On the Optimization of Storage Capacity Allocation for Content Distribution*

Nikolaos Laoutaris, Vassilios Zissimopoulos, Ioannis Stavrakakis

June 11, 2004

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 [3, 9], proxy placement algorithms [20, 22, 27, 8], object placement algorithms [19, 16], and request redirection mechanisms [25]. In fact, the only published paper on the dimensioning of web proxies that we are aware of is due to Kelly and Reeves [18] whereas the majority of related works in the field have disregarded storage dimensioning issues by assuming the existence of infinite storage capacity [30, 10].

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. [31] 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 [31].

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 [5]. Although a 1997 study [13] 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 [38]. This trend seems to have persisted, as recent studies (2001) have shown that stored video and audio files have by now become almost pervasive [5].

Such objects can exhaust the capacity of a cache or a CDN node, even under a low price of storage. It 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.

Our methodology can be used 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 [12]. 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. 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 [34]). 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 [15]). 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 models the prob-

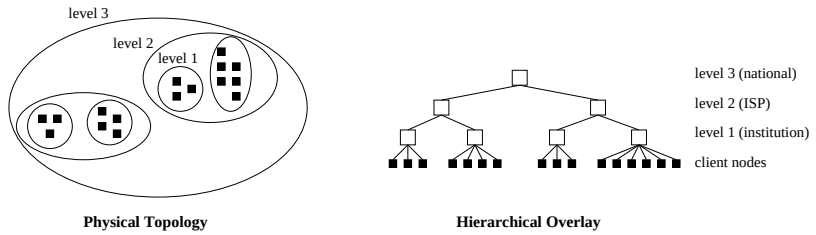

Figure 1: A three level hierarchical content distribution overlay built upon of a set of client nodes using proximity/administration based clustering.

lem with an integer linear program. 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 in various common scenarios 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 when the available storage changes, 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 [6, 37] 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 [37] 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 [36], hierarchical data storage in Grid computing [29], hierarchical peer-to-peer networks [11]); (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 [1]. 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 from 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 heuristic solutions for it; the same section contains a number of numerical results. In Section 4 the algorithms (exact and heuristic) 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. Finally, Sect. 5 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 [22], or by assuming that the hit rate of each installed proxy is a priori known [14]. 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 [17, 4] 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 employ 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 [16, 19] 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 $p_j : \mathcal{O} \rightarrow [0, 1]$; a tree graph T with a node set of n nodes, $\mathcal{V}$, and a distance function $d_{j,v} : \mathcal{J} \times \mathcal{V} \rightarrow R^+$ associated with the jth

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

leaf node and node v; this distance captures the cost paid when client j retrieves an object from node v. 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:

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

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. This cost models only "read" operations from clients. Adding "write" (update) operations from content creators is possible but as stated in [28] 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 where in the network to place storage, how much of it, and which objects to store, so as to achieve a minimal cost (in terms of fetch distance) subject to a single storage 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 [37] (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 [28].

3.2 Integer linear programming formulation of an optimal solution

In this section the storage capacity allocation problem is modeled with an integer linear program (ILP). 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 an auxiliary 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 of (3) is equivalent to the minimization of (1) (the two objectives differ by sign and a constant). Notice that only the $X_{j,v}(k)$'s contribute to the objective function and the $\delta_v(k)$'s do not.

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 [17, 4]. We have shown that these generalized problems can be solved in polynomial time for the case of tree graphs [21] 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 $O(\max\{n^4N, n^2N^2\})$. The solution is sought by first solving a series of k-medians for the different objects using known dynamic programming techniques [35, 33] 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 [26], using again dynamic programming.

From a theoretical point of view, the $O(\max\{n^4N, n^2N^2\})$ result is rather attractive as it involves small powers of the input. In practice, however, such a result might be 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 requested 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 considered

to be difficult to handle in practice [19], especially when there is a requirement for frequent re-optimizations. 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 heuristic 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 lower complexity, provide for a close approximation of the optimal in all our scenarios, and lend themselves to incremental use (such need arising when the available storage changes dynamically).

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 is allocated. Objects are placed in a globally 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 elsewhere in the tree. In the first iteration the algorithm selects an node-object pair (v_1, k_1) that yields a maximum gain and places k_1 at v_1. Subsequent decisions place an object k in node v when the $gain_v(k)$ is maximum among all (v, k) pairs that have not been selected yet; $gain_v(k)$ is defined as:

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

The parenthesized quantity in (7) is the distance reduction that is achieved by client j when fetching object k from node v instead from node $par_v(k)$, i.e., v's closest parent that caches k; initially it is the origin server that is the closest parent for all objects and all nodes but this changes as additional copies get replicated elsewhere in the tree.

Greedy is presented in detail in Table 1. Lines 1-7 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 [7] data structure $g(v, \cdot)$ (n max-heaps, one for its node v); the max-heaps are used so as to allow locating the most valuable object for each node in $O(1)$ (this does not require sorting the N objects). In the S iterations of the algorithm the following three steps are executed: (1) (v^*, k^*), the node-object pair that produces the maximum gain among the set of 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^* up to $par_v(k)$ 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; (3) for each descendant u of v^* that does not hold k^* and has v^* as closest parent with 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 15-18) – the update is in this case necessary because v^* becomes now the closest ancestor with k^* for some of its descendants,

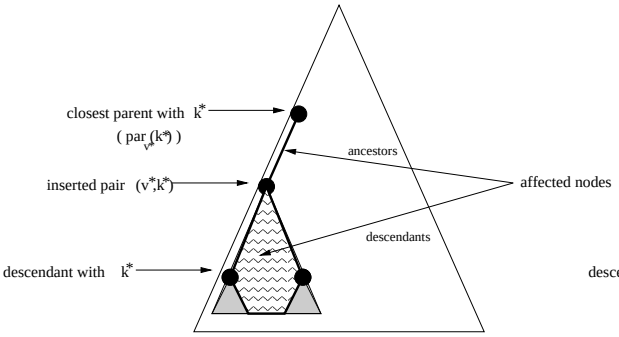

Figure 2: Illustration of the two groups of nodes that are affected from the placement of object k^* at node v^* under iGreedy: (1) descendants that do not store k^* and have v^* as closest parent with k^* (laying inside the wave-filled area); (2) ancestors that do not store k^*, up to the closest parent with k^* (laying on the thick line that connects v^* with $par_{v^*}(k^*)$). Notice that nodes in the solid gray areas are not affected as they do not have v^* as closest parent with k^*.

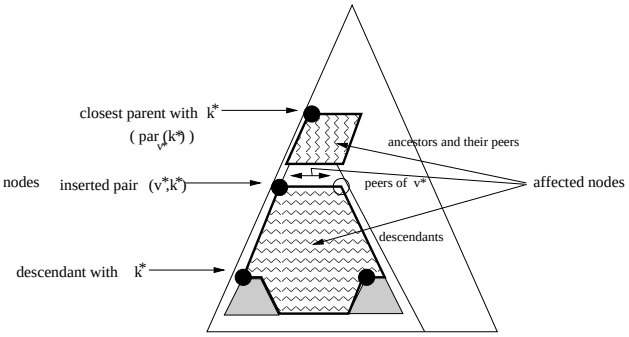

Figure 3: Illustration of the two groups of nodes that are affected from the placement of object k^* at node v^* under iGreedyP: (1) peers of v^* as well as all of their descendants and their peers that do not store k^* and have v^* as closest parent with k^* (horizontal wave); (2) ancestors that do not store k^* and their peers, up to the closest parent with k^* (verical wave). Nodes in the solid gray areas are not affected as they do not have v^* as closest parent with k^*.

thus effectively reducing the previously computed gain for them that was based on a more distant parent. The two groups of nodes that are affected from the placement of object k^* at node v^* are illustrated in Fig. 2. Notice that the various affected max-heaps need to be re-organized since one of their elements changes value; this must be done so as to maintain the max-heap property (i.e., have the maximum value accessible in $O(1)$ from the root of the max-heap).

A straightforward evaluation of the gain function $gain_v(k)$ requires $O(n)$ complexity. Since there are n nodes, evaluating $gain_v(k)$ for a given k for all nodes v would require $O(n^2)$ complexity, if each evaluation were to be carried out independently. Such an independent operation would involve however much overhead due to unnecessary repetitious work. See that the evaluation of the gain function depends on knowing the request rate that goes into the node and the closest parent that stores the object. To obtain the request rate for object k at node v, it suffices to know the corresponding rates at its children, and then sum the rates that go into children that do not cache k; knowing these rates, makes re-examining the entire subtree of v down to the leaf level redundant. Similar observation can be made regarding the identification of the closest parent. We make use of these observations in order to be able to evaluate $gain_v(k)$ for a particular pair (v, k) in $O(1)$. This allows calculating the gain for placing k in each of the n nodes in $O(n)$ instead of $O(n^2)$. In Appendix A we show how this can be achieved by first pre-computing information pertaining to request rates and closest parents and then using it to calculate up to n gain function for a given object in just $O(n)$. Since the gain function is $O(1)$ following the pre-computation step (occurring once at the beginning of each iteration), the complexity of each iteration of the Greedy algorithm depends on the number of nodes that are involved in the iteration and the update of the corresponding data structures.

The initial creation of the n max-heaps can be done in $O(nN)$ (each max-heap containing N values). At the beginning of each iteration there is the pre-processing step to get $rate_v(k)$ and $par_v(k)$ in $O(n)$ as explained in Appendix A. 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 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

```
 1: for each v ∈ V
 2:    for each k ∈ O
 3:       g(v,k) = gain_v(k)
 4:       insert  g(v,k)  in max-heap  g(v,·)
 5:    end for
 6: end for
 7: i = 1, P = {(v,k) : v ∈ V, k ∈ O}
 8: while i ≤ S
 9:    select (v*,k*) ∈ P : g(v*,k*) ≥ g(v,k) ∀(v,k) ∈ P
10:    P = P − {(v*,k*)}, re-organize max-heap g(v*,·)
11:   for each u ∈ path(v*, par_v*(k*)) not caching k*
12:       g(u,k*) = gain_u(k*)
13:       re-organize max-heap g(u,·)
14:   end for
15:   for each u ∈ subtree(v*) not caching k*
                  with par_u(k*) = v*
16:       g(u,k*) = gain_u(k*)
17:       re-organize max-heap g(u,·)
18:   end for
19:   i = i + 1
20: end while
```

Table 1: The Greedy algorithm.

```
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.

n max-heaps $g(v,\cdot)$ (the latter might be unnecessary since n is typically rather small). The second step requires in the worst case the update of $L-1$ ancestors that do not cache k^*, L being the height of the tree. The function $gain_v(k^*)$ is re-evaluated for these nodes (each evaluation in $O(1)$) and the corresponding max-heap is re-organized in order to maintain the heap property; this can be done in $O(\log N)$ for a heap with N objects. The third step requires updating the descendants of v^* that are affected by storing k^* at v^*; this can be done in $O(n \log N)$. As a result the S iterations of the algorithm require $O(S \cdot (n + L \log N + n \log N))$, 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. Since clients are located at the leaves of the tree, if an object is placed at all children of a node u, then 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 "barren" objects. The Greedy algorithm often introduces barren 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 barren 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 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). 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 barren 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 1 *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 barren 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 $T(S) = 2 \cdot S - 1$. □

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 [2] and multimedia clips [5]. The popularity of P2P [31] 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 [32].

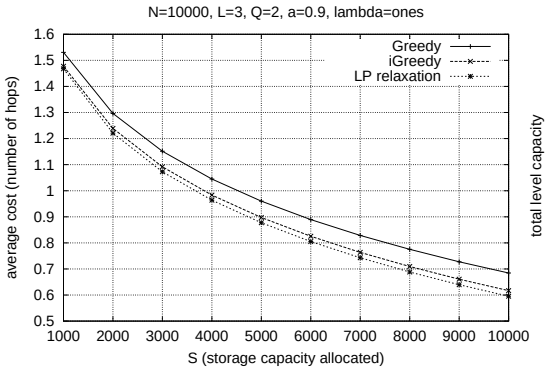

Figure 4: The average cost of Greedy, iGreedy and LP-relaxation.

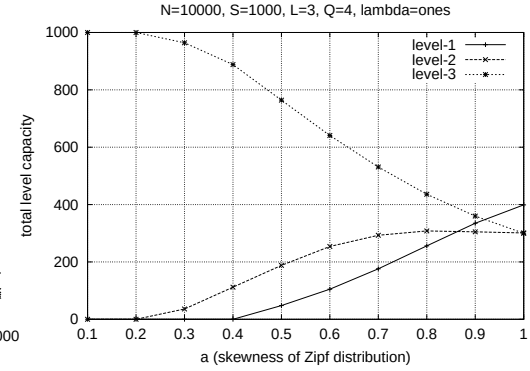

Figure 5: The effect of skewness of popularity on the per-level allocation of storage under iGreedy.

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 [24, 30]; it has also be seen that numerical results from regular tree topologies are in good accordance with experimental results from actual internet tree topologies [23]. The entire set of parameters (demand and topology) for each experiment is indicated in the title of the corresponding graph. 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.

3.6.1 Quality of the approximation

Figure 4 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. From the graph it may be seen that iGreedy is no more than 3% away from the optimal while Greedy may deviate as much as 14% in the presented results. The performance gap between the two owes to the waste of a significant amount of storage in barren objects under Greedy.

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 5 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) increase the amount of storage that is allocated to the leaves (level-1), while less skewed distributions allocate more storage to the root (level-3). 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

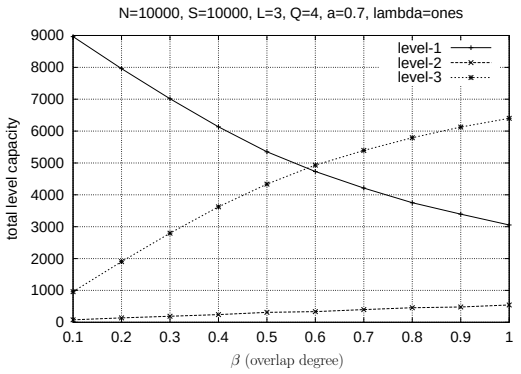

Figure 6: The effect of non-homogeneous demand on the per-level allocation of storage under iGreedy.

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

Table 3: Additional steps for the iGreedyLB algorithm.

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 6 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 6 shows that the root level (level-3) of a hierarchical system is 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.

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.

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

The resulting new ILP cannot be solved by the exact algorithm of Sect. 3.2 due to these extra load constraints.

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 3 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 the node. The function $load_v(k)$ captures the load that would be imposed on node v if it was to store object k; it is defined in eq. (13) in Appendix A. 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).

The joint consideration of load constraints increases the per-iteration complexity by $O(N)$, thus making it $O(Nn \log N)$; the increase owes to the fact that up to $O(N)$ objects may be inserted/deleted from a max-heap due to load-related implications emanating from an object placement. Thus the overall complexity becomes quadratically dependent on N, matching the complexity required by the optimal solution without load constraints in a tree graph (see Sect. 3.2). Making up for the added complexity is the fact that iGreedyLB solves a much more difficult problem.

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 multiple times at the lower levels of the hierarchy in order to distribute the load. Following this intuition, however, leads to the opposite result as it puts all the load to the leaf nodes, ending up overloading them.

iGreedyLB avoids this phenomenon by preserving the per-node load constraints and distributing the load more evenly. 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 here). 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 [36] 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:

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

$peers(v)$ denotes the set of nodes that are peers to v, including v itself. $peercost$ is a fixed[2] 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:

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

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

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

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

[2]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.

the basic iGreedy algorithm. First, the gain function (7) must be updated as follows:

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

The first expression in (12) corresponds to the case that no peer of v stores 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 of v, that do not fetch k from any node or peer in the path[3] from j to u. The second line corresponds to the case that some peer(s) u already store k, thus the extra gain of storing it also at v will be equal to the extra distance *peercost* that will be "spared" (not have to be crossed) by some of the descendants of v. The second modification refers to the removal of "barren" 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 barren the copy of k at the father (without requiring that all children cache k as is the case with iGreedy).

Table 4 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 iteration of the algorithm executes the code in lines 9-27. 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: (1) the peers of v^* (lines 11-14) as well as all of their descendants and their peers that do not store k^* (lines 15-18); (2) the ancestors of v^* and their peers that do not store k^* (lines 19-22) – Fig. 3 illustrates the above mentioned groups of nodes. 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 23-27).

Although iGreedyP has to update a larger group of nodes in each iteration (compare figures 2 and 3), its asymptotic complexity is the same with iGreedy (this is due to the fact that iGreedy may also have to update $O(n)$ nodes).

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

[3] By abuse of notation we write $k \notin peers(w)$ to mean that none of the nodes in $peers(w)$ stores object k.

```
 1: for each v ∈ V
 2:    for each k ∈ O
 3:        g(v,k) = gain_v(k)
 4:        insert  g(v,k)  in max-heap  g(v,·)
 5:    end for
 6: end for
 7: i = 1, P = {(v,k) : v ∈ V, k ∈ O}
 8: while i ≤ S
 9:   select (v*,k*) ∈ P : g(v*,k*) ≥ g(v,k)∀(v,k) ∈ P
10:   P = P − {(v*,k*)}, re-organize max-heap g(v*,·)
11:   for each peer u of v* not caching k*
12:       g(u,k*) = gain_u(k*)
13:       re-organize max-heap g(u,·)
14:   end for
15:   for each w : w ∈ peers(u), u ∈ subtree(v'), v' ∈ peers(v*)
                  w  not caching k*  with par_w(k*) = v*
16:       g(w,k*) = gain_w(k*)
17:       re-organize max-heap g(w,·)
18:   end for
19:   for each w : w ∈ peers(u), u ∈ ancestors(v*)
                  w  not caching k*
20:       g(w,k*) = gain_w(k*)
21:       re-organize max-heap g(w,·)
22:   end for
23:   if k* cached at father(v*)
24:       remove k* from father(v*)
25:       i = i − 1
26:   end if
27:   i = i + 1
28: end while
```

Table 4: The iGreedyP algorithm.

selected setting, will avoid the possibility of 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 7 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 approach the best performance without load balancing. The smallest value, $maxload = 1.0$, yields the best load balancing but also increases the cost, as the 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 8 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 9 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 closer 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

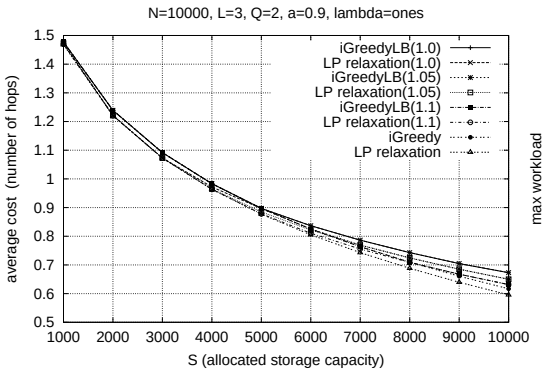

Figure 7: The effect of load balancing on the average cost under iGreedyLB and the LP-relaxation of the ILP with load balancing.

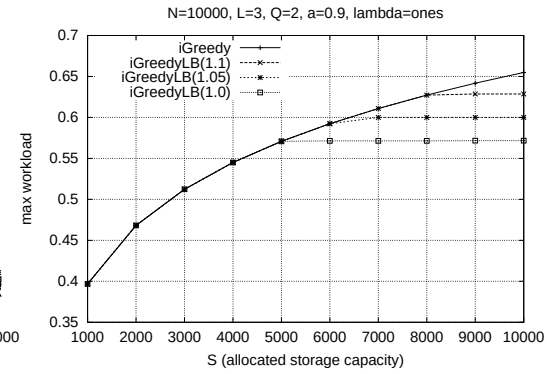

Figure 8: The effect of load balancing on the maximum load that may be imposed on any node in the hierarchy under iGreedyLB.

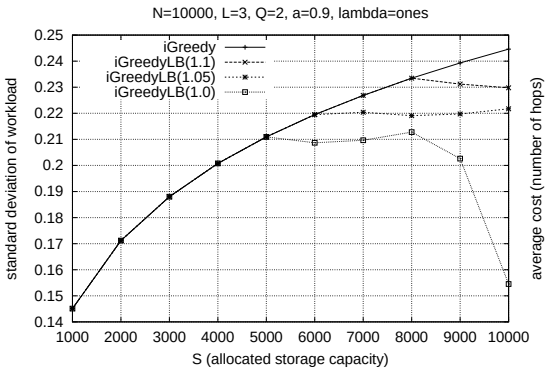

Figure 9: The effect of load balancing on the standard deviation of load in the nodes of the hierarchy under iGreedyLB.

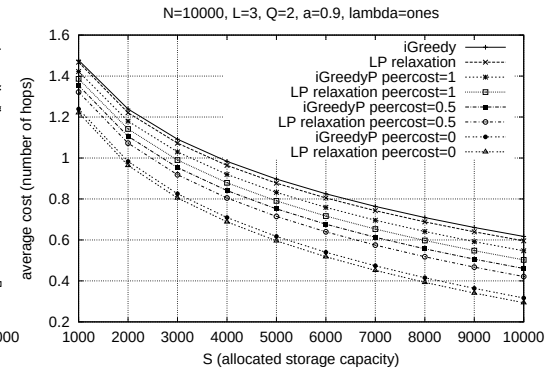

Figure 10: The effect of peering on the average cost incurred under iGreedyP and LP-relaxation of the ILP with peering, for different peering costs.

some nodes are congested while others are severely underutilized, especially those at higher levels.

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 [37]. Notice that the 10% increase in the average cost (number of hops) between iGreedy and iGreedyLB(1.0) results in approximately 15% reduction of maximum load and a 40% 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 10 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 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. 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).

Acknowledgements The authors would like to thank the four reviewers for their efforts in improving the overall quality of the paper, and particularly reviewer 2 for valuable suggestion relating to the employed cost function and the complexity analysis of the heuristic algorithms.

A Evaluating the cost of placing object k in all nodes $v \in \mathcal{V}$ that do not store it in $O(n)$

A.1 Bottom-up pre-processing – request rates

Let $rate_v(k) = \sum_{\substack{j \in leaves(v) \\ k \notin path(j,v)}} p_j(k) \cdot \lambda_j$ denote the request rate for k that goes through (i.e., not serviced at) node v, meaning that v does not store k ($rate_v(k) = 0$ when v does store k). Then for

a given k we can compute all $rate_v(k)$ for the different v by a simple bottom-up traversal of the tree by using:

$$rate_v(k) = \begin{cases} p_v(k) \cdot \lambda_v & \text{if } v \text{ is a leaf node} \\ \sum_{u \in children(v)} rate_u(k) & \text{otherwise} \end{cases} \quad (13)$$

If n_l is the number of nodes at level l, then clearly eq. (13) requires $O(n_1 + n_2 + ... + n_L) = O(n)$ to compute and store all $rate_v(k)$ for the different v.

A.2 Top-down pre-processing – closest parents

Here we show how to compute $par_v(k)$ for a given k for the different v that do not store k, in $O(n)$. This can be done through:

$$par_v(k) = \begin{cases} os & \text{if } v \text{ is the root} \\ par_{father(v)}(k) & \text{otherwise} \end{cases} \quad (14)$$

Having completed the two preprocessing steps, each requiring $O(n)$, we can evaluate any gain function in $O(1)$ through $gain_v(k) = (par_v(k) - d_v) \cdot rate_v(k)$, where d_v is the distance of node v from the leaf level. This allows to compute the gain function for all nodes that do not store k in $O(n)$ instead of $O(n^2)$ as would be required by an independent evaluation of each one.

References

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

[2] Lee Breslau, Pei Cao, Li Fan, Graham Philips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, New York, March 1999.

[3] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 193–206, December 1997.

[4] Moses Charikar, Sudipto Guha, David B. Shmoys, and Eva Tardos. 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.

[5] Maureen Chesire, Alec Wolman, Geoffrey M. Voelker, and Henry M. Levy. Measurement and analysis of a streaming-media workload. In *Proceedings of USITS*, 2001.

[6] Edith Cohen and Haim Kaplan. Balanced-replication algorithms for distribution trees. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, Rome, Italy, September 2002.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Edition.* MIT Press, Cambridge, Massachusetts, 2001.

[8] Eric Cronin, Sugih Jamin, Cheng Jin, Anthony R. Kurc, Danny Raz, and Yuval Shavitt. Constraint mirror placement on the internet. *IEEE Journal on Selected Areas in Communications*, 20(7), September 2002.

[9] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[10] Syam Gadde, Jeff Chase, and Michael Rabinovich. Web caching and content distribution: A view from the interior. *Computer Communications*, 24(2), February 2002.

[11] Luis Garces-Erice, Ernst W. Biersack, Keith W. Ross, Pascal A. Felber, , and Guillaume Urvoy-Keller. Hierarchical P2P systems. In *Proceedings of ACM/IFIP International Conference on Parallel and Distributed Computing (Euro-Par)*, Klagenfurt, Austria, 2003.

[12] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, November 2000.

[13] S. D. Gribble and E.A. Brewer. 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*, December 1999.

[14] Sudipto Guha, Adam Meyerson, and Kamesh Munagala. Hierarchical placement and network design problems. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (IEEE FOCS)*, Redondo Beach, CA, November 2000.

[15] IBM Corp. Autonomic Computing initiative, 2002. `http://www.research.ibm.com/autonomic/`.

[16] Jussi Kangasharju, James Roberts, and Keith W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, (4):376–383, March 2002.

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

[18] T. Kelly and D. Reeves. Optimal web cache sizing: scalable methods for exact solutions. *Computer Communications*, 24(2):163–173, February 2001.

[19] Madhukar R. Korupolu, C. Greg Plaxton, and Rajmohan Rajaraman. Placement algorithms for hierarchical cooperative caching. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms (ACM-SIAM SODA)*, pages 586 – 595, 1999.

[20] P. Krishnan, Danny Raz, and Yuval Shavit. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–581, October 2000.

[21] Nikolaos Laoutaris, Vassilios Zissimopoulos, and Ioannis Stavrakakis. Joint object placement and node dimensioning for internet content distribution. *Information Processing Letters*, 89(6):273–279, March 2004.

[22] Bo Li, Mordecai J. Golin, Giuseppe F. Italiano, Xin Deng, and Kazem Sohraby. On the optimal placement of web proxies in the internet. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, New York, March 1999.

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

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

[25] Jianping Pan, Y. Thomas Hou, and Bo Li. An overview DNS-based server selection in content distribution networks. *Computer Networks*, 43(6), December 2003.

[26] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, New York, 1998.

[27] Lili Qiu, Venkata Padmanabhan, and Geoffrey Voelker. On the placement of web server replicas. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Anchorage, Alaska, April 2001.

[28] Michael Rabinovich. Issues in web content replication. *Data Engineering Bulletin (invited paper)*, 21(4), December 1998.

[29] K. Ranganathan and I. Foster. Identifying dynamic replication strategies for a high performance data grid. In *Proceedings of the International Workshop on Grid Computing*, Denver, Colorado, November 2001.

[30] Pablo Rodriguez, Christian Spanner, and Ernst W. Biersack. Analysis of web caching architectures: Hierarchical and distributed caching. *IEEE/ACM Transactions on Networking*, 9(4), August 2001.

[31] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of internet content delivery systems. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.

[32] Kunwadee Sripanidkulchai. 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`.

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

[34] David A. Turner and Keith W. Ross. A lightweight currency paradigm for the P2P reseource market, 2003. [submitted work].

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

[36] Duane Wessels and K. Claffy. ICP and the Squid web cache. *IEEE Journal on Selected Areas in Communications*, 16(3), April 1998.

[37] Carey Williamson. On filter effects in web caching hierarchies. *ACM Transactions on Internet Technology*, 2(1), February 2002.

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