Choosing Replica Placement Heuristics for Wide-Area Systems

Magnus Karlsson and Christos Karamanolis
Storage Systems Department, HP Laboratories
Palo Alto, CA

Abstract

Data replication is used extensively in wide-area distributed systems to achieve low data-access latency. A large number of heuristics have been proposed to perform replica placement. Practical experience indicates that the choice of heuristic makes a big difference in terms of the cost of required infrastructure (e.g., storage capacity and network bandwidth), depending on system topology, workload and performance goals.

This paper describes a method to assist system designers choose placement heuristics that meet their performance goals for the lowest possible infrastructure cost. Existing heuristics are classified according to a number of properties. The inherent cost (lower bound) for each class of heuristics is obtained for given system, workload and performance goals. The system designer compares different classes of heuristics on the basis of these lower bounds. Experimental results show that choosing a heuristic with the proposed methodology results in up to 7 times lower cost compared to using an "obvious" heuristic, such as caching.

1. Introduction

Data replication is used extensively to improve data access performance in wide-area distributed systems. In systems such as content delivery networks [4], web hosting services [11, 12], and distributed data repositories [8, 13], data are replicated close to the point of access to achieve low access latency.

Several techniques are used to perform replica placement, including variations of caching [7] and centralized algorithms [11]. All approaches aim at meeting some latency-related performance goal; some take such a goal as an input parameter, while others provide an implicit performance level that cannot be specified. Similarly, the metric used to capture the performance goal differs among different approaches. For example, some systems aim at improving the average access latency, while others guarantee that at least a certain percentage of accesses are served within a specified latency threshold.

Consider a storage system (e.g., a video on-demand service) storing 10k unique files, 1TB in total size, over 300 distributed nodes. Assume that the designer wants 90% of read requests to be served within at most 100ms. Achieving this goal by using LRU caching [14] would require at least $30,000 just for disk capacity for the caches. On the other hand, if instead a centralized greedy placement heuristic [11] is used for the same system and workload, the performance goal could be met with just $7,500 for disk capacity, with both approaches having a comparable cost for consumed network bandwidth [6].

The example above illustrates that choosing the right replica placement approach is a non-trivial and non-intuitive exercise. Deciding how many replicas to create and where to place them to meet a performance goal with minimal infrastructure cost is an NP-hard problem (see Appendix) . All replica placement approaches proposed in the literature are heuristics that are designed for certain systems and workloads. A heuristic that works well in one case may be completely inappropriate under different circumstances—it may be too costly or even unable to meet the performance goal.

One way to choose a heuristic is, of course, by implementing a number of them in a live system and evaluating them under real workloads. In the general case, this is not feasible due to the performance and cost implications of poor choices. Moreover, implementing and integrating a heuristic in a live system is a challenging and time consuming task. Even worse, this process may need to be repeated every time the system or workload change.

This paper proposes a method for deciding on the applicability of replica placement heuristics, without affecting the live system. The key idea is to derive the inherent cost of different classes of heuristics. To achieve this, we use an integer programming (IP) formulation of the replica placement problem, together with constraints that capture the possible placement solutions of different heuristics. This formulation is used to calculate lower bounds for the resulting cost of a range of possible heuristics, given specific system topologies, workloads and performance goals. The paper presents experimental results demonstrating that the cost of actual heuristics is close to that of the obtained lower bounds. Thus, the key idea is that lower bounds can be used to argue about the applicability of heuristics for a given workload, system and performance goal.

The focus of the paper is on the practical aspects of the method, not on its theoretical foundations. The latter are described in the appendix . The proposed method is an off-line approach that assists system designers in the following ways: 1) eliminate types of replica placement heuristics that can *not* meet the required performance goal for a certain system topology and workload; 2) among heuristics that can meet the goal, pick one that results in low infrastructure cost; 3) decide on the necessary system configuration (number of nodes, storage capacities, etc).

We demonstrate the applicability of the method with experimental results from a realistic case study—a service that provides file access to the regional and remote offices of a corporation with multiple sites. The evaluation shows that the method identifies the appropriate heuristic and results in considerable savings in infrastructure costs.

In summary, the paper makes the following contributions:

- Proposes an off-line method for choosing a replica placement heuristic, given a system, workload and performance goal.
- Evaluates the applicability of heuristics on the basis of lower bounds on cost, without requiring to deploy the heuristics in a live system.
- Demonstrates the practicality of the proposed method in a real-world case study.

2. Related Work

Data replication and placement for improving performance was first studied in the context of the *file assignment problem* [3] and was shown to be a complex combinatorial optimization problem. Replica placement has been a key technology in content delivery networks [4] as well as web caching and web proxy services [7, 12, 18]. All these systems aim at improving the average access latency, while they minimize the cost of the system resources used for replication. Qiu *et al.* [11] defined the problem of replica placement that minimizes the average access latency for given system resources. They formalized it as a *static k*-median problem for which they obtained general lower bounds; the cost of some known heuristics was compared against those lower bounds. Our goal is to *guarantee* certain latency goals while minimizing the replication cost. We obtain lower bounds for specific classes of heuristics. These bounds provide stronger comparison points for the cost of practical heuristics.

More recently, Yu and Vahdat proposed and formalized the problem of minimal replication cost for a given availability target [20]. They derive general lower bounds, given a workload, network partition patterns and consistency requirements, in a system with *one* data object. In their work, availability is defined as the fraction of accesses that complete successfully. We are also concerned with minimizing the cost of resources used to do replication. However, our objective is to meet a specified latency goal. We obtain lower bounds for certain classes of heuristics and for realistic number of data objects. One of our definitions of performance goals is a generalization of theirs (they require that 100% of the requests satisfy the latency threshold). To avoid an explosion to the computational complexity of the problem and to handle large number of objects, we do not capture failures and consistency.

We have shown that the replica placement problem is a dynamic variation of the classic SET-COVER problem [5, 16]. It is dynamic because access patterns and, as a result, placement of replicas may change over time. There are dynamic extensions of the original static SET-COVER that have been proposed in the literature [9], but none of them captures latency goals. Daskin and Owen have proposed an extension that captures latency goals, but only in a static context [10]. There are no known approximation algorithms for either extension of SET-COVER. Thus, our method uses a linear programming relaxation of the IP problem formulation to obtain lower bounds that, in practice, are close-to-tight.

3. Problem Formulation

The proposed method compares classes of heuristics on the basis of lower cost bounds, given a system, workload and performance goal. To that purpose, we formalize the *minimal replication cost for performance* (MC-PERF) problem as an Integer Programming (IP) optimization problem. We solve this problem to derive the required lower bounds.

3.1. Basic Problem

In the problem formulation, the system is represented as a set of interconnected nodes (N). The nodes store replicas of a set of data objects (K). Each node has some demand for different objects in the system, captured as requests originating from the users on that node.

Replicas of objects can be dynamically created or removed on any of the nodes. Such changes may only occur at the beginning of *evaluation intervals.* The intervals represent the highest frequency at which the placement heuristic is executed on any node in the system. For example, a caching heuristic is run on a node upon every single object access initiated at that node; a complex centralized place-

Variable	Meaning
$\mathbf{store_{nik}}$	Binary: node n stores object k during interval i.
$\mathbf{create_{nik}}$	Binary: object k is created on node n during interval i.
$\mathbf{covered_{nik}}$	Binary: node n can access object k within the latency threshold during interval i.
$\mathbf{route_{nmik}}$	Binary: node n can access object k in interval i from node m.
$\mathbf{open_{n}}$	Binary: node n is enabled, i.e., it can store replicas.
$read_{nik}$	The number of read requests from node n to object k in interval i.
$write_{nik}$	The number of write requests from node n to object k in interval i.
$latency_{nm}$	The latency for accessing an object at node m from node n.
$dist_{nm}$	Binary: node n can reach node m within the latency threshold T_{lat}.
$know_{nm}$	Binary: node n uses information from node m to make its placement decision.
$fetch_{nm}$	Binary: node n can fetch objects from node m.
$hist_{nik}$	Binary: node n has a record in its activity history of object k being accessed on node n in interval i.
T_{lat}	The target latency threshold.
T_{qos}	The fraction of accesses served in at most T_{lat}, in the case of the first metric.
T_{avg}	The average latency goal for the second metric.
α	The unit cost for storage.
β	The unit cost for replica creation.
γ	The unit cost of the penalty for an access taking more than T_{lat}.
δ	The unit cost for an update message.
ζ	The unit cost for enabling and managing a node.

Table 1. The variables used in the IP model of MC-PERF. Variables in bold are the unknown decision variables. n and m are nodes, i is an interval and k is an object.

ment heuristic may be run once a day. An execution in the system consists of a finite sequence of evaluation intervals (I).

We extend the definition of *replication cost* by Yu and Vahdat [20] to cover *multiple objects*, as captured by cost function (1). The replication cost is defined to be the sum of the *storage cost* and the *replica creation cost* (the cost of networking resources used to create a replica), over all objects, nodes and intervals. We do not consider replica removal cost, as it is negligible in most real systems.

In the cost function, $store_{nik}$ is a binary variable capturing whether node n stores object k during interval i; $create_{nik}$ denotes whether object k is created on node n at the beginning of interval i. Constants α and β represent the unit cost for storage and replica creation cost, respectively. They also provide a way to change the weight of each of the two elements in the cost function. The definition could be easily extended to let α and β vary for different objects and nodes. The variables used in the problem definition are summarized in Table 3. For notational convenience, we write $\forall n,m$ instead of $\forall n,m \in N$, $\forall i$ instead of $\forall i \in I$, and $\forall k$ instead of $\forall k \in K$.

$$\sum_{i\in I}\sum_{n\in N}\sum_{k\in K}(\alpha\cdot store_{nik}+\beta\cdot create_{nik}) \tag{1}$$

The objective of MC-PERF is to minimize (1), subject to a number of constraints, that capture the performance goal metric that is of interest. Other metrics can be expressed, but they have to be monotonically increasing or monotonically decreasing with the addition of a replica. In this paper, we focus on two representative metrics that we consider of practical importance.

The first metric is a utility function that specifies the percentage of requests that have to be satisfied within a latency threshold (referred to as *QoS goal* in the rest of the paper). It is captured in (2) as the minimum required request ratio (T_{qos}) served within the latency threshold; e.g., 99% of the requests are served within 250 ms. This performance goal can be defined for a single user or for an entire group of users (e.g., department), as well as for a single data object or for a set of objects (e.g. all the files of a user). Constraint (2) defines this on a per user basis and over all objects; it can be easily modified to account for any of the other three possible definitions.

Constraints (3) – (6) define the problem variables and system invariants. Constraint (4) states that there are no replicas in the system to start with; however, this could be trivially modified to account for any initial placement. $dist_{nm}$ is a binary variable capturing whether node n can reach node m within the latency threshold, given the system's topology.

$$\frac{\sum_{i\in I}\sum_{k\in K} read_{nik}\cdot covered_{nik}}{\sum_{i\in I}\sum_{k\in K} read_{nik}} \geq T_{qos} \quad \forall n \tag{2}$$

$$create_{nik} \geq store_{nik} - store_{n,i-1,k} \quad \forall n,i,k \tag{3}$$

$$store_{n,-1,k} = 0 \quad \forall n,k \tag{4}$$

$$covered_{nik} \leq \sum_{m\in N} dist_{nm}\cdot store_{mik} \quad \forall n,i,k \tag{5}$$

$$store_{nik}, covered_{nik}, create_{nik} \in \{0,1\} \quad \forall n,i,k \tag{6}$$

The second metric refers to the average latency perceived by the users. It states, for example, that the average latency must not be above 250 ms. To express this type of performance goal, constraint (2) is replaced by equations (7) –

(10) below. In particular, constraint (7) states that the average perceived latency in the system must be less or equal to a specified threshold. The other constraints capture the fact that a request is routed only to one node that stores the referenced object.

$$\frac{\sum_{m\in N}\sum_{i\in I}\sum_{k\in K} read_{nik}\cdot latency_{nm}\cdot route_{nmik}}{\sum_{i\in I}\sum_{k\in K} read_{nik}} \le T_{avg} \quad \forall n \quad (7)$$

$$\sum_{m\in N} route_{nmik} = 1 \qquad \forall n,i,k \qquad (8)$$
$$route_{nmik} - store_{nik} \le 0 \quad \forall n,m,i,k \qquad (9)$$
$$route_{nmik} \in \{0,1\} \quad \forall n,m,i,k \qquad (10)$$

We have shown that problem MC-PERF, as defined here, is NP-hard. The proof is based on a reduction of the well-known SET-COVER problem [17] to MC-PERF. The details of the proof are outside the scope of this paper; they can be found in the appendix .

3.2. Model Extensions

The problem definition as stated above does not make any provisions for those requests that do not meet the performance goal, for any of the two metrics. For example, if 99% of the requests are served within 250 ms, the remaining 1% may take arbitrarily long to be served (or can even be completely discarded). Often, it makes sense to serve the requests that miss the performance goal in a best effort manner. To capture this, term (11) can be added to the cost function (1). This term introduces a penalty (additional cost proportional to γ) for any access that takes longer than the latency threshold T_{lat}.

$$\gamma\cdot\sum_{i\in I}\sum_{n\in N}\sum_{k\in K}(read_{nik}\cdot(1-covered_{nik})\cdot \sum_{m\in N}(latency_{nm}-T_{lat})\cdot route_{nmik}) \qquad (11)$$

Similarly, term (12) can be added to the cost function to capture the cost of writes (updates) in the system. The basic definition, implies that data replicas can be stored on any node in the system. However, enabling a node to run the placement heuristic and store replicas may involve some cost. This cost is captured by including term (13) in the cost function. In that case, constraint (14) states that placement can be performed only on "open" nodes.

$$\delta\cdot\sum_{i\in I}\sum_{n\in N}\sum_{k\in K}(write_{nik}\cdot\sum_{m\in N} store_{mik}) \qquad (12)$$
$$\zeta\cdot\sum_{n\in N} open_n \qquad (13)$$
$$open_n \ge store_{nik} \quad \forall n,i,k \qquad (14)$$
$$open_n \in \{0,1\} \qquad \forall n \qquad (15)$$

4. Classes of Heuristics

Solving the above IP problem, we can get the theoretically lowest possible infrastructure cost. This is the general lower bound for MC-PERF. However, for the purposes of this paper, we need to obtain the lowest possible cost achievable with different types of heuristics. To accomplish this, we identify a set of properties that capture the techniques and assumptions found in different heuristics. These properties are expressed as additional constraints in the IP formulation. Solving MC-PERF with some additional constraints that represent a certain class of heuristics, we get the lowest possible cost of those heuristics, for a given system, workload and performance goal.

4.1. Heuristic Properties

The heuristic properties we consider are listed in Table 2. As will be discussed in Section 4.2, they are representative of an extensive range of placement heuristics. The properties are discussed in more detail in the following paragraphs.

Storage constraint. Placement heuristics that use a fixed amount of storage throughout the execution satisfy the storage constraint property. Examples include caching heuristics and many file allocation algorithms [3]. There are two variations of this property that are captured by the two versions of constraint (16). The first states that the amount of storage used is the same for every single node and interval. This corresponds, e.g., to a system where the administrator configures every node with the same cache size. The second variation reflects the case where the amount of storage used varies between nodes (but not with time). This captures systems where e.g., larger caches are placed on strategic nodes with high traffic. The constraint captures just the fact that the storage size is the same across all intervals (and across all nodes for the first variation); it does *not* specify what that capacity is. It is the solution to the problem that provides the minimal capacity that satisfies this constraint.

$$\sum_{k\in K} store_{nik} = \sum_{k\in K} store_{0,0,k} \quad \forall n,i \qquad (16)$$

$$\sum_{k\in K} store_{nik} = \sum_{k\in K} store_{n,0,k} \quad \forall n,i \qquad (16a)$$

Replica constraint. This constraint reflects heuristics (typically centralized [3, 11]) that use a fixed number of replicas for each object throughout the execution. There are two variations of this constraint. The first states that *all* objects have the same number of replicas in the system, for all intervals in the execution. It refers to heuristics that create a fixed number of replicas for all objects irrespective of demand. The second variation states that each object has its own replication factor for the duration of the execution. This

Heuristic properties	Abbrev	Lower bound for any heuristic...
storage constraint	SC	that is using a fixed amount of storage in every interval.
replica constraint	RC	that is using a fixed number of object replicas in every interval.
routing knowledge	Route	where each node knows the contents of some (zero or more) other nodes.
global/local knowledge	Know	where the placement decision was made using information from many / one (local) node.
activity history	Hist	for which only objects accessed during this many past intervals can be placed.
reactive placement	React	that is reactive; proactive placement implied otherwise.

Table 2. The six heuristic properties considered in this paper.

is the case for heuristics that create more replicas for popular objects. Again, the solution to the problem provides the optimal replication factor that satisfies this constraint.

$$\sum_{n \in N} store_{nik} = \sum_{n \in N} store_{n,0,0} \quad \forall i,k \tag{17}$$

$$\sum_{n \in N} store_{nik} = \sum_{n \in N} store_{n,0,k} \quad \forall i,k \tag{17a}$$

Routing knowledge. With some heuristics, such as caching, a node has no knowledge of what replicas other nodes cache. Upon a miss, the node has to fetch the object from a predetermined, usually remote location that stores all objects (origin server). On the other hand, with cooperative caching, a node knows what objects are stored at some other, nearby nodes and can fetch them from there [19]. Similarly, many centralized heuristics fetch the closest replica in the system [11]. We call this property *routing knowledge*. It is represented by matrix $fetch$. $fetch_{nm} = 1$, when node n knows of the objects stored on m. Constraints (18) and (19) state that n can only fetch objects from a node m for which $fetch_{nm} = 1$.

$$covered_{nik} \leq \sum_{m \in N} dist_{nm} \cdot store_{mik} \cdot fetch_{nm} \quad \forall n,i,k \tag{18}$$

$$route_{nmik} - fetch_{nm} \leq 0 \quad \forall n,m,i,k \tag{19}$$

Global/Local knowledge. When a heuristic makes a placement decision for a specific node, it takes into account activity at that node and potentially other nodes in the system. In one extreme, some heuristics (e.g., caching) make placement decisions for a node based on the accesses that originated on that node alone (local). On the other extreme, heuristics that use knowledge from the entire system (typically, centralized heuristics) may place an object on a node as a result of activity somewhere else in the system (global). To represent these two cases and anything in between, we use matrix *know*. $know_{nm}$ indicates that knowledge of accesses originating at node m, even if not directed to node n, is used to decide the placement of objects on n. We call all nodes m such that $know_{nm} = 1$ to be in the the *sphere of knowledge* of node n. In the next paragraph, we discuss how matrix *know* is incorporated into our model.

Activity history. A heuristic decides to place a replica on the basis of the system's activity during one or more intervals. The activity history property captures the number of intervals considered. This is specified in the model by matrix $hist$. $hist_{nik} = 1$, if node n accessed object k during or before interval i within the history considered by the heuristic.

We exemplify this with the two extreme cases. First, when the history is a *single* interval, a heuristic that decides about placement in interval i on node n can consider only objects referenced within the node's sphere of knowledge during i. Caching is an example of such heuristics. Second, when the history is *all* intervals, a heuristic can consider any object referenced within the node's sphere of knowledge, throughout the execution, up to and including i. We introduce constraint (20) to capture the activity history. Note, that it does not make sense to place an object more than one interval before it is accessed for first time, because this would increase storage cost unnecessarily, with the same networking cost. Thus, this constraint provides a lower bound for heuristics that perform prefetching.

$$create_{nik} \leq \sum_{m \in N} hist_{nik} \cdot know_{nm} \quad \forall n,i,k \tag{20}$$

Reactive placement. Until now, the IP formulation implicitly captures solutions of *proactive* heuristics; the activity history covers not only previous intervals but also the current interval, at the beginning of which the placement decision is made. This corresponds to heuristics that perform placement knowing the accesses to take place in the current interval or prefetch objects that have not been accessed before. However, there are many heuristics that are reactive, i.e., they can only place objects that have been accessed in the past (before the current interval). Examples include caching (without prefetching) and other on-line heuristics that have no prior knowledge of the workload.

We capture the solutions of reactive heuristics by further constraining the activity history property. The new version states that if $hist_{n,i-1,k} = 0$ for a given object k during the previous interval in the sphere of knowledge of node n, object k cannot be created in n during the current interval. For example, if the activity history is just a single interval and the heuristic is using only local information, an object cannot be created on a node unless it was accessed by that node during the previous interval. This is true, for example, with local caching.

$$create_{nik} \le \sum_{m \in N} hist_{n,i-1,k} \cdot know_{nm} \quad \forall n,i,k \tag{20a}$$

$$hist_{n,-1,k} = store_{n,-1,k} \quad \forall n,k \tag{21}$$

4.2. Classes of Heuristics

Solving the basic MC-PERF problem of Section 3.1, we obtain a general lower bound that applies to any possible replica placement algorithm. By including one or more of the above constraints to the MC-PERF formulation, we constrain the possible solutions to only those that are possible with a class of heuristics. Thus, solving the IP problem with additional constraints provides lower bounds for the cost of the corresponding heuristic classes. We use these lower bounds, in Section 6, to argue about the merits of different classes of heuristics. Table 3 includes an extensive list of heuristics from the literature and shows how they are captured by various combinations of heuristic properties.

A number of centralized heuristics that use global knowledge of the system to make placement decisions are constrained only by a storage constraint or a replica constraint. Other heuristics are run in a completely decentralized fashion, only having knowledge of activity on the local node where they run. The common caching protocols are subcases of those heuristics; they react only to the last access initiated on the local node; all misses go to a designated "origin" node. Cooperative caching is captured by constraints that reflect the extended knowledge of 1) activity on other nodes in the system, and 2) what objects other nodes store. Last, performing proactive placement based on knowledge or speculation of accesses to happen in the current time interval (as opposed to just past intervals), captures variations of caching and cooperative caching, with prefetching.

4.3. Evaluation Interval Values

An important parameter of the MC-PERF problem formulation is the *evaluation interval* (Δ) used to calculate lower cost bounds. Intuitively, the evaluation interval must be as small as possible for the solutions considered—the more often you make placement decisions, the better your chances to meet the performance requirements and reduce cost. In theory, as Δ approaches 0, the lower bound converges to a value that is the lowest possible for any Δ; the storage part of the cost function is minimized. However, that lower bound would be lower than the cost of any realistic heuristic. In practice, Δ should be set according to the smallest possible evaluation period of the targeted heuristics.

There are two types of heuristics to be considered when setting Δ in MC-PERF. First, there are heuristics that perform placement evaluations every P units of time. For such heuristics, we have shown that an evaluation interval $\Delta = P_{min}/2$, where P_{min} is the smallest P on any node and at any time in the system, is sufficient to provide a lower bound (see the appendix) . Second, there are heuristics (e.g., caching) that perform placement evaluations after every access on a node. In that case, Δ is the minimum time between any two accesses within the sphere of knowledge of any node.

5. Deriving Lower Bounds

In general, an IP problem can be solved exactly with an IP solver, resulting in a *tight lower bound*. However, such an approach is feasible only at a very small scale. To efficiently calculate *lower bounds*, one has to sacrifice tightness. On the other hand, those lower bounds must be close to the tight lower bound, otherwise they would be of no use. Thus, the requirement is to obtain lower bounds that are close to tight but still can be computed in reasonable time.

There are two common ways to obtain lower bounds: either devise an *approximation algorithm* for the problem, or use a *linear programming relaxation* (LP) of the problem combined with a rounding algorithm. It has been shown that SET-COVER cannot be approximated with a constant approximation factor [17], and there are no known approximation methods for either dynamic SET-COVER or static SET-COVER with performance constraints (both can be reduced to instances of MC-PERF). These are worst-case estimates for approximation algorithms, for any input data. Instead, we obtain *lower bounds* for MC-PERF using linear relaxation and a rounding algorithm (a heuristic itself), as this is applicable to instances of the problem (specific input data) and, in general, provides lower bounds that are tighter than the theoretical worst-case.

The linear relaxation is obtained by letting the binary decision variables in MC-PERF have fractional values (between 0 and 1). Solving the LP problem is orders of magnitude faster. The cost of the LP solution is always less than or equal to the cost of the IP solution. We developed a domain-specific rounding algorithm that rounds the fractional values of the LP solution to produce a feasible solution to the IP problem, with a low additional cost due to the rounding. The details of the rounding algorithm are an operations research topic and are described in the appendix . In that paper, we show that the algorithm results in close-to-tight lower bounds for instances of MC-PERF (always within 10% as opposed to up to 80% obtained with generic rounding algorithms).

The number of constraints and variables in MC-PERF is $O(|N||I||K|)$, where $|N|$ is the number of nodes, $|I|$ the number of intervals and $|K|$ the number of objects. We use CPLEX [2] to solve the LP problem. On a workstation

Heuristic properties						**Class of heuristics represented**	**Examples**
SC	RC	*Route*	*Know*	*Hist*	*React*		
×		global	global	multi		storage constrained heuristics	[3, 4]
	×	global	global	multi		replica constrained heuristics	[3, 11]
×		local	local	multi		decentralized storage constrained heuristics w/ local routing	[4, 12]
×		local	local	single	×	local caching	[14]
×		global	global	single	×	cooperative caching	[7]
×		local	local	single		local caching with prefetching	[14]
×		global	global	single		cooperative caching with prefetching	[19]

Table 3. Examples of heuristic classes captured by combinations of heuristic properties.

with a 2.4GHz Pentium IV processor, the running time of CPLEX on the instances of the LP problem studied in this paper (realistically sized systems) ranges from less than 1 minute to about 12 hours with all constraints enabled. The rounding algorithm execution takes just seconds even for large systems. These times are adequate for the proposed off-line method.

6. Experimental Results

This section presents experimental results from applying the proposed method to a real-world system, a remote office file access service. We show that:

1. lower bounds provide a good guideline for the inherent cost of heuristics;
2. different workloads, systems and/or performance goals result in choosing completely different placement heuristics;
3. the choice of a good heuristic can result in up to 7.5x savings in infrastructure cost (storage capacity and network bandwidth), compared to a commonly used default heuristic;
4. the method scales sufficiently well for realistic systems and workloads.

We demonstrate the versatility of the method by showing the above points for two different deployment scenarios. First, in systems where the infrastructure already exists and we need to come up with a good heuristic to deploy in the system. Second, in systems where there is no infrastructure in place and we need to decide on the locations where replicas can be placed as well as a heuristic that delivers the desired performance at a low cost.

The case study we consider reflects the network of a corporation with twenty different sites that are distributed across multiple geographic locations and are interconnected by network links of different latencies and capacities. In fact, for our experiments, we use a network topology based on an actual Internet AS-level topology [15], with 20 nodes. A single AS-level hop in the topology takes between 100 ms and 200 ms. Each node represents a site and has a set of local users. The population of users is unevenly distributed emulating the fact that some sites are bigger or more active than others. All users, irrespective of location, access a common set of data objects (files). Each node in the topology is a candidate location for placing object replicas. One of the nodes represents the data center located at the headquarters of the corporation. This node stores all the objects in the data set.

We use two workloads representing two completely different object popularity distributions. Workload WEB has a heavy-tailed Zipf distribution with many unpopular objects. It is derived from the WorldCup98 web-server logs [1] and is representative of accesses to web pages. Workload GROUP includes only popular objects with a uniform popularity distribution. It represents a working group accessing the files of an active collaborative project. All nodes are highly active and they all generate requests to the 1,000 data objects we consider. In total, we have 300K requests in WEB and 16M requests in GROUP. The most popular object has 36K accesses in both workloads. The least popular object has just 1 access in WEB and 8.5K accesses in GROUP. The duration of both workloads is 1 full day. To keep the discussion simple, all objects are of the same size.

For the experiments, we consider the cost of replicating an object to be 1 and the cost of storing one object for one hour to be also 1.[1] All other costs (enabling nodes, writes, penalties for late responses) are set to zero. Performance goals are specified on a per-user basis over-all objects. The discussion in this section is concerned only with the QoS goal metric; the methodology for average latency performance goals is the same. For the calculations of the lower bounds, we use an evaluation interval of 1 hour (to keep the computational complexity reasonable). Deployed heuristics are evaluated using simulation. We use their actual evaluation interval, which might be smaller than that, e.g., every single access in the case of caching. Despite the coarse interval used for the IP problem solution, we have observed in practice that the obtained lower bounds are still indica-

1 What matters for our evaluations is the relative cost of different approaches, not an actual monetary value.

tive of the evaluated heuristics.

6.1. Heuristic Selection Methodology

In this section, we go through the first type of scenario, where a system designer has to choose (and then implement and apply) a heuristic for a remote office file service. We assume that an infrastructure already exists, with 20 file servers, one on each site, participating in offering the file service to the users. The storage and bandwidth of these servers are shared with other users and applications. Thus it is important to consume as few resources as possible. The designer has examples of possible workloads and a specification of the desired QoS goal. For the experiments, we assume that the desired latency threshold is 150 ms.

The designer starts by calculating the general lower bound and the lower bounds for all classes of heuristics that could be implemented in the system. The key idea of the method is to choose a heuristic from the class with the lowest bound. If this lower bound is close to the general lower bound, there exists no heuristic that could be significantly better than the chosen one. Otherwise, the designer should probably look for other classes of heuristics, with bounds closer to the general one, and choose a heuristic among them. If there is a number of classes with comparable lower bounds and no better class can be found, then a heuristic from any of them is a good choice. Of course, the designer must also take into account the properties of the heuristics under consideration. For example, in certain deployments, a heuristic that requires global knowledge of the system may not be applicable.

Figure 1 shows the lower bounds of the classes of heuristics that can be implemented in the remote office system. From the figure, we see that a storage constrained heuristic should be a good choice for WEB, while a replica constrained heuristic should be preferred for GROUP. In either case, caching does not seem appropriate, since the corresponding lower bounds are much higher. For WEB, local caching cannot even achieve a QoS goal above 99%.

In the case of workload WEB, the replica constrained bound is up to 2 times the storage constrained bound and 7 times the general bound. The reason is that, due to the replica constraint, unpopular objects in the heavy-tailed WEB have as many replicas as popular ones. For GROUP, the replica constraint has minimal impact to cost—the lower bound nearly overlaps with the general lower bound. This is because GROUP includes only objects that are accessed often. Thus, creating the same number of replicas for every object works well.

In the case of WEB, storage constrained heuristics have the lowest cost. All nodes generate many requests and there are few popular objects. The same small storage capacity on all nodes is sufficient to store enough objects to satisfy the QoS goal. For GROUP, the storage constraint results in costs that are always more than 5 times the replica constrained lower bound. The reason is that assuming the same capacity across all nodes results in higher overall storage use.

Both classes of *caching heuristics* (local and cooperative) have by far the highest cost for high QoS levels, in the case of WEB. Their lower bounds are 1.2 to 5 times the storage constrained bound. The reason is that the combination of all the properties (constraints) of caching result in less optimal use of resources for WEB. In the case of GROUP, the lower bounds for caching and cooperative caching overlap with the bound for storage constrained heuristics, at 5 times the cost of the replica constrained bound. The limiting factor for this workload is the storage constraint property of caching.

Given the analysis above, it is clear that caching heuristics are not appropriate for this system. Either they cannot meet the performance goals or they have a high inherent cost. A storage constrained heuristic would be most appropriate for WEB, and a replica constrained heuristic for GROUP. Based on the expected workload, the designer would now look in Table 3 or in the literature and pick the greedy global heuristic [4] for WEB or Lili Qiu's greedy placement heuristic [11] for GROUP. The chosen heuristic is then applied in the system. In practice, the designer is done at this stage.

Here, we evaluate the robustness of the method by obtaining the actual cost of heuristics when applied in the system. The results are shown in Figure 2. In summary: 1) The proposed method identifies the right type of heuristic; the cost of the actual heuristic is below the second lowest bound (with the exception of the 99.99% point for WEB), indicating that it is impossible for any other examined class of heuristics to achieve the QoS goal at a lower cost. 2) Using caching, a popular heuristic, would result in up to 7.5 times cost increase in our case study.

6.2. Infrastructure Deployment Methodology

Here, we discuss the scenario where there are no nodes (file servers) deployed in the system. The designer has to decide not only what heuristic to use but also which sites to deploy nodes in. The users of one site are all assigned to a specific node. If there is no node deployed in their local site, then they are assigned to the node of another, neighboring site. All user accesses are directed to their assigned node. If a file is not found there, then the request is forwarded to the headquarters node that stores all files. In these experiments, we do not consider prefetching; all heuristics considered are reactive.

The methodology used in this scenario consists of two phases. In phase one, the designer solves the MC-PERF problem, with one difference—a node opening cost is in-

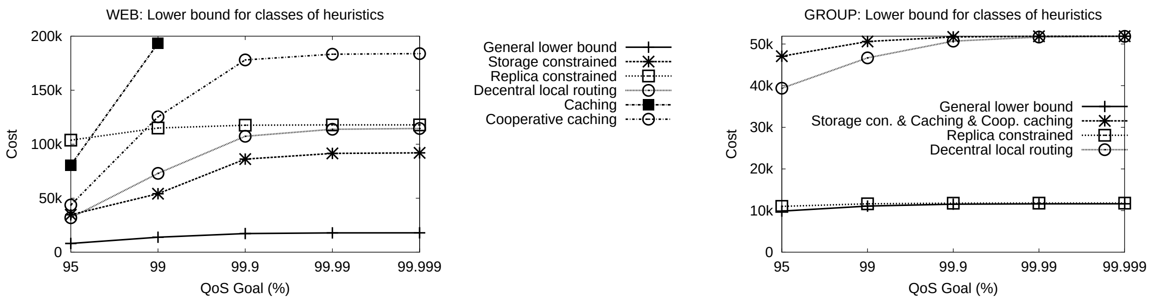


Figure 1. The lower bound for various heuristic classes as a function of QoS for WEB and GROUP.

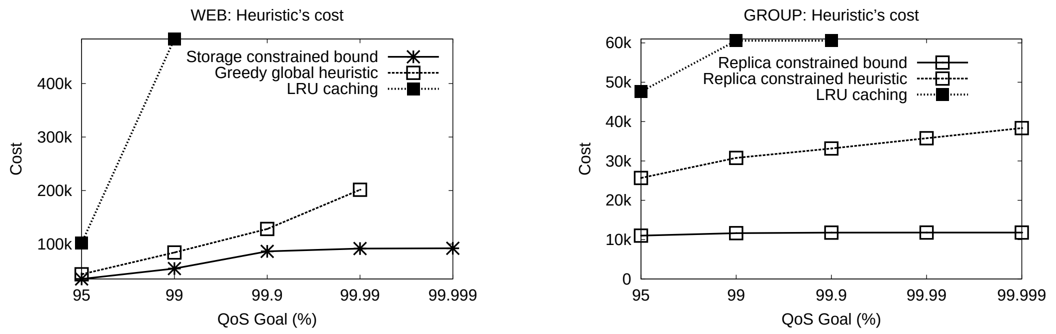


Figure 2. The cost of the chosen heuristic (deployed in the system) compared to the lower bound. The cost of LRU caching is also shown for comparison.

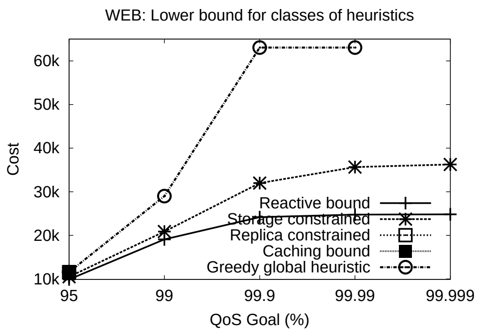
cluded in the cost function ($\zeta = 10,000$ for our experiments). The solution provides the smallest number of nodes needed and their location in the system, to achieve the required performance goal. This information is returned in decision variable *open*, a matrix capturing the locations where nodes should be deployed. In our case study, the solution indicates that 6 nodes, at certain locations in the topology, are sufficient to offer the required QoS goal.

In phase two, the designer calculates lower bounds for the 6-node topology; all other nodes are removed from the topology. The method proceeds as described in Section 6.1 (without considering opening costs anymore). Lower bounds have to be calculated for the 6-node topology, since the solution is more constrained than the 20-node case—accesses are now directed to only one of the 6 nodes. The results are shown in Figure 3. For WEB, a replica constrained or a caching heuristic can only deliver up to 95% QoS. Note that this is a different conclusion from Figure 1, when both these classes of heuristics could deliver a higher QoS. As in the previous section, a storage constrained heuristic seems to be the right choice for WEB. For GROUP, on the other hand, things are very different in this scenario. The storage constrained, replica constrained and caching bounds are all low and close to each other. The choice could be a heuristic from any of these classes. Thus, the most appealing choice is caching, as it is well understood.

7. Conclusions

Choosing the right heuristic for replica placement in a large distributed system typically depends on the intuition of the system designer. A simple experimental evaluation [6] shows that the heuristic used makes a considerable difference in terms of the required infrastructure cost, including the cost for storage capacity and network bandwidth. Moreover, the choice of the heuristic to be used is often not an obvious one.

The paper proposes a methodology to assist system designers choose replica placement heuristics that meet their performance goals for the lowest possible infrastructure cost. The method can be applied in one of two ways. First, it can be used to decide on the heuristic to deploy in an existing infrastructure. Second, it can be also used to decide about the combination of heuristic to deploy and locations that should be opened for replica placement in the system. We demonstrate the applicability of the method in a real-world case study and for two different workloads.



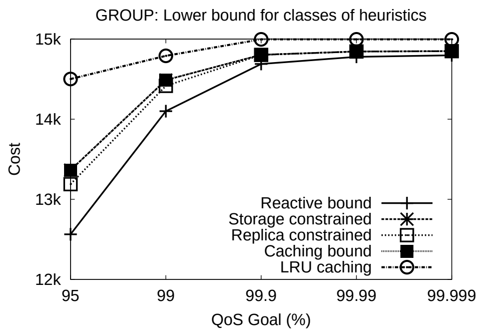


Figure 3. The lower bound when only the six nodes deployed are entered as a topology. The replica constrained and caching bounds are overlapping for GROUP.

The proposed method is an off-line approach, in the sense that it has to be run explicitly by the designer as changes in the system occur. Currently, we are investigating on-line approaches to dynamically adapt the placement heuristic to changing systems and workloads.

References

[1] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. Technical Report HPL-1999-35R1, HP Laboratories, 1999.

[2] *ILOG CPLEX*. www.ilog.com.

[3] L. Dowdy and D. Foster. Comparative Models of the File Assignment Problem. *ACM Computer Surveys*, 14(2):287–313, 1982.

[4] J. Kangasharju, J. Roberts, and K. Ross. Object Replication Strategies in Content Distribution Networks. *Computer Communications*, 25(4):367–383, March 2002.

[5] M. Karlsson and C. Karamanolis. Bounds on the Replication Cost for QoS. Technical report, Hewlett Packard Labs, July 2003.

[6] M. Karlsson and M. Mahalingam. Do We Need Replica Placement Algorithms in Content Delivery Networks? In *Proceedings of the International Workshop on Web Content Caching and Distribution (WCW)*, pages 117–128, August 2002.

[7] M. Korupolu, G. Plaxton, and R. Rajaraman. Placement Algorithms for Hierarchical Cooperative Caching. *Journal of Algorithms*, 38(1):260–302, January 2001.

[8] J. Kubiatowicz et al. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, November 2000.

[9] S. Owen and M. Daskin. Strategic facility location: A review. *European Journal of Operational Research*, 111:423–447, 1998.

[10] S. Owen and M. Daskin. Two new location covering problem: The partial covering P-center problem and the partial set covering problem. *Geographical Analysis*, 31:217–235, 1999.

[11] L. Qiu, V. Padmanabhan, and G. Voelker. On the Placement of Web Server Replicas. In *Proceedings of IEEE INFOCOM*, pages 1587–1596, April 2001.

[12] M. Rabinovich and A. Aggarwal. RaDaR: A Scalable Architecture for a Global Web Hosting Service. In *Proceedings of the 8th International World Wide Web Conference*, pages 1545–1561, May 1999.

[13] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 15–30, Boston, MA, USA, December 2002.

[14] A. Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[15] *Telestra.net*. http://www.telestra.net.

[16] C. Toregas, C. ReVelle, and L. Bergman. The location of emergency service facilities. *Operations Research*, 19:1363–1373, 1971.

[17] V. Vazirani. *Approximation Algorithms*. ISBN: 3-540-65367-8. Springer-Verlag, 2001.

[18] A. Venkataramanj, P. Weidmann, and M. Dahlin. Bandwidth Constrained Placement in a WAN. In *ACM Symposium on Principles of Distributed Computing (PODC)*, August 2001.

[19] G. Voelker, E. Anderson, T. Kimbrel, M. Feeley, J. Chase, A. Karlin, and H. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 33–43, 1998.

[20] H. Yu and A. Vahdat. Minimal Replication Cost for Availability. In *21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 98–107, June 2002.

Appendix

In section A of the Appendix, we prove that the MC-PERF problem is NP-hard. In section B, we prove some interesting properties regarding the evaluation interval, a fundamental element of the problem definition. The main part of the Appendix is Section C, where we provide the full details of the rounding algorithm and its extensions, as well as a proof that it always arrives to a feasible solution.

A. NP-Hardness

In this section, we show that our problem is NP-hard by reducing SET-COVER to MC-PERF.

Theorem 1 *The minimal replication cost for QoS problem (*MC-PERF*) is NP-hard.*

Proof for Theorem 1: We show this by a polynomial time reduction of SET-COVER [17] to MC-PERF. Let each candidate set in SET-COVER correspond to a unique node in MC-PERF; this set of nodes is denoted C. Let each element in SET-COVER correspond to a unique node in MC-PERF; this set of nodes is denoted E. Sets C and E are disjoint. Thus, the set of nodes considered in MC-PERF is $N = C \cup E$. For all $c \in C$ and $e \in E$, set $dist_{ce} = 1$ and $dist_{ec} = 1$ iff element e is covered by candidate set c in SET-COVER. All other entries in the *dist* matrix are set to 0. In MC-PERF, consider an overall user QoS target of 100%, and let there only be one evaluation interval and one object. Set $demand_{e,0,0} = 1 \forall e \in E$ and set all other entries in the *demand* matrix to 0. Let $\alpha = 1$ and $\beta = 0$.

With 100% QoS, all the requests in the system have to access the object within the latency threshold. As requests originate only from nodes in E, we only have to make sure that all nodes in E can access the object within the threshold. Recall that $dist_{ce}$ is 0 expect if node c corresponds to a candidate set that covers the element that corresponds to e. Thus, the only way to satisfy the request from e within the latency threshold is to store the object replica on a node that correspond to one of the candidate sets covering e. We can now easily see that choosing the minimal replication cost in this instance of the MC-PERF problem is the same as choosing the minimal number of covering sets in the SET-COVER problem. □

B. The Evaluation Interval

This section proves two results regarding *evaluation intervals*, an important parameter of our problem formulation. First, if a lower bound is obtained for evaluation interval Δ, we show what other intervals this lower bound applies to. Second, we show what evaluation interval should be used to obtain a lower bound for heuristics that are evaluated after every single access.

Theorem 2 *A lower bound produced with an evaluation interval of Δ, is a lower bound for any evaluation interval Δ' such that $\Delta' \geq 2\Delta$ or $\Delta' = \Delta$.*

Proof for Theorem 2: Assume for contradiction, that there exists a $\Delta' \geq 2\Delta$ that provides a placement with lower cost. That means that there is at least one object, o_1, for which the following hold. There is one access at $t_1 - \varepsilon$, where $\varepsilon < \Delta$, and another access at t_1. The cost of storing object o_1 for duration Δ is $\alpha\Delta$. Assume that because of the two accesses, o_1 has to be stored on some node for two intervals, for a cost of $2\alpha\Delta$. Consider another evaluation interval, Δ', such that both of these accesses occur within the same evaluation interval. Δ' provides a placement with lower cost iff the cost of storing the object during this evaluation interval $\alpha\Delta' < 2\alpha\Delta$. This can only be true if $\Delta' < 2\Delta$, a contradiction. □

In the following, we show the evaluation interval that should be used to obtain a lower bound for heuristics that are evaluated after every single access. This might be a small evaluation interval that forces us to solve the problem for a large number of intervals. The lemma below restricts the evaluation intervals we consider, without affecting the lower bound.

Lemma 1 *Let $A_{nm} = dec_{nm} \vee dist_{nm}, \forall n,m$. The placement and accesses of node n can only be affected by what happens on node m iff $A_{nm} = 1$.*

Proof for Lemma 1: Node n can only interact with node m if it either can fetch objects from m ($dist_{nm} = 1$) or use knowledge from node m in making its decision ($dec_{nm} = 1$). If either of these are true $A_{nm} = 1$. □

Theorem 3 *Let m_1 be the minimum time between any two accesses between all nodes n and m where $A_{nm} = 1$, and m_2 be the next lowest such time such that $m_1 \neq m_2$. The lower bound of a heuristic evaluated after each single interval can be computed with* MC-PERF *by setting the evaluation interval (Δ) to $\Delta = m_1/2$ if $2m_1 \geq m_2$ and to $\Delta = m_1$ if $2m_1 < m_2$.*

Proof for Theorem 3: As the evaluation interval of a heuristic executed at each single access is the time between two accesses, m_1 is the lowest evaluation interval in the system. According to Theorem 2, a Δ of $m_1/2$ is the lower bound of any heuristic evaluated with an evaluation interval greater to or equal to m_1. This would then suffice to correctly provide the lower bound of such a heuristic. However, if there are no inter reference times in the workload between m_1 and $2m_1$ there is no need to include this range, thus Δ can be set to m_1 to save some computational resources. The minimum time need only to be computed between all nodes n and m that could possibly affect each other ($A_{nm} = 1$). □

C. Rounding Algorithm

The linear relaxation of the IP formulation of the problem is derived by allowing the decision variables in the problem to have fractional instead of binary values, as shown below.

$$store_{nik}, covered_{nik}, create_{nik} \leq 1 \quad \forall n,i,k \tag{6a}$$
$$store_{nik}, covered_{nik}, create_{nik} \geq 0 \quad \forall n,i,k \tag{6b}$$
$$\tag{6c}$$

$$route_{nmik} \leq 1 \quad \forall n,m,i,k \tag{10a}$$
$$route_{nmik} \geq 0 \quad \forall n,m,i,k \tag{10b}$$
$$\tag{10c}$$

$$open_n \leq 1 \quad \forall n \tag{15a}$$
$$open_n \geq 0 \quad \forall n \tag{15b}$$
$$\tag{15c}$$

As discussed in Section 5, the approach used to obtain lower bounds for MC-PERF and its extensions is a combination of linear relaxation and a rounding algorithm. We propose a greedy rounding algorithm that provides feasible solutions with cost $cost_{feas}$ close enough to $cost_{LP}$ to allow meaningful conclusions about the tightness of the lower bound. The algorithm is described in detail, in Figures 6, 7 and 5. The intuition behind the algorithm have already been described in Section 5.

The LP solution of the problem may assign fractional values to variable $store_{nik}$ for certain nodes, objects and intervals. A rounding algorithm rounds all fractional values either up to 1 or down to 0. In general, rounding down decreases the cost of the solution but also decreases the achieved QoS (percentage of accesses within the latency threshold). On the other hand, rounding up increases the cost and the achieved QoS. Thus, a rounding algorithm should find the right balance between rounding values up and down, so that the QoS goal is met (feasible solution) with minimal additional cost due to the rounding, i.e., without paying for additional QoS that is not required. Correctness can be ensured by not allowing a value to be rounded down, unless it has been preceded by a round-up that increased the QoS by at least as much as the negative QoS impact due to the subsequent round-down.

Based on this principle, we propose a simple greedy rounding algorithm, which works as follows. First, some fractional value of the LP solution is rounded up to 1. This increases QoS and cost. The algorithm then rounds down as many fractional values as possible to reduce cost, as long as the QoS goal is not violated. When no more values can be rounded down, the process is repeated, until there are no more fractional values in the solution. The final result is a feasible solution, with cost $cost_{feas}$.

The fundamental issue that the algorithm needs to address is how to choose the values to be rounded up and down and the order in which to do that. To make these decisions, the algorithm uses a third metric (in addition to cost and QoS), called *reward*. To reach a rounding decision, the algorithm considers the achieved QoS *only* due to values that are set to 1, at each stage of its execution. Reward reflects the impact to that QoS by rounding up or down a specific value.

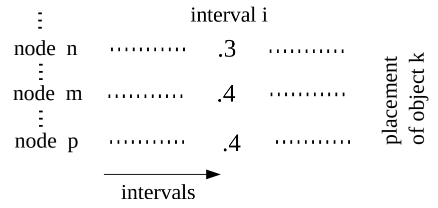


Figure 4. Example of reward versus QoS impact. Nodes n, m and p are within the latency threshold from each other. According to the LP solution, they store (fractional values of) object k during interval i; there are no other nodes within the latency threshold from them that store k during i. Since, the total value for k across the three nodes is ≥ 1 in i, rounding up any one of them does not have any impact on QoS. Reward is the impact to QoS if all fractional values are considered 0.

To explain why we need this metric, consider the round-up case, where we would like to pick the value that provides the highest increase in QoS at the lowest cost. In Figure 4, rounding up any one of the three fractional values would have zero impact to QoS, since it would not increase the demand satisfied within the latency threshold. However, we have to round up at least one of those three values to produce a feasible solution. Thus, we choose to round up the value with the highest reward and lowest cost impact (lowest $cost/reward$ ratio). Similarly, for rounding down, the value with the highest $cost/reward$ ratio is chosen in each iteration.

For notational convenience, the fractional value in $store_{nik}$ is denoted v. Let $v.node = n$, $v.interval = i$ and $v.object = k$. Also, let $v.value = store_{nik}$, $v.succ = store_{n,i-1,k}$, and $v.prev = store_{n,i+1,k}$. To cover the corner cases for the beginning or end of the sequence of intervals, we set $v.prev = 0$ if $v.interval = 0$ and $v.succ = v.value$ if $v.interval$ is the last interval.

The algorithm is shown in Figure 5. For every fractional value, we calculate the three metrics that the algorithm uses: *cost*, *reward*, and the QoS impact (*qos*) of rounding it up or down. Calculating *reward*, and *qos* is straightforward—they are proportional to the different demand satisfied due to the rounding. *cost* consists of *storage cost* and *replica creation cost*. The impact on storage cost is proportional to the value change.

The impact on replica creation cost, on the other hand, is not necessarily proportional to the value change. It depends on the values before and after the target interval (for the same node and the same object). Consider, for example, rounding up the fractional value of $store_{nik}$. If at least one of

```
$cost$ = lower bound from linearization
$qos = T_{qos}$
$\mathcal{V} \leftarrow$ set of all fractional values in $store$
while $\mathcal{V} \neq \emptyset$
    $\forall v \in \mathcal{V}$ calculate_round_up_benefit($v$)
    find $v \in \mathcal{V}$ with lowest $v.cost/v.reward$
    // round up v
    $store_{v.node,v.interval,v.object} = 1$
    $cost = cost + v.cost$
    $qos = qos + v.qos$
    $\mathcal{V} \leftarrow \mathcal{V} - \{v\}$
    repeat
        $\forall v \in \mathcal{V}$ calculate_round_down_benefit($v$)
        $C \leftarrow \{v \in \mathcal{V} : qos + v.qos \geq T_{qos}\}$
        if $C \neq \emptyset$
            find $v \in C$ with $v.reward = 0 \wedge v.cost < 0$
            if no $v$ found
                find $v \in C$ with highest $v.cost/v.reward$
            // round down v
            $store_{v.node,v.interval,v.object} = 0$
            $cost = cost + v.cost$
            $qos = qos + v.qos$
            $\mathcal{V} \leftarrow \mathcal{V} - \{v\}$
    until $C = \emptyset$
if storage constraint present
    find $c_{max}$, the max no. of objects stored on a node
        during any interval
    $\forall i \in I, n \in N; cost = cost + \alpha \cdot (c_{max} - \sum_{k \in K} store_{nik})$
    find $\bar{c}_n$, the max no. of objects stored on node $n$
        during any interval
    $\forall n \in N; cost = cost + \beta \cdot (c_{max} - \bar{c}_n)$
if replica constraint present
    find $c_{max}$, the max no. of replicas of any object
        during any interval
    $\forall i \in I, k \in K; cost = cost + \alpha \cdot (c_{max} - \sum_{n \in N} store_{nik})$
    find $\bar{c}_k$, the max no. of replicas of object $k$
        during any interval
    $\forall k \in K; cost = cost + \beta \cdot (c_{max} - \bar{c}_k)$
output $cost$
```

Figure 5. **The rounding algorithm used to find the tightness of the lower bound.**

```
function calculate_round_up_benefit($v$)
    if ($v.succ \geq v.value \geq v.prev$)
        $v.cost = \beta \cdot (1 - v.succ)$
    else if ($v.succ < v.value < v.prev$)
        $v.cost = \beta \cdot (1 - v.prev)$
    else if ($v.succ \geq v.value \wedge v.prev > v.value$)
        $v.cost = \beta \cdot (1 - v.prev - (v.succ - v.value))$
    else if ($v.succ \leq v.value \wedge v.prev < v.value$)
        $v.cost = \beta \cdot (1 - v.value)$
    $v.cost = v.cost + \alpha \cdot (1 - v.value)$
    // reward only from nodes within the latency threshold
    // that are not fully covered by another node
    $M \leftarrow \{n \in N : dist_{n,v.node} = 1 \wedge$
            $\sum_{m \in N} \lfloor store_{m,v.interval,v.object} \rfloor \cdot dist_{nm} = 0\}$
    $v.reward = \sum_{n \in M} demand_{n,v.interval,v.object}$
    // QoS impact only if sum of all stored replicas
    // within latency threshold is less than one before rounding
    $M \leftarrow \{n \in N : dist_{n,v.node} = 1\}$
    if $\sum_{n \in M} store_{n,v.interval,v.object} < 1$
        $v.qos = v.reward \cdot (1 - \sum_{n \in M} store_{v.node,v.interval,v.object})$
    else
        $v.qos = 0$
```

Figure 6. **The rounding up benefit function.**

$store_{n,i-1,k}$ and $store_{n,i+1,k}$ is set to 1, then the cost for a full replica of k on node n during i is included in the solution cost and rounding up $store_{nik}$ has zero impact on replication cost. If both those intervals have values less than $store_{nik}$, the impact on replica creation cost is $\beta(1 - store_{nik})$. If one of them, say $store_{n,i-1,k}$, has a higher value, then the impact is $\beta(1 - store_{n,i-1,k})$. In fact, the impact may be negative if both neighboring intervals have higher values than $store_{nik}$. The algorithm is show in Figure 6

The rounding down algorithm displayed in Figure 7 is symmetric to the rounding up case. There are the following cases to consider (reflected in "if" statements in Figure 7): 1) If either $v.succ$ or $v.prev$ have values higher than $v.value$, then unnecessary replica creation cost has been included in the solution, as the lesser value of $v.succ$ and $v.prev$ is not needed. This can be deducted from the cost. If $v.succ$ is higher, the deduction is $\beta \cdot v.prev$; if $v.prev$ is higher, it is $\beta \cdot v.succ$. 2) When the values of both $v.succ$ and $v.prev$ are lower than $v.value$, we no longer have to pay for the creation cost between $v.prev$ and v. Thus, this is deducted from cost. However, there is now a replica creation cost between v and $v.succ$ that we previously did not have to pay for. Finally, the impact to cost is $-\beta(v.succ - (v.value - v.prev))$. 3) For the last case, when both $v.prev$ and $v.succ$ have values higher than $v.value$, we pay for replica creation between v and $v.succ$. As the value is rounded down from $v.value$ to 0, we have to pay an extra $\beta \cdot v.value$ to reflect the replica creation cost for $v.succ$.

The algorithm of Figure 5 is described for a QoS defined over all users and all objects. Adapting the algorithm to any of the other definitions of QoS is straightforward. For a per-object QoS, the algorithm is run separately for every single object k in the system. For each run, QoS and reward refer to the corresponding object alone. For a per-user QoS, each node has its own QoS requirement (recall that we assume one user per node). When a fractional value is rounded down, the rounding must not violate any node's QoS requirement.

In order for the rounding algorithm to produce feasible solutions with one or more of the heuristic properties enabled, we have to add some functionality to it.

When the storage constraint is enabled, we search the solution for the node that stored the most object replicas during an interval (to keep the discussion simple, we assume that all objects are of the same size and thus capacity is measured in number of objects). Denote this number c_{max} and denote the capacity used in interval i and node n c_{ni}. For every node and interval, we add $\alpha(c_{max} - c_{ni})$ to the cost, forcing all nodes to use the same amount of stor-

```
function calculate_round_down_benefit($v$)
    if ($v.succ \geq v.value \geq v.prev$)
        $v.cost = -\beta \cdot v.prev$
    else if ($v.succ < v.value < v.prev$)
        $v.cost = -\beta \cdot v.succ$
    else if ($v.succ \geq v.value \wedge v.prev > v.value$)
        $v.cost = -\beta \cdot v.value$
    else if ($v.succ \leq v.value \wedge v.prev < v.value$)
        $v.cost = -\beta \cdot (v.succ - (v.value - v.prev))$
    $v.cost = v.cost - \alpha \cdot v.value$
    // reward only from nodes within the latency threshold
    // that are not fully covered by another node
    $M \leftarrow \{n \in N : dist_{n,v.node} = 1 \wedge$
            $\sum_{m \in N} \lfloor store_{m,v.interval,v.object} \rfloor \cdot dist_{nm} = 0\}$
    $v.reward = -\sum_{n \in M} demand_{n,v.interval,v.object}$
    // QoS impact only if sum of all stored replicas
    // within latency threshold would become
    // less than one after rounding
    $M \leftarrow \{n \in N : dist_{n,v.node} = 1\}$
    if $\sum_{n \in M} store_{n,v.interval,v.object} - store_{v.node,v.interval,v.object} < 1$
        $v.qos = -v.reward \cdot (1 - (\sum_{n \in M} store_{n,v.interval,v.object}$
                $- store_{v.node,v.interval,v.object}))$
    else
        $v.qos = 0$
```

Figure 7. The rounding down benefit function.

age. For nodes that never (in no interval) store this many objects, we also have to add $\beta(c_{max} - \bar{c}_n)$ creation cost, where $\bar{c}_n$ is the maximum number of objects stored in any interval on node n. This cost is then the cost for a feasible solution with the storage constraint enabled. The replica constraint is accounted for in an analogous way, even though the solution is searched for the object out of all nodes and intervals that is replicated the most times.

The routing knowledge affects only the input data and as such there is no need for any modifications in the rounding algorithm. The global/local knowledge, the activity history and the reactive property constraints are never violated by the rounding algorithm as proven below.

Proposition 1 *The rounding algorithm presented in Figure 5 produces a feasible solution to* MC-PERF *even when the the global/local knowledge, access history and/or reactive properties are included in the LP-problem.*

Proof for Proposition 1: The global/local knowledge, reactive, and the access history constraints all force $store_{nik}$ to become 0 for certain values of n, i and k. This means that the solution produced by the LP-relaxation has zeroes in the places where the constraints demand so. As the rounding algorithm never rounds up any zero value in the solution, these constraints are never violated. □

To decrease the running time of our algorithm, we have been experimenting with rounding entire sequences of consecutive intervals with the same fractional value as one unit. We have observed that this optimization decreases the running time of the rounding algorithm by over an order of magnitude with an increase to the cost of the solution that is less than 5%.