*,\cdot)$
11:   **for each** peer u of v^* not caching k^*
12:     $g(u,k^*) = gain_u(k^*)$
13:     re-organize max-heap $g(u,\cdot)$
14:   **end for**
15:   **for each** $w : w \in peers(u), u \in ancestors(v^*)$
        w not caching k^*
16:     $g(w,k^*) = gain_w(k^*)$
17:     re-organize max-heap $g(w,\cdot)$
18:   **end for**
19:   **if** k^* cached at $father(v^*)$
20:     remove k^* from $father(v^*)$
21:     $i = i - 1$
22:   **end if**
23:   $i = i + 1$
24: **end while**

Table 6: The iGreedyP algorithm.

iteration of the algorithm executes the code in lines 9-23. First, the node-object pair that returns the maximum gain in $\mathcal{P}$, is selected, and then removed from $\mathcal{P}$ (lines 9-10). Then the potential gain that would be incurred with the caching of the selected object k^* is updated for all the nodes that are affected from the selection of (v^*, k^*); these nodes are the peers of v^* (lines 11-14) as well as all its ancestors and their peers (lines 15-18). Finally, if the father of v^* includes k^* then it is removed from there, thus freeing one storage unit (this increases the number of iterations by one, lines 19-23).

4.3 Numerical results under iGreedyLB and iGreedyP

This section presents a number of numerical results pertaining to the variations of iGreedy for load balancing and request peering. To examine the efficiency of the heuristic algorithms, their performance is plotted against the bound of the corresponding optimal performance, thus: iGreedyLB is compared against the LP-relaxation of the load-balancing ILP of Sect.4.1, and iGreedyP against the LP-relaxation of the ILP with peering of Sect. 4.2.

Figure 8 shows the effect of different degrees of load balancing on iGreedyLB and on the optimal solution with load balancing. In all cases iGreedyLB approximates very closely the optimal. A

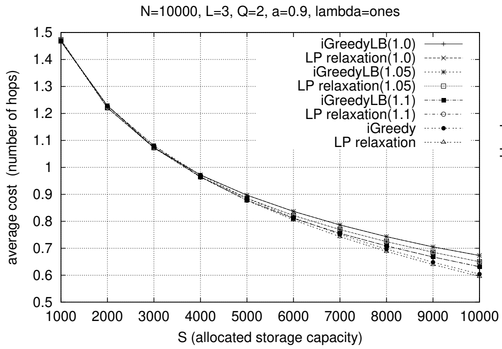

Figure 8: The effect of load balancing on the average cost under iGreedyLB and the LP-relaxation of the ILP with load balancing.

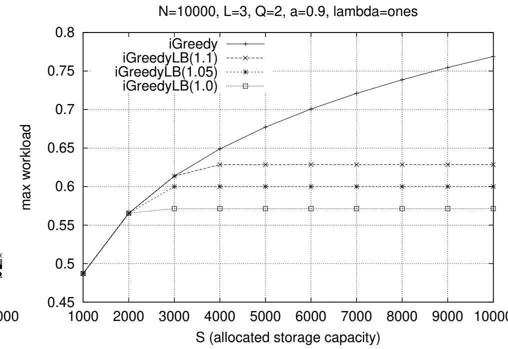

Figure 9: The effect of load balancing on the maximum load that may be imposed on any node in the hierarchy under iGreedyLB.

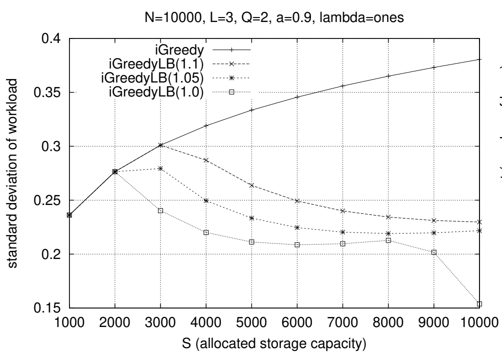

Figure 10: The effect of load balancing on the standard deviation of load in the nodes of the hierarchy under iGreedyLB.

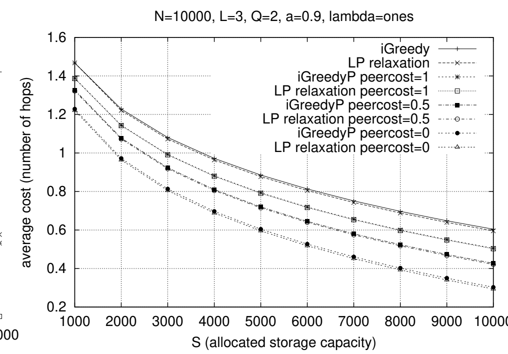

Figure 11: The effect of peering on the average cost incurred under iGreedyP and LP-relaxation of the ILP with peering, for different peering costs (indicated in parentheses in the graph).

maximum load parameter (*maxload*) is used to capture the maximum amount of load a node may service. Setting $maxload = 1.0$ means that a node is able to service up to a fraction $\sum_{j \in J} \lambda_j / n$ of the total load. This is the smallest fraction that, if used by all nodes in the selected setting, will avoid request blocking due to load constraints. Larger values of *maxload* mean that a node may service a fraction $maxload \cdot \sum_{j \in J} \lambda_j / n$ of the total load, this fraction being more than an "equal share" of load (in iGreedy a single node can potentially service the entire load). Figure 8 shows that the plain iGreedy performs better in terms of average cost than the various load-constrained iGreedyLB(*maxload*). Increasing *maxload* helps iGreedyLB(*maxload*) to approximate the best performance without load balancing. The smallest value, $maxload = 1.0$, leads to the best load balancing situation where all nodes get an approximately equal amount of load but also results is an increased cost as the optimal allocation of storage deviates (due to load constraints) from the optimal unconstrained one. The maximum average cost gap between the load-unaware iGreedy and the most load-constrained iGreedyLB(1.0) is 10%.

The following two figures depict the positive effects of load balancing. Figure 9 shows that load balancing is effective in suppressing the maximum load that may be imposed on any node. For different values of *maxload*, the maximum load of a node is eventually stabilized to the highest allowed value under iGreedyLB, while it increases continuously under iGreedy as more storage is added. Figure 10 shows the standard deviation of load computed over all the nodes in the hierarchy, for different degrees of load balancing. Load balancing leads to smaller values of standard deviation, thus most nodes have a load that is close to the mean load in the hierarchy. This means that the load is evenly distributed and nodes are utilized better as opposed to the unconstrained case where some nodes are congested while others are severely underutilized, especially those at higher levels. Notice that by adding more storage, the most load constrained iGreedyLB(1.0) achieves almost perfect load balancing (the total load is evenly shared among all nodes).

The presented load balancing techniques may help in addressing the well known underutilization problem in the higher levels of hierarchical caches due to the "filtering" of most requests at the lower levels [20]. Note that the 10% increase in the average cost (number of hops) between iGreedy and iGreedyLB(1.0) results in a 25% reduction of maximum load and a 60% reduction in the standard deviation of load in the hierarchy. These results may lead to a substantial reduction in the average delay by noting that whereas the propagation delay increases linearly with the distance (number of hops) the end system delay (I/O delay to fetch an object from the disk + transmission) increases much more aggressively as the end system becomes congested (thrashing effect).

Figure 11 shows that iGreedyP approximates closely the optimal with request peering. By employing request peering, a significant reduction in the average cost may be achieved. This reduction grows as peers become able to communicate more economically, i.e., when the cost of accessing a peer reduces (i.e, with smaller values of *peercost*). Notice that for $S = 10000$ the cost incurred under iGreedyP with $peercost = 0$ is half the corresponding cost of iGreedy, i.e., the direct links between peers lead to a 50% reduction of cost when the communication over the peer links incurs zero cost; this is frequently the actual case because peer links are usually local direct connections between geographically neighboring networks, thus can be used at approximately no cost.

5 Related work

The *placement of web-server replicas* (mirrors) is a commonly employed way of adding storage capacity to the internet. A number of recent works have addressed the fundamental questions of how many mirrors are needed and where to place them [4, 5, 6]. Web-proxies, unlike mirrors[5], store a fraction of the content of a web-site and do that for multiple web-sites. The content can be statically replicated on the proxy or it can be refreshed by dynamic request-driven caching/replacement algorithms such as LRU, LFU and their variants. Web proxy placement has been studied in [3, 43] – where it is assumed that all proxies replicate the same set of (most popular) objects and, thus, no cooperation between different proxies exist (as allowed in our work) – and in [25] where it is assumed that each proxy has a known hit ratio. The dimensioning of a two level hierarchical cache that operates under LRU has been studied in [9] for the special case of homogeneous demands (i.e., all clients request objects at the same rate using a common popularity distribution over the entire object universe). Other works on the performance of hierarchical caching assume that proxies have infinite storage capacity [10, 11].

Much attention has been paid to the closely related *object placement problem*, under which proxy locations and storage capacities are known and the challenge is to fill up each proxy with an appropriate set of objects so as to optimize the overall performance (delay, bandwidth consumption).

[5]Some authors use the terms proxy and mirror interchangeably. Here we make a clear distinction between the two terms.

The object placement has been studied in [28] for hierarchical caches. Compared to that work we add an additional degree of freedom by deriving cache capacities as well, and avoid the use of approximate distance metrics. Recently, the authors of [44] have combined the study of proxy and object placement. Their work differs from ours in a number of ways, the most important being that they do not guarantee the allocation of an exact amount of storage but constrain the allocation of storage implicitly by considering write costs.

6 Conclusions and future work

This paper considers the problem of how to best allocate an available storage budget to the nodes of a hierarchical content distribution system. The current work addresses the problem of allocating a storage resource differently than previous attempts, taking into consideration related resource allocation subproblems that affect it. The dependencies among the subproblems are not neglected under the current approach and, thus, an optimal solution for all the subproblems is concurrently derived, guaranteeing optimal overall performance. Since the complexity in deriving this optimal solution is high, fast/efficient heuristic algorithms are derived as well. To this end, we have proposed iGreedy, a linear time efficient heuristic algorithm. iGreedy and its variants (for workload balancing and request peering) are shown to achieve a performance that is very close to the optimal.

iGreedy may be used for the derivation of a joint storage capacity allocation and object placement plan to be employed in a system that performs replication, e.g., a CDN. Alternatively, just the derived per node storage capacity allocation may be used for the dimensioning of a hierarchical system that operates under a request driven caching/replacement algorithm. The provisioning of storage under iGreedy is compared to that under two empirical methods and a gain is demonstrated. Additionally, iGreedy is used for the derivation of numerical results that provide insight into the effects of user request patterns (skewness of demand, homogeneity of demand) on the vertical dimensioning of a hierarchical system.

iGreedy has been modified to cater to load balancing and request peering. Load balancing is shown to be able to provide for an even spread of load in the hierarchy, at the cost of a small increase in average distance between users and objects, which, however, may be compensated for a substantial reduction in the delay as a result of accessing uncongested end-systems. Request peering is shown to be able to provide for a significantly reduced cost. Request peering may even lead to the halving of the overall cost, when utilizing inexpensive direct peer links.

An interesting line of future work would be to investigate mechanisms that can provide for an efficient storage capacity allocation in a distributed manner. A distributed computation has several advantages, including better scaling properties and reduced exchange of information and would be particularly suitable for networks not operating under a central authority (as in a CDN) but rather having to cater to multiple local utilities. Finally, although it has been shown that the optimal storage capacity allocation in a tree can be derived in polynomial time, it remains to be seen whether the same is possible when considering per-node load constraints (Sect. 4.1) or allowing for cooperating peers (Sect. 4.2).

References

[1] Cao, P., Irani, S.: Cost-aware WWW proxy caching algorithms. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems. (1997) 193–206

[2] Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking **8** (2000) 281–293

[3] Krishnan, P., Raz, D., Shavit, Y.: The cache location problem. IEEE/ACM Transactions on Networking **8** (2000) 568–581

[4] Li, B., Golin, M.J., Italiano, G.F., Deng, X., Sohraby, K.: On the optimal placement of web proxies in the internet. In: Proceedings of the Conference on Computer Communications (IEEE Infocom), New York (1999)

[5] Qiu, L., Padmanabhan, V., Voelker, G.: On the placement of web server replicas. In: Proceedings of the Conference on Computer Communications (IEEE Infocom), Anchorage, Alaska (2001)

[6] Cronin, E., Jamin, S., Jin, C., Kurc, A.R., Raz, D., Shavitt, Y.: Constraint mirror placement on the internet. IEEE Journal on Selected Areas in Communications **20** (2002)

[7] Pan, J., Hou, Y.T., Li, B.: An overview DNS-based server selection in content distribution networks. Computer Networks **43** (2003)

[8] Fonseca, R., Almeida, V., Crovella, M., Abrahao, B.: On the intrinsic locality properties of web reference streams. In: Proceedings of the Conference on Computer Communications (IEEE Infocom), San Francisco (2003)

[9] Kelly, T., Reeves, D.: Optimal web cache sizing: scalable methods for exact solutions. Computer Communications **24** (2001) 163–173

[10] Rodriguez, P., Spanner, C., Biersack, E.W.: Analysis of web caching architectures: Hierarchical and distributed caching. IEEE/ACM Transactions on Networking **9** (2001)

[11] Gadde, S., Chase, J., Rabinovich, M.: Web caching and content distribution: A view from the interior. Computer Communications **24** (2002)

[12] Saroiu, S., Gummadi, K.P., Dunn, R.J., Gribble, S.D., Levy, H.M.: An analysis of internet content delivery systems. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002). (2002)

[13] Chesire, M., Wolman, A., Voelker, G.M., Levy, H.M.: Measurement and analysis of a streaming-media workload. In: Proceedings of USITS. (2001)

[14] Gribble, S.D., Brewer, E.: System design issues for internet middleware service: Deductions from a large client trace. In: Proceedings of the First USENIX Symposium on Internet Technologies and Systems. (1999)

[15] Wolman, A., Voelker, G., Sharma, N., Cardwell, N., Brown, M., Landray, T., Pinnel, D., Karlin, A., Levy, H.: Organization-based analysis of web-object sharing and caching. In: Proceedings of the Second USENIX Symposium on Internet Technologies and Systems. (1999)

[16] Gibson, G.A., Van Meter, R.: Network attached storage architecture. Communications of the ACM **43** (2000) 37–45

[17] Turner, D.A., Ross, K.W.: A lightweight currency paradigm for the P2P reseource market (2003) [submitted work].

[18] IBM Corp.: Autonomic Computing initiative (2002) `http://www.research.ibm.com/autonomic/`.

[19] Cohen, E., Kaplan, H.: Balanced-replication algorithms for distribution trees. In: Proceedings of the 10th Annual European Symposium on Algorithms (ESA), Rome, Italy (2002)

[20] Williamson, C.: On filter effects in web caching hierarchies. ACM Transactions on Internet Technology **2** (2002)

[21] Wessels, D., Claffy, K.: ICP and the Squid web cache. IEEE Journal on Selected Areas in Communications **16** (1998)

[22] Ranganathan, K., Foster, I.: Identifying dynamic replication strategies for a high performance data grid. In: Proceedings of the International Workshop on Grid Computing, Denver, Colorado (2001)

[23] Garces-Erice, L., Biersack, E.W., Ross, K.W., Felber, P.A., , Urvoy-Keller, G.: Hierarchical P2P systems. In: Proceedings of ACM/IFIP International Conference on Parallel and Distributed Computing (Euro-Par), Klagenfurt, Austria (2003)

[24] Bartal, Y.: On approximating arbitrary metrics by tree metrics. In: Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science (IEEE FOCS). (1996)

[25] Guha, S., Meyerson, A., Munagala, K.: Hierarchical placement and network design problems. In: Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (IEEE FOCS), Redondo Beach, CA (2000)

[26] Kariv, O., Hakimi, S.: An algorithmic approach to network location problems, part II: p-medians. SIAM Journal on Applied Mathematics **37** (1979) 539–560

[27] Charikar, M., Guha, S., Shmoys, D.B., Tardos, E.: A constant factor approximation algorithm for the k-median problem. In: Proceedings of the 31st Annual Symposium on the Theory of Computing (ACM STOC). (1999)

[28] Korupolu, M.R., Plaxton, C.G., Rajaraman, R.: Placement algorithms for hierarchical cooperative caching. In: Proceedings of the 10th Annual Symposium on Discrete Algorithms (ACM-SIAM SODA). (1999) 586 – 595

[29] Rabinovich, M.: Issues in web content replication. Data Engineering Bulletin (invited paper) **21** (1998)

[30] Laoutaris, N., Zissimopoulos, V., Stavrakakis, I.: Joint object placement and node dimensioning for internet content distribution. Information Processing Letters (2004) [to appear].

[31] Vigneron, A., Gao, L., Golin, M., Italiano, G., Li, B.: An algorithm for finding a k-median in a directed tree. Information Processing Letters **74** (2000) 81–88

[32] Tamir, A.: An $O(pn^2)$ algorithm for p-median and related problems on tree graphs. Operations Research Letters **19** (1996) 59–64

[33] Papadimitriou, C.H., Steiglitz, K. In: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, New York (1998)

[34] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. In: Introduction to Algorithms, 2nd Edition. MIT Press, Cambridge, Massachusetts (2001)

[35] Breslau, L., Cao, P., Fan, L., Philips, G., Shenker, S.: Web caching and Zipf-like distributions: Evidence and implications. In: Proceedings of the Conference on Computer Communications (IEEE Infocom), New York (1999)

[36] Sripanidkulchai, K.: The popularity of gnutella queries and its implication on scalability (2001) white paper online at `http://www-2.cs.cmu.edu/~kunwadee/research/p2p/gnu`.

[37] Nussbaumer, J.P., Patel, B.V., Schaffa, F., Sterbenz, J.P.G.: Networking requirements for interactive video on demand. IEEE Journal on Selected Areas in Communications **13,5** (1995) 779–787

[38] Nonnenmacher, J., Biersack, E.: Performance modelling of reliable multicast transmission. In: Proceedings of the Conference on Computer Communications (IEEE Infocom), Kobe, Japan (1997)

[39] Gummadi, K.P., Dunn, R.J., Saroiu, S., Gribble, S.D., Levy, H.M., Zahorjan, J.: Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM Press (2003) 314–329

[40] Chen, Y., Qiu, L., Chen, W., Nguyen, L., Katz, R.H.: Efficient and adaptive web replication using content clustering. IEEE Journal on Selected Areas in Communications **21** (2003)

[41] Che, H., Tung, Y., Wang, Z.: Hierarchical web caching systems: Modeling, design and experimental results. IEEE Journal on Selected Areas in Communications **20** (2002)

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

[43] Bassali, H.S., Kamath, K.M., Hosamani, R.B., Gao, L.: Hierarchy-aware algorithms for CDN proxy placement in the internet. Computer Communications **26** (2003) 251–263

[44] Xu, J., Li, B., Lee, D.L.: Placement problems for transparent data replication proxy services. IEEE Journal on Selected Areas in Communications **20** (2002)