

# Do We Need Replica Placement Algorithms in Content Delivery Networks?

Magnus Karlsson and Mallik Mahalingam  
HP Laboratories

1501 Page Mill Road 1134, Palo Alto, CA 94304, U.S.A.

E-mail: karlsson@hpl.hp.com

## Abstract

Numerous replica placement algorithms have been proposed in the literature for use in content delivery networks. However, little has been done to compare the various placement algorithms against each other and against caching. This paper debates whether we need replica placement algorithms in content delivery networks or not.

The paper uses extensive evaluation of algorithms, systems and web workloads to answer this question. We conclude that a simple delayed-LRU caching scheme outperforms, or at least performs as well as the best replica placement algorithms. As LRU caching is less complex than most replica placement algorithms, caching is clearly the preferred option. However, we believe that replica placement algorithms will be necessary once properties such as availability, reliability, performance and bounded update propagation will have to be guaranteed by content delivery networks.

## 1 Introduction

Content delivery networks (CDN) such as Akamai [1] and Digital Island [4] are nowadays used by many web sites as they effectively reduce the client-perceived latency and balance load. They accomplish this by serving content from a dedicated, distributed infrastructure located around the world and close to clients. One of the foremost problems in CDNs is to decide where to place site contents in the CDN infrastructure. The algorithms used so

far can be categorized into two high-level groups; *caching algorithms* and *replica placement algorithms (RPA)*. Caching is generally a completely distributed algorithm that evaluates its placement after each single access, while an RPA is evaluated with the frequency of hours or days and can thus be much more elaborate and even centralized. The replica placement problem is an instance of the classical file allocation problem (FAP) [5], in which only algorithms that allow replicas to be created are considered. There have been a plethora of proposed RPAs [6, 7, 8, 9, 11, 12, 13, 16, 17, 20] (see [10] for an extensive analysis) for possible use in CDNs, but no direct comparison between them or with caching has been performed. The questions we would like to answer is how do the various RPAs compare, and are they any better than simple caching?

In this paper, we quantitatively study previously proposed RPAs to discover under what system conditions and performance metrics, if any, RPAs outperform caching. The algorithms considered are previously proposed algorithms specifically targeting CDNs and some other algorithms from other systems that we believe could be useful in this context. In order to describe the algorithms concisely, we use the replica placement framework presented in [10] and show how local caching algorithms map into the framework as an RPA that is decentralized and evaluated after each single access. We evaluate the performance of these algorithms by simulating the produced placements' impact on the system's client-perceived latency, both as an average over all accesses and as a threshold metric. This is performed using a Internet-like network topology

and a trace taken from a large web server.

We find that taking the number of accesses into account when making a decision with an RPA improves performance substantially, and that the greedy heuristic [16] overall provides the best results for a system without any capacity constraints. The RPAs taking storage constraints into consideration outperform all RPAs that do not, questioning the usefulness of RPAs without storage constraints in CDNs. We find that the fast, decentralized Popularity algorithm [9] and the Greedy Ranking algorithm taking storage into consideration [9] produce the best placements.

When comparing the best RPAs against caching in the same location in the network, we first find that RPAs outperform a simple LRU caching algorithm. The reason for this is the assumption that RPAs know where the closest object of a replica is located. This assumption can be made (up to some system scale) as these algorithms are only evaluated infrequently, e.g., once a day. We then propose to evaluate the caching algorithm with the same low frequency in order for caching to benefit from the same perfect closest replica knowledge. By evaluating a simple LRU caching algorithm as infrequently as RPAs, caching outperforms or performs as well as the best RPAs.

Based upon our experimental results, we conclude that RPAs are not needed for the systems we considered, as a simple caching algorithm can be made to perform as well, if not better than, the best RPAs. However, we believe that RPAs will become useful once availability, reliability and update-propagation latency guarantees have to be provided in a CDN. It is also an open question if our results extend to bandwidth and load constrained systems.

The rest of the paper is organized as follows. In Section 2 we describe the system model and all the placement algorithms that we examine. Section 3 describes our experimental methodology. The core of the paper is Section 4 that reports on experimental results. We then discuss possible fruitful future directions of RPA research in Section 5 and finally, in Section 6, the paper is concluded.

## 2 Assumptions and Algorithms

We start by providing the system model and the assumptions we make in Section 2.1. In Section 2.2, we define the replica placement problem and the algorithms we choose to compare, and how caching can be viewed as an RPA.

### 2.1 System Model

The system considered in this paper is a data repository consisting of a set of  $N$  storage nodes, hereafter just called *nodes*, interconnected with some network. On these nodes, replicas of *objects* ( $K$ ) are stored representing data aggregates such as entire web-sites, directories, or single documents. In the system, *clients* ( $C$ ) generate read accesses to the objects located on the servers. The overall goal of the replica placement problem that we study in this paper is to decide the location of object replicas in the system so as to minimize the client-perceived latency given an existing infrastructure. We will examine both average latency metrics and threshold-based QoS metrics.

### 2.2 Replica Placement Algorithms

We use the classification framework introduced in [10] to describe RPAs concisely and pin-point their differences. This framework is summarized in this Section and the RPAs we consider are mapped into it.

An *algorithm* in this paper consists of a *problem definition* and a *heuristic*. The problem definition consists of a *cost function* that has to be minimized or maximized under some *constraints*. The problem definitions we are interested in for use in CDNs can be divided into two main categories, depending on whether they consider accesses to objects or not. The problem definitions within both these categories can be further classified into two groups, according to whether they consider a single object or multiple objects at a time. A single-object problem definition places objects independently of each other.

Table 1 lists the problem definitions that are considered in this study. They are all problem definitions that have been used in the CDN literature. For a full list of replica placement problem definitions for possible use in CDNs, we refer to [10]. The variables are defined as follows:

**Reads** ( $reads_{ik} \geq 0$ ) : The number of read accesses by a client  $i$  to an object  $k$  per time unit.

**Distance** ( $dist_{ij} \geq 0$ ) : The distance between a client  $i$  and a node  $j$  represented with a metric such as network latency, number of network hops, or total link “cost”.

**Object Size** ( $size_k > 0$ ) : The size of object  $k$  in bytes.

**Access Time** ( $acctime_{jk} \geq 0$ ) : A time-stamp of the last time object  $k$  was accessed at node  $j$ .

**Placement Matrix** ( $x_{jk} \in \{0, 1\}$ ) : Indicates whether node  $j$  stores object  $k$ . This is unknown and will contain the result of the placement.

**Routing Matrix** ( $y_{ijk} \in \{0, 1\}$ ) : Indicates whether client  $i$  sends requests for object  $k$  to node  $j$ . This is unknown and will contain the result of the optimization. For the algorithms in this paper, it is enough to find out either  $x$  or  $y$  as they can be deduced from each other.

The two constraints considered in the CDN literature are the *number-of-replica* constraint ( $\sum_{j \in N} x_{jk} \leq P, \forall k$ ) that limits the number of replicas placed, and the *storage capacity* constraint ( $\sum_{k \in K} size_k \cdot x_{jk} \leq S_j, \forall j$ ) that places an upper bound on the storage capacity of the node.

The heuristics used to achieve the goal set out by the problem definition can be described using three primitives: *metric scope*, *approximation method* and *cost function simplification*. Metric scope is the set of clients, nodes, objects that are considered when making a placement decision. The heuristic can specify this scope anywhere from considering zero to all. If, for example, only one node is considered, the heuristic is decentralized and has to be

run everywhere, but if all nodes are considered, it is centralized. If the object scope is local, only objects stored locally or accessed locally will be considered.

An approximation method is the technique used to make the placement decision. The ones considered in this paper are *ranking*, *improvement*, and *Lagrangian relaxation*. Ranking starts with the computation of the cost impact of all possible combinations (within the metric scope) of placing one extra object on one node; sorts these costs and selects the best one that does not violate any constraints. If a constraint is violated, it tries the next placement in the list. This is repeated until no more objects can be placed. A *greedy ranking* heuristic recomputes the cost function after each object is placed. The specific improvement heuristic used in this paper is the 2-distance improvement heuristic [3]. It starts with an initial placement. This placement could be random or seeded by another heuristic. It then randomly picks one object and puts it on another node, making sure that the constraints are still satisfied. If this placement has a better cost it keeps it, otherwise it reverts back to the previous one. This process is then repeated a predefined number of times. Lagrangian relaxation [19] is a method that relaxes the constraints of the original problem by moving them into the cost function, which makes the new problem easier to solve.

On top of this, an algorithm can also modify (usually simplify) the cost function that the original problem definition specified. For example, the problem definition might specify the cost function  $\sum_{i \in C} \sum_{j \in N} reads_{ik} \cdot dist_{ij} \cdot y_{ijk}$ . But a heuristic might disregard the distance and use just  $\sum_{i \in C} \sum_{j \in N} reads_{ik} \cdot y_{ijk}$ . It might work well anyway.

Table 2 lists the heuristics we examine in this paper. A short description of each of them follows. *Greedy Global* [9, 16] is a centralized heuristic that uses greedy ranking of global data in the system. *Ranking Local* [9, 12] is a decentralized ranking heuristics that just uses the data available in each node. *Popularity* [9] is the same as ranking local with the exception that it always only considers the number of accesses as the ranking criteria. *Rank-*

ID	Cost function	References
<b>Group 1: Does not consider any object accesses.</b>		
	Single Object	
maxdist	$\max_{i \in C, j \in N} dist_{ij} \cdot y_{ijk}$	[6, 7, 8]
dist	$\sum_{i \in C} \sum_{j \in N} dist_{ij} \cdot y_{ijk}$	[6, 8]
<b>Group 2: Considers read accesses to objects.</b>		
	Single Object	
so_readdist	$\sum_{i \in C} \sum_{j \in N} reads_{ik} \cdot dist_{ij} \cdot y_{ijk}$	[6, 8, 13, 16, 17]
	Multiple Objects	
mo_readdist	$\sum_{i \in C} \sum_{j \in N} \sum_{k \in K} reads_{ik} \cdot dist_{ij} \cdot y_{ijk}$	[9, 11]

Table 1: The problem definitions that have been proposed in the CDN literature and investigated in this paper.

*ing Dist* [12] is the same as popularity except that it uses distance as the cost function. *Hotspot* [16] is a centralized ranking heuristic that only considers accesses from clients located within a specified radius around each node. *Fan-Out* [8, 17] places objects at the nodes with the highest fan-out, irrespective of the actual cost function. *Swap* [3] is the centralized 2-dist improvement heuristic described previously. *Greedy+* (called 1-Greedy in [8]) seeds the Swap heuristic with the result of the Greedy Global heuristic. *Lagrangian* [3, 19] is the relaxation technique described previously, and *Lagrangian+* is the Swap heuristic seeded with that result.

In this paper, we use plain LRU caching as a representative from the caching domain. Caching maps nicely into the framework as can be seen in Table 2. The key difference that is missing from the Table is that caching is evaluated after each single access, while an RPA is usually evaluated much less frequently.

The possible constraints column in Table 2 lists what constraints are possible together with a heuristic. Note, that this is also dictated by the cost function in the problem definition. The ones that do not consider object accesses, either in their problem definition or heuristic, place every single object in the same nodes as the placement is independent of any object property. Thus, all single object problem definitions cannot take a storage constraint into consideration as it is intra-object dependent, and local heuristics cannot guarantee a number-of-replica constraint as it has an intra-node dependency.

### 3 Experimental Methodology

To compare the RPAs, we evaluate them using trace-driven simulations. We measure the performance impact of their placements on a real Internet topology for a web-server workload, that are explained in Section 3.1. We also describe the system performance metrics that we use to compare the algorithms in Section 3.2, and the tool we used to generate the placements of all these algorithms, in Section 3.3.

#### 3.1 Web Workloads and Topology

We use the WorldCup98 web logs [2] (day 50 to 59). To reduce the client population to a tractable size, we clustered all encountered client IP addresses according to the Autonomous System (AS) they belong to using BGP prefixes [18]. This clustering preserves the topological locality and reduces the number of clients from 2,770,107 to 5,399. These clusters represent both the clients in the system and the nodes on which objects may be placed. To generate systems with less nodes than the number in the log (e.g., 300), we choose the desired number of nodes in a way that preserves the original access distribution from [2]. Each URL is treated as a separate object. There are 34,000 objects in the WorldCup98 log and they are reduced to 10,000 by random selection. We assume that objects are of uniform size, thus we do not evaluate the algorithms effectiveness on variable size objects.

Heuristic	Approximation Method	Metric Scope			Cost Function Simplification	Possible Constraints
		Client	Node	Object		
Greedy Global [9, 16]	Greedy ranking	all	all	all	-	P,SC
Ranking Local [9, 12]	Ranking	local	one	local	-	SC
Popularity [9]	Ranking	local	one	local	$reads_{ik}$	SC
Ranking Dist [12]	Ranking	local	one	local	$dist_{ij}$	SC
Hotspot [16]	Ranking	vicinity	all	all	$reads_{ik}$	P,SC
Fan-Out [8, 17]	Ranking	indep	all	indep	$fanout_j$	P
Swap [3]	Improvement	all	all	all	-	P,SC
Greedy+ [8]	Greedy ranking + Improvement	all	all	all	-	P,SC
Lagrangian [3, 19]	Lagrangian	all	all	all	-	P,SC
Lagrangian+ [3, 19]	Lagrangian + Improvement	all	all	all	-	P,SC
LRU Caching	Ranking	local	one	local	$acctime_{jk}$	SC

Table 2: The heuristics examined in this paper. “Vicinity” means a number of clients within some radius of the node. The possible constraints column lists the constraints that a heuristic can consider.

We derive the distance matrix  $dist_{ij}$  using the AS level topology that we generate by processing BGP reports [18]. In an ideal world,  $dist_{ij}$  would represent the average latency between nodes  $i$  to  $j$ . However, this is impossible to measure, unless one happens to be on that specific routing path. So, we obtain latency approximations by measuring the number of AS-level hops between two nodes by constructing an AS graph and use Dijkstra’s shortest path algorithm between all pairs of AS in this graph. It has been shown that AS-level hops is a fair approximation of actual latencies on the Internet [15]. To turn these hop numbers into latencies with some variation (as the actual latencies undoubtedly have), we use the formula  $Latency = \lfloor 50 \cdot (hops + \Delta) \rfloor ms$ , where  $\Delta$  is a random value between  $-0.5$  and  $+0.5$ .

### 3.2 Performance Metrics

As we often compare algorithms with different cost functions and constraints, a comparison of minimized cost function values would often not be meaningful. Instead, we compare RPAs on what their impact is on the perceived performance of the CDN system itself. In this paper, we use two system performance metrics: one based on the average client-perceived latency, and the other one is a QoS-metric based on a client-perceived latency threshold. The latter metric is useful due to the fact that for example, service level agreements (SLA) used

in CDNs might be in the form *the  $X$ th percentile of requests have a response latency below  $Y$  msec*. Performance above some threshold is unacceptable by the users. In this study, instead of fixing this threshold to one number, we display graphs from which the result  $X$  of any threshold  $Y$  can be determined. This way the reader can choose any threshold deemed interesting. The evaluation in Section 4 refers to the Cumulative Distribution Function (CDF) of the client-perceived latency, given the produced placements. The larger this percentile for a given threshold, the better the algorithm is for this system and workload.

### 3.3 Algorithm Implementations

The placements of all the algorithms in this study were produced by *Coeus* [10]. This tool takes the problem definitions and heuristic primitives from Section 2.2 and produces the placement for that algorithm. In this way, it can run any previously published CDN algorithm and many variations thereof. We decided not to run any new algorithms, because we believe that the point of diminishing returns have already been reached for RPAs when used in the context of today’s CDNs. *Coeus* uses existing algorithms and/or implementations of them when available. The placements produced by *Coeus* have been validated against existing implementations of algorithms when possible. All algorithms in this paper make decisions about placements using pre-

Heuristic	Computation	No. Messages	Message Size
Greedy Global	$O(RCNK)$	$O(N)$	$O(NK + CK)$
Ranking Local	$O(CK)$	0	0
Popularity	$O(CNK)$	$O(N)$	$O(CK)$
Ranking Dist	$O(CN)$	1	$O(CN)$
Hotspot	$O(CNK)$	$O(N)$	$O(CK)$
Fan-Out	$O(N)$	1	$O(N)$
Swap	$O(I_{swap}CNK)$	$O(N)$	$O(NK + CK)$
Lagrangian	$O(I_{lagr}RCNK)$	$O(N)$	$O(NK + CK)$
LRU Caching	$O(1)$	0	0

Table 3: The various decision costs of the algorithms evaluated in this paper.  $C$  is the number of clients,  $N$  the number of nodes,  $K$  the number of objects, and  $R$  is the number of object replicas allocated every time the algorithm is executed.  $I_{swap}$  and  $I_{lagr}$  are the number of iterations Swap and Lagrangian relaxation are run for.

viously observed data, i.e. they do not know the future. The RPAs are run once every day unless otherwise specified, and we assume that they produce their placements instantaneously. The latter assumption has little impact on the results as the workload we study is quite stable.

Here are some details about the specific parameters of algorithms. The vicinity of the Hotspot algorithm is defined to be any client within 50 ms. Swap iterates for 5000 iterations and only generates feasible solutions. Lagrangian relaxation iterates for 500 iterations.

The decision costs of the algorithms are shown in Table 3. If the cost function is `dist` or `maxdist`,  $K = 1$ , as all objects are placed in the same way and only the placement of one object has to be calculated. We assume that the distance matrix and the fan-out vector change infrequently. Thus this information only needs to be fetched once. This can be seen in Table 3 for Ranking Dist and Fan-Out. The computation cost of Lagrangian relaxation is only valid for the `so_readdist` problem formulation. To get the decision cost of a combined heuristic such as Greedy+, add the computation costs of the two heuristics and form the union of the number of messages cost and the union of the message sizes.

## 4 Experimental Results

In this Section, we show the results for the World-Cup98 workload. We start by identifying the best RPA without any infrastructure constraints in Section 4.1. In Section 4.2, we examine RPAs with storage constraints to identify the best ones. The best algorithms are then compared against caching in Section 4.3. In Section 4.4, we show that by simply evaluating the caching algorithm less frequently, caching outperforms even the best RPAs for CDNs.

### 4.1 Infinite Capacity

In this Section we study the performance of RPAs on a system with infinite capacity. The algorithms in this case can allocate as many objects to a node as they want and are only constrained by a number of replica constraint. We only report the results for 16 replicas for each object, as the results are similar for values between 4 and 64 replicas. We examine all the heuristics from Table 2 that can take a  $P$  constraint applied to the problem formulations `so_readdist`, `dist`, and `maxdist` from Table 1. The reason that `mo_readdist` is not included is that it produces a placement identical to `so_readdist` for a  $P$  constraint as there are no constraints or dependencies between objects.

Figure 1 lists the average client-perceived latencies for various heuristics applied to the

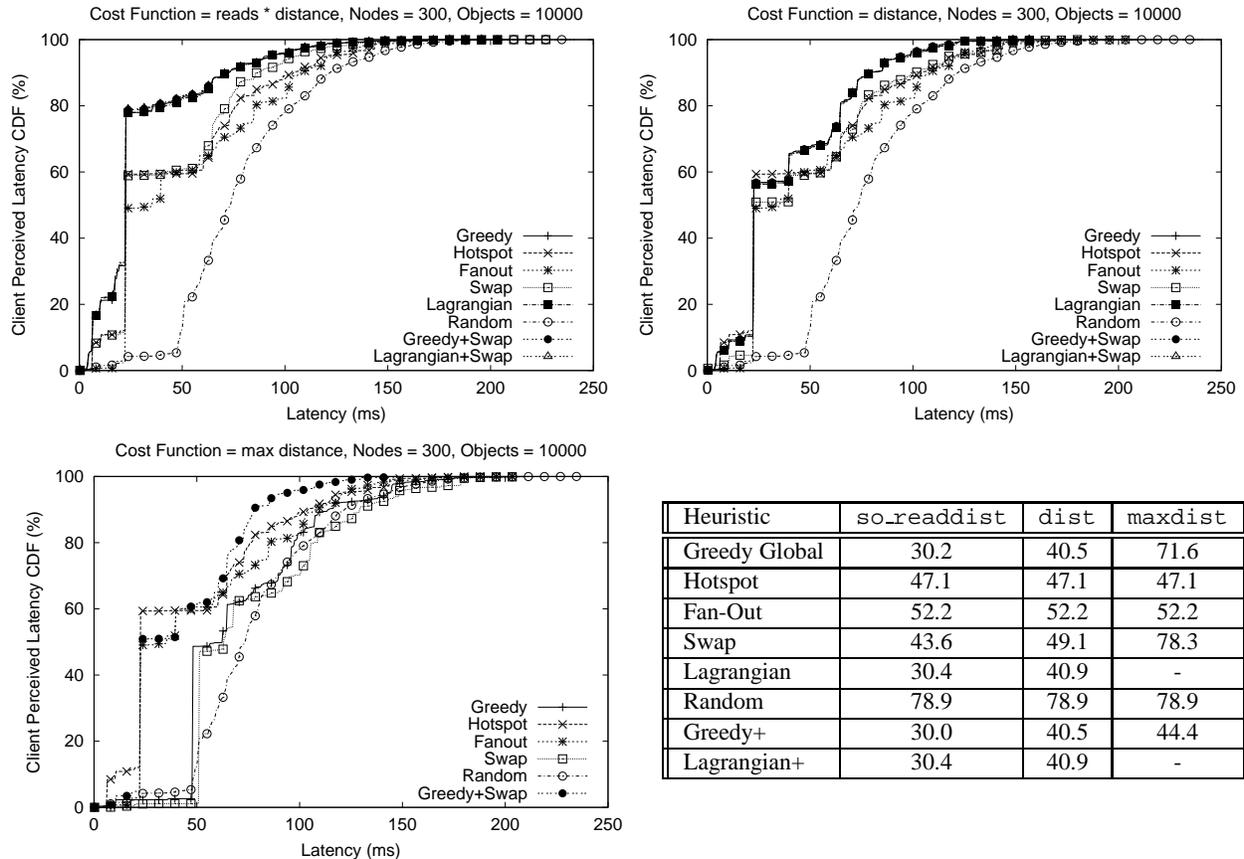


Figure 1: The graphs show the client-perceived latencies for various algorithms when the system has no infrastructure constraints. The table shows the average client-perceived latencies (in ms.) for the same system. We do not have an implementation of Lagrangian relaxation for `maxdist`.

`so_readdist`, `dist`, and `maxdist` problem definitions, and the results of the threshold-based QoS-metric. If we start by comparing problem definitions against each other, we see that `so_readdist` provides better results than `dist` which is better than `maxdist` for both the average latency and the threshold-based metric. This is because some clients generate far more accesses than others. When the threshold is above approximately 150 ms, the cost function does not matter much. But for thresholds under 150 ms and for average latency values, `so_readdist` is the preferred cost function. Thus, for the rest of the paper, we will not show any further results for `dist` and `maxdist`.

Focusing only on `so_readdist`, we would like to know what heuristic provides the best results. The first observation to make is that adding a Swap

heuristic to Greedy and Lagrangian only improves the results marginally, if at all, from plain Greedy and Lagrangian. Note that this can be hard to see in the graph, as the results for Greedy, Greedy+, Lagrangian and Lagrangian+ nearly completely overlap each other. The table with the average latencies shows the small differences between them more clearly. Among the single heuristics, Greedy and Lagrangian produce the best placements over all the performance metrics. Greedy is preferred, as it has the lowest decision cost of the two according to Table 3.

## 4.2 Finite Storage Capacity

In this Section, we compare RPAs with and without storage constraints. This corresponds to

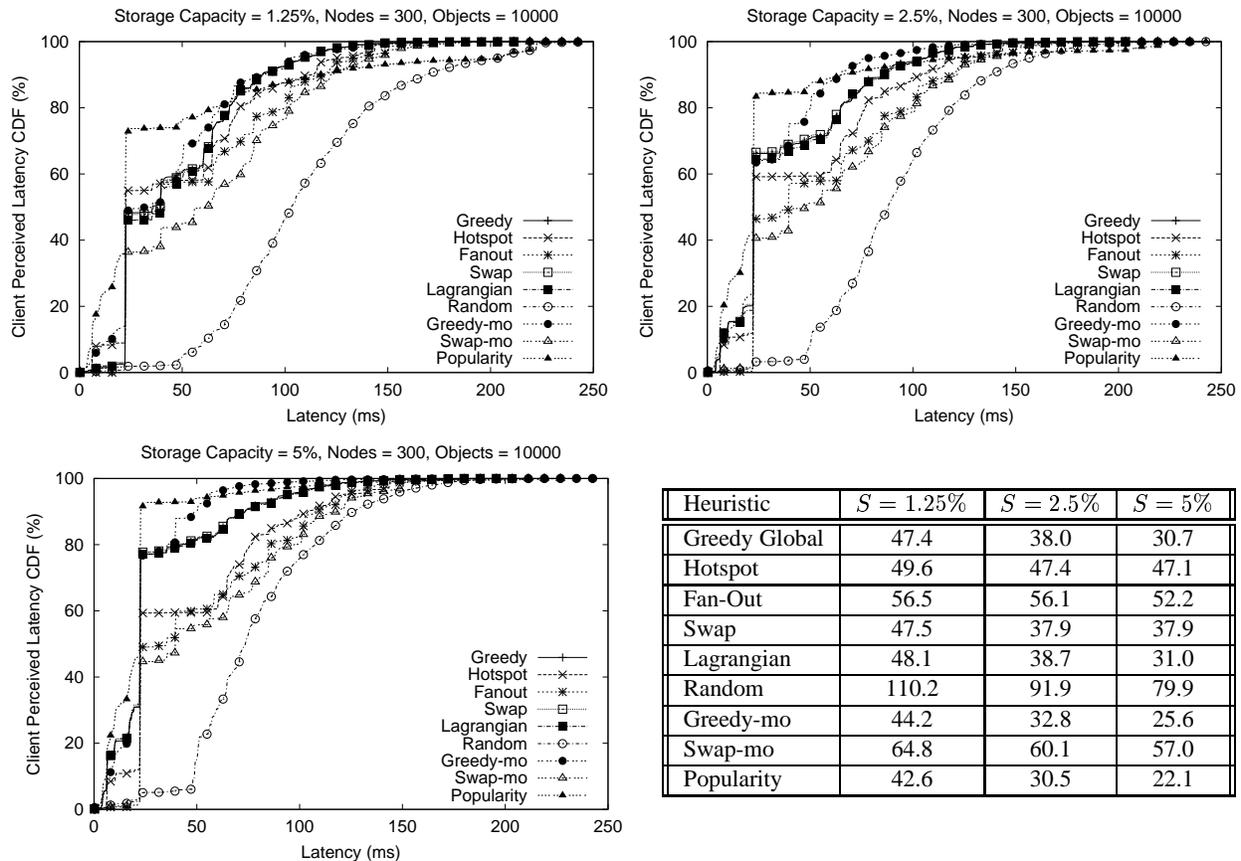


Figure 2: The client-perceived latencies for various algorithms when the system has storage constraints. The table shows the average client perceived latencies in ms.

`so_readdist` with a number of replica constraint and `mo_readdist` with a storage capacity constraint, respectively. In using an `so_readdist` algorithm in a system with storage constraints, we assume the following. Each object in a `so_readdist` algorithm has a replication factor equal to  $\lceil N \cdot S/K \rceil$  ( $S$  being the storage capacity of each node). This assures that all RPAs allocate roughly the same amount of object replicas. When a `so_readdist` algorithm has decided on a placement and there are more objects allocated to a node than there is storage capacity, only the most frequently accessed objects on that node will be allocated up to the storage capacity constraint. The rest are dropped.

Figure 2 shows the average client-perceived latency and the threshold-based results for nodes with storage capacities of 1.25%, 2.5% and 5% of all

the objects in the system. Here, *Greedy-mo* and *Swap-mo* are the Greedy and Swap heuristics for `mo_readdist`, i.e. they do take storage constraints into consideration. Popularity is a local algorithm that also takes storage constraints into consideration. We will not show any results for Ranking Local since it provides similar results as Popularity [9].

Starting with the system that can store 5% of the objects, we can see from Figure 2 that overall the decentralized Popularity algorithm performs the best followed by Greedy-mo. This is true both for the average metric and the threshold-based one. For  $S = 1.25\%$  and  $S = 2.5\%$ , Popularity is not as good as a number of other algorithms when the threshold is between 75 ms and 225 ms. This is because Popularity only optimizes accesses that hit locally, thus the step around 25 ms. The

`so_readdist` algorithms (Greedy, Hotspot, Fan-out, Swap, and Lagrangian) perform significantly worse than Popularity up until you get to approximately  $S = 40\%$  (not shown), when it does not matter much what algorithm you choose, since every node can store 40% of the objects. The placements of swap-mo can be significantly improved by executing more iterations. However, the decision cost will be far higher than for Greedy-mo.

In summary: there does not seem to be any reason to use the best `so_readdist` algorithms as they are outperformed by Popularity and Greedy-mo even when there is plenty of storage space. Popularity is the best algorithm when considering average latencies and for thresholds under approximately 75 ms. What is even better is that it is the fastest if we exclude Fan-Out that does not perform well. In the next section, the two best RPAs, Popularity and Greedy-mo, are compared to caching.

### 4.3 Comparison against Caching

Generally, it is assumed that since an RPA is evaluated as infrequently as e.g., once a day, each node knows where the closest replica of each object it is not storing is located. This information could probably be dispersed in the system at little cost since it would be done infrequently. In this section, we will assume that this information can be dispersed without any cost. On the other hand, for local caching that potentially changes its placement after each single access, this is prohibitively expensive. Thus, we will assume that a node using a caching algorithm will not know where the closest copy of an object it is not storing is located. At a miss, it will have to go to some origin node to fetch the object. We assume that this origin node is located 200 ms away in the network. This 200 ms latency can be seen as a knee in Figure 3, and the results for another origin node latency could easily be estimated by moving this knee. For comparison reasons, we also show the results for an Cooperative caching scheme [21] using plain, local LRU that knows at no cost where the closest replica of each object is. This scheme is denoted “optimal LRU cooperative caching” in the figure.

Figure 3 shows the results for Caching, Greedy-mo and Popularity. The point to make from these graphs is that regular local caching does not perform well at all if looking at averages (not shown in the Figure) and for latency thresholds above 50 ms. This is because caching has to pay the heavy penalty of going to the origin server as it does not have any knowledge of where the closest replica is. For thresholds under 50 ms, it produces placements nearly as good as the best RPA Popularity. As expected, as the storage capacity increases the difference between caching and the two RPAs disappears. This means that if we believe that storage is going to be ample, caching will be just as good as the best RPAs.

Could we now make caching perform well even for low storage constraints? That is what we will examine in the next section.

### 4.4 Impact of Evaluation Interval

A lot of research has gone into cooperative caching and the field has been at the point of diminishing returns for quite a while. These schemes all tried to reach the performance of the optimal LRU cooperative caching curve in Figure 3 by evaluating the caching algorithm after each single access and at regular time intervals communicating information regarding the location of the nearest replicas. What if we instead take our regular LRU caching algorithm and run it infrequently and only communicate closest replica information at these evaluation points? Thus, in between these evaluation points there will be no changes to the cached contents. In Section 2.2, we identified that LRU caching is in essence a completely decentralized RPA, run after each access that uses ranking of access times as its approximation method and has a storage constraint. If we take this RPA (which, if it is run after each single access is called LRU caching) and run it e.g., once a day, it should be possible to communicate the locations of the closest copy of an object to the nodes, just as it is possible for the other RPAs. The overhead and amount of information sent is going to be the same as for any other RPA. Thus, we gain the information of where the closest replica of an object is located at the cost of only being able to

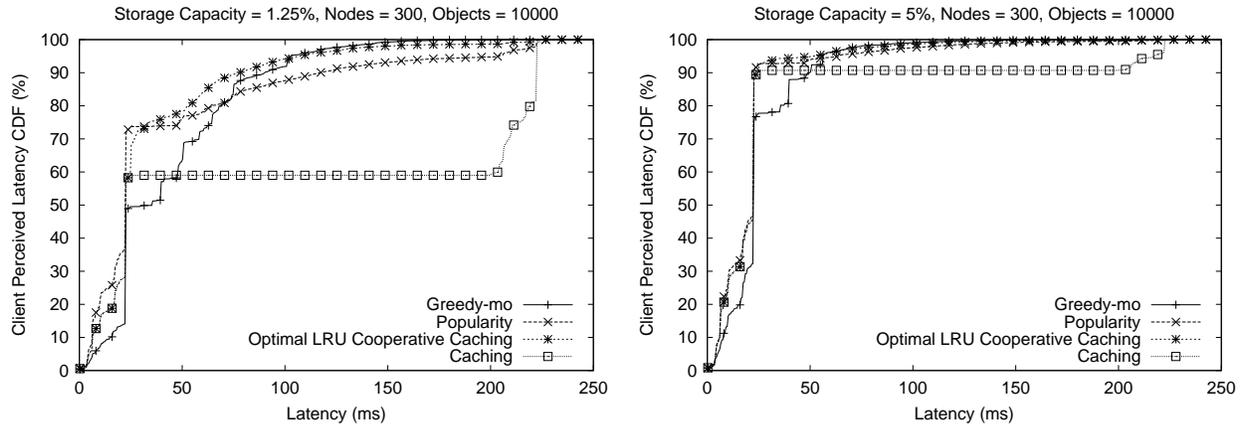


Figure 3: Comparison of caching, Greedy-mo, Popularity and optimal LRU cooperative caching for various storage constraints.

evaluate and change the contents of the node infrequently. We are going to refer to this algorithm as *delayed caching*.

Figure 4 shows the results for Popularity and LRU caching when the evaluation interval is varied between 1 access, for caching, to 10,000 seconds (roughly 3 hours). As can be seen from the Figure, delayed caching is better or as good as Popularity when looking at averages (not shown in the Figure). For the threshold based metric, delayed caching is worse when the threshold is lower than 50 ms, but better above that threshold. The reason for this is that delayed caching stores some infrequently accessed objects because it only looks at the access time. This is bad from a local perspective, but good from a global perspective as some infrequently accessed objects will be spread around the system. As we would expect, the greater the storage capacity, the less the difference between the two schemes. When comparing caching and Popularity, where both have no knowledge of where the closest replica is (Caching and Popularity- in the graph), Popularity performs better.

To conclude this section; with simple caching we can achieve, on the average, a better or as good a placement as the best RPAs if we just increase the evaluation interval from the usual one access to hours or even days. By doing this, it is possible to use the same mechanisms that RPAs use to infrequently disseminate information about the location

of other objects. Thus, there seem to be no need to use an RPA under the conditions and in the system we have considered, as delayed caching is just as good.

## 5 Discussion and Open Questions

While we showed that there is little use of current RPAs in today’s CDNs, as caching is as good if not better, we do believe that there is a need for RPAs in future CDNs. When a CDN has to provide some level of performance, reliability or availability guarantees, there will be a need for RPAs to achieve this goal, as regular caching cannot provide any useful guarantees for these metrics. However, as far as we know, there has been little work within the field of CDNs towards this.

Moreover, RPAs might be useful when CDNs allow users and content providers to write to objects. There are several algorithms for limiting the performance penalty for writes to caches, such as write-invalidate and write-update. However, it is unclear how well they would work in a CDN. If CDNs start to provide update guarantees in the form of “all your replicas of the object will be consistent within  $x$  seconds of a write”, RPAs should be useful as caching would not be able to provide such guarantees. But, to the best of our knowledge, there are no CDN RPAs out there that provide such guarantees.

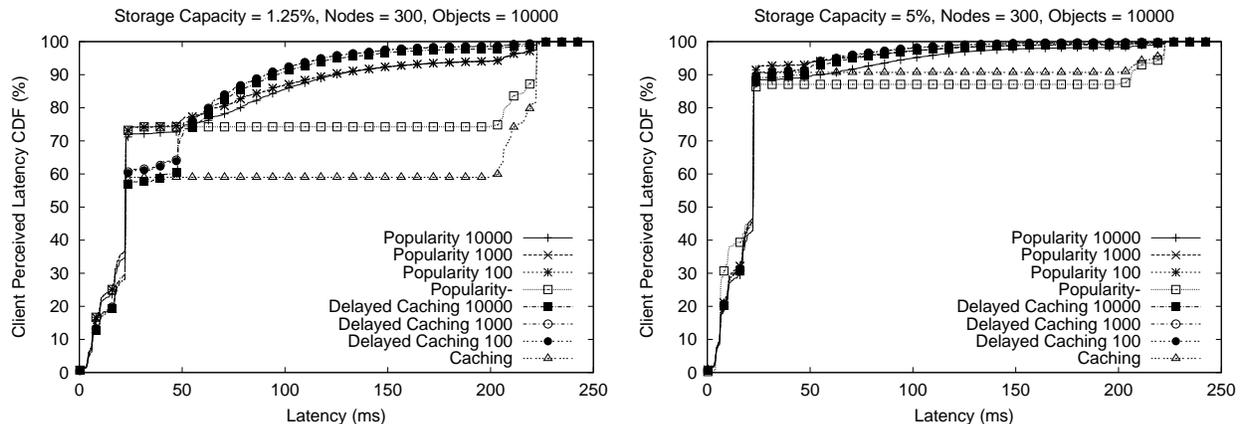


Figure 4: Comparison of caching and Popularity for various evaluation intervals (in seconds). Caching without a number after it is regular caching evaluated after each single access. Popularity- is the Popularity algorithm without any knowledge of the location of closest replicas.

A final open question is if our results are still valid on a system with nodal bandwidth or load constraints, or a system with network link capacity constraints. There are some algorithms that consider this [20, 14], but it is unclear how much of a performance benefit, if any, compared to caching they provide.

## 6 Conclusions

In this paper, we compare the benefits of caching with replica placement algorithms (RPA) when used in content delivery networks (CDN). First, we identify what RPAs provide the best client-perceived latency improvements. We find that taking object accesses into account usually makes a big difference. We find that the decentralized Popularity algorithm [9] and the storage constrained, multi-object Greedy algorithm [9] overall work the best across a number of performance metrics. Popularity is generally preferable as it is much less computationally expensive.

When we compare caching to Popularity and the above Greedy algorithm, we find that when there is plenty of storage, caching is as good, if not better, than the best RPA. However, caching performs much worse than the RPAs when there is a limited amount of storage. This is mainly due to the

fact that RPAs are only evaluated infrequently, e.g., once a day, and thus it is possible to communicate information about where a node can fetch from the closest replica of an object. As caching changes its stored content frequently, this is not possible for a local cache, and it has to go to the origin server that is usually located at quite some distance. However, by only evaluating the caching algorithm as infrequently as the RPAs, we can communicate the knowledge about the closest copy of an object in the same way as with an RPA. In this way, simple caching generally performs better than an RPA even for storage constrained systems.

While we conclude this paper by stating that there seems to be no need for RPAs in the CDN we studied, we believe that they will be useful, once CDNs start to provide consistency, performance, availability, and reliability guarantees. However, for CDNs little research has been done in that space. It is also an open question how our results extends to bandwidth and load constrained CDNs.

## Acknowledgments

The authors would like to thank Christos Karamanolis, Yasushi Saito and the anonymous reviewers for greatly improving the quality of this paper.

## References

- [1] Akamai. Cambridge, MA, USA. <http://www.akamai.com>.
- [2] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. Technical Report HPL-1999-35R1, HP Laboratories, 1999.
- [3] J. Current, M. Daskin, and D. Schilling. Discrete Network Location Models. In Z. Drezner and H. Hamacher, editors, *Facility Location Theory: Applications and Methods*, 2001. Forthcoming.
- [4] Digital Island. <http://www.digitalisland.com>.
- [5] L. W. Dowdy and D. V. Foster. Comparative Models of the File Assignment Problem. *ACM Computer Surveys*, 14(2):287–313, 1982.
- [6] S. Hakimi. Optimum Location of Switching Centers and the Absolute Centers and Medians of a Graph. *Operations Research*, 12:450–459, 1964.
- [7] S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. On the Placement of Internet Instrumentation. In *Proceedings of IEEE INFOCOM*, pages 295–304, March 2000.
- [8] S. Jamin, C. Jiu, A. Kurc, D. Raz, and Y. Shavitt. Constrained Mirror Placement on the Internet. In *Proceedings of IEEE INFOCOM*, pages 31–40, April 2001.
- [9] J. Kangasharju, J. Roberts, and K. Ross. Object Replication Strategies in Content Distribution Networks. *Computer Communications*, 25(4):367–383, March 2002.
- [10] M. Karlsson, C. Karamanolis, and M. Mahalingam. A Framework for Evaluating Replica Placement Algorithms. Technical Report HPL-2002, HP Laboratories, July 2002. [http://www.hpl.hp.com/personal/Magnus\\_Karlsson](http://www.hpl.hp.com/personal/Magnus_Karlsson).
- [11] M. Korupolu, G. Plaxton, and R. Rajaraman. Placement Algorithms for Hierarchical Cooperative Caching. *Journal of Algorithms*, 38(1):260–302, January 2001.
- [12] A. Leff, J. Wolf, and P. Yu. Replication Algorithms in a Remote Caching Architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1185–1204, November 1993.
- [13] B. Li, M. Golin, G. Italiano, and X. Deng. The Optimal Placement of Web Proxies in the Internet. In *Proceedings of IEEE INFOCOM*, pages 1282–1290, March 1999.
- [14] S. Mahmoud and J. Riordon. Optimal Allocation of Resources in Distributed Information Networks. *ACM Transactions on Database Systems*, 1(1):66–78, March 1976.
- [15] K. Obraczka and F. Silvia. Network Latency Metrics for Server Proximity. In *Proceedings of the IEEE Globecom*, November 2000.
- [16] L. Qiu, V. Padmanabhan, and G. Voelker. On the Placement of Web Server Replicas. In *Proceedings of IEEE INFOCOM*, pages 1587–1596, April 2001.
- [17] P. Radoslavov, R. Govindan, and D. Estrin. Topology-Informed Internet Replica Placement. *Computer Communications*, 25(4):384–392, March 2002.
- [18] Telestra.net. <http://www.telestra.net>.
- [19] V. Vazirani. *Approximation Algorithms*. ISBN: 3-540-65367-8. Springer-Verlag, 2001.
- [20] A. Venkataramanj, P. Weidmann, and M. Dahlin. Bandwidth Constrained Placement in a WAN. In *ACM Symposium on Principles of Distributed Computing (PODC'01)*, August 2001.
- [21] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 16–31, December 1999.