Bounds on the Replication Cost for QoS

Magnus Karlsson, Christos Karamanolis
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2003-156
July 23rd , 2003*

E-mail: karlsson@hpl.hp.com

data replication, QoS, placement algorithms, lower bounds, integer programming

Data replication is used extensively in wide-area distributed systems to achieve low data-access latency. Minimizing the cost of the resources used for replication is a key problem in these systems. The paper proposes a method to calculate lower bounds for the replication cost required to achieve certain QoS goals. We obtain bounds for the general case as well as for certain classes of replica placement heuristics. We observe that the cost of heuristics depends heavily on the workload and QoS goal. Based on these results, we discuss the inherent properties of heuristics that affect their cost and applicability to different environments.

* Internal Accession Date Only
Approved for External Publication

Bounds on the Replication Cost for QoS

Magnus Karlsson and Christos Karamanolis
Storage Systems Department, HP Laboratories
Contact author e-mail: karlsson@hpl.hp.com

Abstract

Data replication is used extensively in wide-area distributed systems to achieve low data-access latency. Minimizing the cost of the resources used for replication is a key problem in these systems. The paper proposes a method to calculate lower bounds for the replication cost required to achieve certain QoS goals. We obtain bounds for the general case as well as for certain classes of replica placement heuristics. We observe that the cost of heuristics depends heavily on the workload and QoS goal. Based on these results, we discuss the inherent properties of heuristics that affect their cost and applicability to different environments.

Keywords: Data Replication, QoS, Placement Algorithms, Lower Bounds, Integer Programming

1 Introduction

Data replication is used extensively in content delivery networks [4], web hosting services [10, 11], and distributed data repositories [1, 7, 12]. Data are replicated close to the point of access to achieve low access latency. A key challenge in these systems is to decide how many replicas to create and where to place them, in order to meet some required Quality of Service (QoS) goal with the lowest possible infrastructure cost. The QoS goal typically refers to a desirable maximum latency. There is a large number of different heuristics used to perform replica placement, including variations of caching [6], and centralized algorithms [10]; some take the QoS goal as an input while others provide an implicit QoS level that cannot be specified.

Given the inherently unpredictable nature of of the Internet (and of any large network), it is either impossible or too costly to guarantee that 100% of the requests in the system meet some reasonable latency threshold. Thus, we define a QoS goal as the fraction of requests that have to be served within a specified latency threshold. The QoS goals have to be satisfied over time and for potentially changing workloads.

Given a system and a workload, it is important to know what is the best replica placement, in terms of infrastructure cost, that meets the specified QoS goal. This knowledge is important for evaluating the overall cost of replica placement heuristics and the suitability of different heuristics for certain systems, workloads and QoS goals.

The paper formalizes the *minimal replication cost for QoS* (MC-QOS) problem as an integer programming (IP) problem and shows that it is NP-hard. Based on this formulation, the focus is on deriving lower bounds for the replication cost. First, we derive *general lower bounds*. That is, bounds that apply to any possible replica placement in the system, for a certain workload and QoS goal.

Such lower bounds provide important general conclusions about the cost of replica placement. However, we have observed in practice, that the placement cost varies widely among different classes of heuristics (e.g., caching [5], centralized greedy [10]), in different environments. Moreover, the cost of common heuristics is often many times that of general lower bounds. The main objective of the paper is to study the inherent cost of different heuristics and identify what are the best heuristics for certain systems, workloads and QoS goals. To achieve that, we capture a number of fundamental *heuristic properties* as additional constraints in the IP problem formulation. Examples of properties include reactive versus proactive placement and the distribution of knowledge about the placement of replicas on other nodes in the system. Using the extended problem definition, we derive lower bounds for classes of common heuristics. On the basis of experimental results, we reach conclusions about the classes of heuristics that are better-suited for specific examples of systems and workloads.

Given that there are no good approximation methods for MC-QOS, we obtain lower bounds by calculating the optimal solution to the linear relaxation of the problem. This lower bound must be close to the minimal cost of the corresponding IP problem in order to provide meaningful conclusions about the cost of heuristics. We propose a new rounding algorithm for the specific problem to derive feasible (integer) solutions from the lower bounds. These solutions are consistently closer to the lower bounds than the results of simple round-all-to-1 methods. Thus, the algorithm enables us to reach good tightness conclusions.

In summary, the paper makes four main contributions:

- Formalizes the MC-QOS problem for first time in the literature.

- Shows how to derive lower bounds for certain classes of heuristics in order to identify what heuristic properties are desirable (or should be avoided) to achieve a cost closer to the general lower bound.
- Introduces a domain-specific algorithm for obtaining the tightness of lower bounds.
- Reports on the lower bounds of some popular classes of heuristics, for real workloads and systems.

2 Related Work

Data replica placement 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 [6, 11, 16]. 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.* [10] 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 heuristics was compared against those lower bounds. In the case of MC-QoS, the goal is to *guarantee* certain QoS goals while minimizing the replication cost, in dynamic systems. We obtain both general lower bounds as well as lower bounds for specific classes of heuristics. The latter 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 [19]. They obtain 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, with a 100% performance guarantee. We are also concerned with minimizing the cost of resources used to do replication. However, in our case, the objective is to meet the QoS goal (percentage of accesses within latency threshold) and we formalize the problem for systems with *multiple* objects; we do not consider failures or network partitions. Our lower bounds match with theirs, when in our QoS definition 100% of the requests

have to satisfy the latency threshold, there are no failures and no consistency constraints, and there is only one object in the system.

MC-QOS is a dynamic variation of the classic SET-COVER problem [14]. 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 [8], but none of them captures QoS goals. Daskin and Owen have proposed an extension that captures QoS goals (similar to those in MC-QOS), but only in a static context [9]. There are no known approximation algorithms for either extension of SET-COVER.

3 Basic Problem Formulation

In this section, we define the *minimal replication cost for QoS* (MC-QOS) problem as an Integer Program (IP) optimization 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 that node. Requests originating from a node represent activities of users on that node. Users are *not* explicitly captured in our model. To keep the discussion simple, we assume a single user per node for the rest of the paper.

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 algorithm is executed on any node in the system. For example, a caching algorithm on a node is run upon every single object access initiated at that node; a complex centralized placement 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 [19] to cover *multiple objects*, as captured by cost function (1). 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. The notations are summarized in Table 1. For notational convenience, we write $\forall n$ to mean $\forall n \in N$, $\forall i$ to mean $\forall i \in I$ and $\forall k$ to signify $\forall k \in K$. The problem is formalized as follows.

Variable	Meaning
$\mathbf{store_{nik}}$	Whether node n stores object k during interval i.
$\mathbf{create_{nik}}$	Whether object k is created on node n during interval i.
$\mathbf{remove_{nik}}$	Whether object k is removed from node n during interval i.
$\mathbf{covered_{nik}}$	Whether node n can access object k within the latency threshold during interval i.
$demand_{nik}$	The number of read requests from node n to object k in interval i.
$dist_{nm}$	Whether node n can request objects from node m within the latency threshold (T_{lat}).
dec_{nm}	Whether node n uses information from node m to make its placement decision.
T_{lat}	The target latency threshold.
T_{qos}	The QoS goal – the fraction of accesses served in at most T_{lat}.
α	The unit cost for storage.
β	The unit cost for replica creation.

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

Minimize:

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

Subject to:

$$create_{nik}\geq store_{nik}-store_{n,i-1,k}\quad \forall n,i,k \tag{2}$$

$$store_{n,-1,k}=0\quad \forall n,k \tag{3}$$

$$covered_{nik}\leq\sum_{m\in N}dist_{nm}\cdot store_{mik}\quad \forall n,i,k \tag{4}$$

$$\sum_{i\in I}\sum_{k\in K}demand_{nik}\cdot covered_{nik}\geq T_{qos}\cdot\sum_{i\in I}\sum_{k\in K}demand_{nik}\quad \forall n \tag{5}$$

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

In the cost function, $store_{nik}$ is a binary variable capturing whether node n stores object k during interval i and $create_{nik}$ denotes whether object k was created on node n at the beginning of interval i, as defined by (2). 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. We consider the case when the system does not have any replicas to start with (3), however, this could be trivially modified to account for any initial placement.

The specified *QoS goal* is a fundamental concept in the definition of MC-QOS. We define this as the fraction of access requests that are guaranteed to be served within a latency threshold. For example, that 99% of the requests are guaranteed to be served in at most 1s. This is defined in (4) and (5). For notational convenience, we will refer to a guarantee that X% of requests will satisfy some

latency threshold, simply as *X % QoS*. Moreover, QoS can be defined for a single user or for an entire group of users (e.g., department), as well as for a single object or for a set of objects (e.g. all the files of a user). Constraint (5) defines this on a per user basis, overall objects, even though it can be easily modified to account for any of the other three definitions of QoS. Our QoS definition is equivalent to some definitions of availability [18]. The model and algorithm presented in this paper can be trivially modified to account for other QoS and performance metrics, e.g., average latency. The requirement is that the metric is monotonically increasing or monotonically decreasing with the addition of a replica and that it can be expressed as a linear constraint.

Formally, MC-QOS is the problem of *minimizing* cost function (1), given a workload, a system in which object replicas are placed, and a QoS goal that has to be met. We prove that this problem is NP-hard in Appendix A by showing that SET-COVER [15] reduces to MC-QOS.

4 Classes of Heuristics

The *general lower bound* obtained by applying an IP solver, or the much faster but approximate method of Section 5, to the IP formulation of MC-QOS provides important general conclusions about the cost of replica placement. However, the main objective of this paper is to study the inherent cost of different heuristics and identify what are the best heuristics for certain distributed systems, workloads and QoS goals. In order to accomplish this, we identify a set of properties that capture different techniques and assumptions used in heuristics. Most of them are captured as additional constraints in the IP formulation of MC-QOS.

4.1 Heuristic properties

The heuristic properties we consider are listed in Table 2. This is *not* intended to be an exhaustive list of properties describing any possible placement heuristic. The properties of Table 2 are described 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. Caching algorithms and many file allocation algorithms [3] are examples of such heuristics. There are two variations of this property. The first states that the

Heuristic properties	Lower bound for any heuristic...
storage constraint	that is using a fixed amount of storage in every interval.
replica constraint	that is using a fixed number of object replicas in every interval.
routing knowledge	where each node knows the contents of these other nodes.
decision knowledge	where each node use information from these nodes in the placement decision.
decision window	for which only objects accessed in this window can be placed.
reactive placement	that is reactive. Proactive placement implied otherwise.

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

amount of used storage is the same for every single node and interval. This corresponds, for example, to a system where the administrator configures every node with the same cache size. The second variation reflects the case where the amount of used storage varies between nodes (but not with time). This captures systems where larger caches are placed on strategic nodes with high traffic. The two variations are captured by the two versions of constraint (7) added to the IP formulation. 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. The solution to the problem (as discussed in Section 5) provides the minimal capacity that satisfies this constraint.

$$\sum_{k \in K} store_{nik} = \sum_{k \in K} store_{0,0,k} \quad \forall n, i \tag{7}$$

$$\sum_{k \in K} store_{nik} = \sum_{k \in K} store_{n,0,k} \quad \forall n, i \tag{7a}$$

Replica constraint. This constraint reflects heuristics that use a fixed number of replicas for each object throughout the execution. This is typical mainly of centralized heuristics [3, 10]. Again, there are two variations of the property (expressed by the two versions of constraint (8)). 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 is the case for heuristics that create more replicas for popular objects than for unpopular ones. Again, the solution to the problem provides the optimal replication factor that satisfies this constraint.

the IP/LP-solver provides the optimal replication factor that satisfies this constraint.

$$\sum_{n \in N} store_{nik} = \sum_{n \in N} store_{n,0,0} \quad \forall i, k \tag{8}$$

$$\sum_{n \in N} store_{nik} = \sum_{n \in N} store_{n,0,k} \quad \forall i,k \tag{8a}$$

Routing knowledge. With some heuristics, such as caching, a node has no knowledge of what other nodes store. Upon a miss, the node has to fetch the object from some origin location. With cooperative caching [17], a node knows what objects are stored by some other, nearby nodes and can fetch them from there. Similarly, many centralized heuristics [10] fetch the closest replica from any node in the system. We call this property *routing knowledge* and we assume that the origin node is further away than the latency threshold. There is no need to introduce a new constraint in the IP-model, as this property can be represented by modifying $dist_{nm}$, after it is filled in with the values indicating if node m can serve requests from node m within T_{lat}. If node n has no routing knowledge of node m, $dist_{nm}$ is set to 0, even if the two nodes are within the threshold, to reflect the lack of routing knowledge.

Decision knowledge. When a heuristic makes a placement decision for a specific node, it takes into account accesses that originated at the local node and potentially other nodes in the system. Many decentralized algorithms make decisions based only on accesses that originated from the node they run on. This means that the placement cannot change, unless some accesses occur on that node. On the other hand, 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. To represent these two cases and anything in between, we use a matrix dec_{nm} to indicate that knowledge of demand originating at node m, even if not directed to node n, is used to decide for the placement of objects on n. For some node n, we call all nodes m such that $dec_{nm} = 1$ to be in the the *sphere of knowledge* of node n.

If, for a given node n, there are no accesses from any node m such as $dec_{nm} = 1$, then no removals or creations of objects are allowed to occur on node n. For creations, this is stated by constraint (9), as $create_{nik}$ is already defined by (2) and the minimization of (1). For removals, we define variable $remove_{nik}$ to be 1 only if object k was removed from node n in interval i. Since $remove_{nik}$ is not minimized in the cost function, as $create_{nik}$ is, we use three additional constraints in order to completely define it, (11) - (13). Also, $remove_{nik}$ is constrained similarly to $create_{nik}$ using (14).

$$\sum_{m \in N} \sum_{l \in K} demand_{mil} \cdot dec_{nm} - create_{nik} \geq 0 \quad \forall n,i,k \tag{9}$$

$$remove_{nik} \in \{0,1\} \quad \forall n,i,k \tag{10}$$

$$remove_{nik} \geq store_{n,i-1,k} - store_{nik} \quad \forall n,i,k \tag{11}$$

$$remove_{nik} \leq store_{n,i-1,k} \quad \forall n,i,k \tag{12}$$

$$remove_{nik} \leq 1 - store_{nik} \quad \forall n,i,k \tag{13}$$

$$\sum_{m \in N} \sum_{l \in K} demand_{mil} \cdot dec_{nm} - remove_{nik} \geq 0 \quad \forall n,i,k \tag{14}$$

Decision window. When a heuristic decides to place some replica on node n at the beginning of interval i, it always places objects that are accessed in i or have been accessed before, in the system. This property captures the "window" of intervals considered when deciding what objects a heuristic is allowed to place. The corresponding constraint captures two cases: *single* or *all* intervals. In the former case, when deciding about placement at the beginning of interval i in node n, a heuristic can consider only objects referenced within the node's sphere of knowledge during i. Caching is an example of such a heuristic. In the latter case, a heuristic can consider any objects referenced within the node's sphere of knowledge, throughout the execution, up to and including i. Note, that it does not make sense to place an object before it is ever accessed, because this would increase storage cost unnecessarily. So, even if oracle-like heuristics could place objects well before they are ever accessed, the decision window constraint provides a lower bound for them. The basic definition covers the *all* intervals case. We introduce an additional constraint to capture the *single* interval case.

$$create_{nik} \leq \sum_{m \in N} demand_{nik} \cdot dec_{nm} \quad \forall n,i,k \tag{15}$$

Reactive placement. The basic IP formulation, including the above additional constraints, implicitly refers to solutions of *proactive* heuristics. That is, heuristics with a decision window that covers not only previous intervals but also the current interval, at the beginning of which the placement decision is made. 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. Reactive heuristics are captured by further constraining the decision knowledge or decision window constraints (we can have one of them at the time). The new versions of the constraints state that if no access occurred during the previous interval in the sphere of knowledge of node n (for any object or for a specific object, respectively), no

placement change can happen in n during the current interval.

$$\sum_{m \in N} \sum_{l \in K} demand_{m,i-1,l} \cdot dec_{nm} - create_{nik} \geq 0 \quad \forall n,i,k \tag{9a}$$

$$\sum_{m \in N} \sum_{l \in K} demand_{m,i-1,l} \cdot dec_{nm} - remove_{nik} \geq 0 \quad \forall n,i,k \tag{14a}$$

$$create_{nik} \leq \sum_{m \in N} demand_{n,i-1,k} \cdot dec_{nm} \quad \forall n,i,k \tag{15a}$$

$$demand_{n,-1,k} = 0 \quad \forall n,k \tag{16}$$

The general lower bound for the basic definition of MC-QOS is the solution obtained without setting any of the additional constraints of this section.

4.2 Classes of heuristics

By introducing one or more of the above constraints to the formulation, we can argue about the lower bounds for classes of heuristics. We give here examples of such classes and obtain lower bounds for them in Section 6.2. In all cases, the additional properties result in bounds for the specific class that are higher or equal to the general lower bound of the problem.

The solutions due to *caching* algorithms, without prefetching and cooperation, are constrained by the following property settings: 1) storage constrained, 2) no routing knowledge to other nodes; 3) reactive; 4) single interval decision window; 5) local decision knowledge, 6) evaluated after every single access. Similarly, lower bounds for *cooperative caching* algorithms can be computed by including all the constraints for caching but without local-only routing knowledge.

One large class of completely *decentralized algorithms* [5] can be modeled by setting the routing and decision knowledge to be only local and by setting the reactive property and the single interval decision window. By extending the routing knowledge beyond local, we can capture the subset of *decentralized algorithms with global routing knowledge* [4, 11].

4.3 Evaluation interval values

An important parameter of the problem formulation is the *evaluation interval* used to calculate the lower bound. Intuitively, to obtain the lower bound, the evaluation interval should be as small as

possible for the solutions considered. However, it makes no sense to set the evaluation interval to anything smaller than the execution time of one evaluation round of the targeted heuristics.

Let P_{min} be the smallest time between two consecutive rounds of placement evaluation performed by the heuristic. We show in Theorem 2 in Appendix B that an evaluation interval of $\Delta = P_{min}/2$ provides the lower bound for all possible solutions from that heuristic, if there is a P in the system such that $P_{min} \leq P < 2 \cdot P_{min}$; otherwise, if $P \geq 2P_{min}$ for all $P \neq P_{min}$. In the latter case, P_{min} can itself be used to obtain a lower bound. The same observations apply to the case where the evaluation is defined as the time between two consecutive accesses on a single node (e.g., caching heuristics). In that case, P_{min} is the minimum time between any two accesses within the sphere of knowledge of any node.

The above equations provide the evaluation interval Δ for the lower bound, only if the decision knowledge constraint is activated; that is, a placement evaluation is allowed to occur only if some access occurs in the system. When that constraint is not activated, the lower bound decreases monotonically for smaller evaluation intervals. In fact, as Δ approaches 0, the lower bound converges to a value that is the lowest possible for any Δ. In general, calculating this lower bound with our approach is infeasible, due to the small Δ values that need to be considered until convergence is observed. Obtaining lower bounds without the decision knowledge constraint is an open issue. However, in practice, minimum interval durations can be obtained from the running time of the targeted heuristics.

5 Obtaining a Lower Bound

A *tight lower bound* can be computed by solving the IP problem exactly (using an IP solver), but 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. Therefore, 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 round-

ing algorithm. It has been shown that SET-COVER cannot be approximated with a constant approximation factor [15], and there are no known approximation methods for either dynamic SET-COVER or static SET-COVER with QoS (both can be reduced to instances of MC-QOS). These are worst-case estimates for approximation algorithms and are valid for any input data. We obtain, here, a lower bound for MC-QOS using linear relaxation and a rounding algorithm, as it is applicable to instances of the problem (specific input data) and, in general, provides lower bounds that are tighter that the theoretical worst-case.

5.1 Linear lower bound

The *lower bound* to MC-QOS is the solution to the linear relaxation of the IP problem. The linear relaxation (LP) is obtained by letting the binary decision variables in the problem formulation have fractional values. The cost of this solution is denoted $cost_{LP}$ and can be be computed with an LP-solver in polynomial time. By definition, $cost_{LP} \leq cost_{IP}$, where $cost_{IP}$ is the cost of the tight lower bound (the unknown solution to the IP problem). Ideally, we would like to know the *tightness* of the lower bound, defined as $(cost_{IP} - cost_{LP})/cost_{IP}$. Since we don't know $cost_{IP}$, we obtain instead an upper bound on the tightness by producing one feasible, binary solution to the IP problem, by rounding the fractional values of the LP solution to 0 or 1. The cost of that solution is denoted $cost_{feas}$. As $cost_{feas} \geq cost_{IP}$, the fraction $(cost_{feas} - cost_{LP})/cost_{LP}$ provides an upper bound on the tightness of the lower bound, which we call *perceived tightness.*

Unfortunately, we cannot use standard rounding techniques such as "round-to-1", "round-to-0" or "round to nearest 0 or 1" to obtain the feasible solution. The first often provides bad perceived tightness; the other two generally provide infeasible solutions. In the following section, we introduce a new *rounding algorithm*, for our problem, that produces feasible solutions with costs close to $cost_{LP}$. As a result, we can argue that the linear relaxation approach provides close-to-tight lower bounds for instances of MC-QOS (see Section 6.1).

5.2 Rounding algorithm

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.

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 1, 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.

The algorithm is outlined in Figure 2. 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.

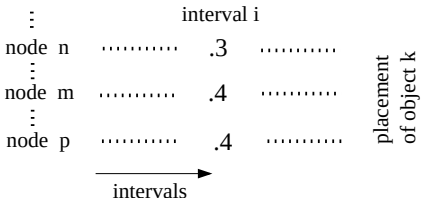

Figure 1: Example of reward versus QoS impact. Nodes *n*, *m* and *p* are within the latency threshold from each other. According to the LP solution, they store (fractional values of) object *k* during interval *i*; there are no other nodes within the latency threshold from them that store *k* during *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 for interval *i*. Reward is the impact to QoS if all fractional values are considered 0.

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 $store_{n,i-1,k}$ 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 $store_{nik}$. More details can be found in Appendix C. The impact on replica creation cost due to *rounding down* a value is calculated with a process which is symmetric to the above.

The algorithm of Figure 2 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.

The rounding algorithm is described in full detail in Appendix C, together with a proof of its correctness (always resulting in a feasible solution).

5.3 Lower bounds for classes of heuristics

In the previous section, we discussed how to obtain lower bounds for the general MC-QOS problem. The rounding algorithm can be used, with some extensions, to ontain feasible solutions for classes

```
cost = cost of LP solution // lower bound
qos = QoS threshold
V ← set of all store_nik with fractional values in LP solution
while V ≠ ∅
    forall v ∈ V
        calculate v.cost, v.qos and v.reward for rounding up v
    find v ∈ V with lowest v.cost/v.reward
    round up v
    cost = cost + v.cost
    qos = qos + v.qos
    V ← V − {v}
    repeat
        forall v ∈ V
            calculate v.cost, v.qos and v.reward for rounding down v
        C ← values that can be rounded down without violating
            the QoS constraints
        if C ≠ ∅
            find v ∈ C with v.reward == 0 and v.cost < 0
            if no v found
                find v ∈ C with highest v.cost/v.reward
            round down v
            cost = cost + v.cost
            qos = qos + v.qos
            V ← V − {v}
    until C = ∅
output cost
```

Figure 2: The rounding algorithm in pseudo-code. As a notational convenience, *v.cost*, *v.qos* and *v.reward* represent the cost impact, QoS impact and reward, respectively, for rounding value *v* in the LP solution.

of heuristics. That is, when one or more of the constraints of Section 4 are enabled in the problem definition.

When the *storage constraint* is enabled, we search the solution for the node that stored the most 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 storage. For nodes that never (in no interval) store this many objects, we also have to add $\beta(c_{max} - \overline{c}_n)$ creation cost, where $\overline{c}_n$ 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* constraint affects only the input data and as such there is no need for any modifications in the rounding algorithm. The *decision knowledge*, *decision window* and *reactive property* constraints are never violated by the rounding algorithm, as the LP solution has zeroes in

the places where the constraints demand so, and the rounding algorithm never rounds up a zero entry. The algorithm with the additions outlined here is described in detail in Appendix C.

6 Experimental Results

This section presents experimental results and conclusions from applying the proposed method to MC-QoS. We show that: 1) our rounding algorithm provides feasible solutions with cost that is close to the lower bound; 2) our method scales sufficiently well for realistic systems and workloads; 3) the lower bounds obtained for classes of heuristics provide evidence about the heuristic properties that affect their cost in different environments.

For our experiments, we simulate real systems and workloads. We use a network topology based on an actual Internet AS-level topology [13], with 20 nodes. Each node in the topology is a candidate location for placing object replicas. A single hop in the topology takes between 5 ms and 15 ms. For the QoS goals considered, the latency threshold is set to 15 ms.

We use two workloads derived from actual web traces, with different object popularity distributions: **(w1)** with a heavy-tailed Zipf distribution with many unpopular objects; **(w2)** with a short-tailed Zipf distribution with only popular objects. The nodes are all highly active and all generate requests to the 1,000 objects we consider. In total, we have 300K requests in w1 and 16M requests in w2. The most popular object has 36K accesses in both workloads. The least popular object has just 1 access in w1 and 8.5K accesses in w2. The duration of both workloads is 1 full day and the evaluation interval is set to 1 hour. We consider all objects to be 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]. QoS is specified per-user, over-all objects.

The number of constraints and variables in the MC-QoS formulation 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. The computational complexity of the rounding algorithm is $O(V|N||K|)$, where V is the number of fractional values in the solution. On a workstation with a 360MHz PA-RISC processor, the running time of CPLEX on the instances of the LP problem studied in this section

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

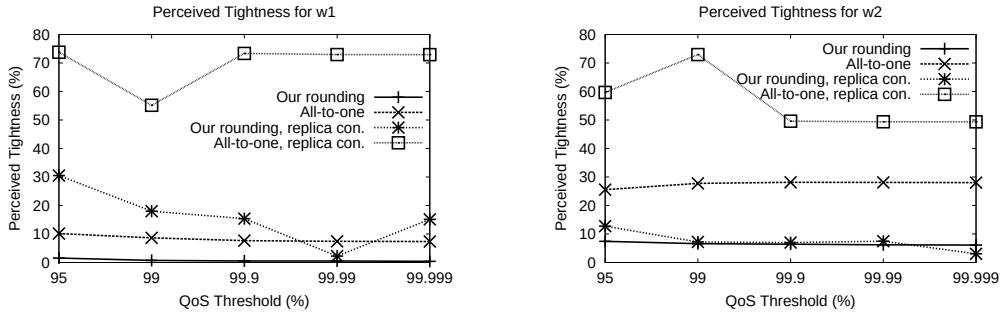

Figure 3: The perceived tightness with our rounding algorithm and an all-to-one rounding scheme (lower is better).

ranges from 1 minute to 3 days (with all constraints enabled). The running time of the rounding algorithm is less than 30 minutes, even in the worst case.

6.1 Tightness

All-to-1 rounding provides good lower-bound estimates when there are few fractional values in the solution, but not in general. The worst-case perceived tightness of the all-to-1 rounding scheme for the general MC-QOS problem is $|N| - 1$. This occurs when the LP solution is $store_{n,0,0} = 1/|N|$, for all $n \in N$, but the optimal integer solution is $store_{n,0,0} = 1$ for one n and 0 for the rest. On the other hand, we do not know what the worst-case perceived tightness is for our rounding algorithm. Thus, we compare the two approaches experimentally.

Figure 3 shows the perceived tightness of the two rounding schemes for w1 and w2. As shown, the perceived tightness of the all-to-1 scheme is sometimes poor and sometimes good. Our algorithm, on the other hand, provides good feasible solutions in all cases except in the case of w1, with replica constraint enabled and a low QoS goal. We are currently investigating if this is because $cost_{LP}$ is far from $cost_{IP}$ or not. Even then, our algorithm does at least two times better than the all-to-1 approach. The results from the other heuristic properties are not shown due to space constraints. For the other cases, our algorithm consistently provides a perceived tightness below 10%, while the all-to-1 rounding scheme varies between 5% and 80% for our workloads.

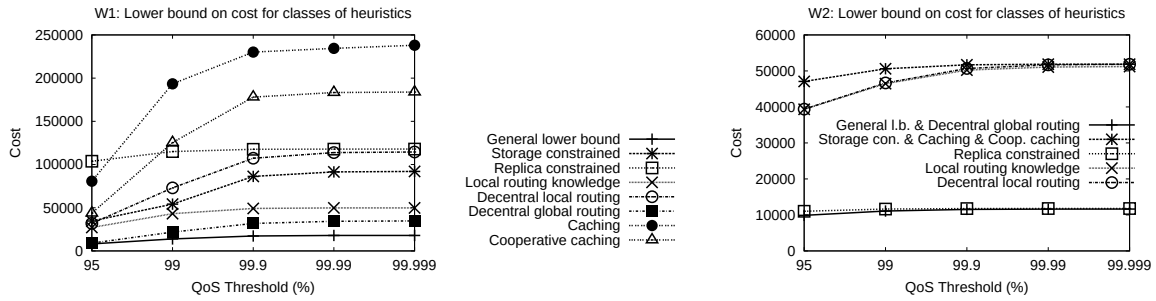

Figure 4: The lower bound for various heuristic properties as a function of QoS for w1 and w2.

6.2 Lower bounds

In this section, we present lower-bound results for the general problem as well as for the classes of heuristics discussed in Section 4.1. Figure 4 presents these results for the two workloads we consider.

Replica-constrained heuristics have about 7 times higher cost than the general lower bound, for workload w1. The reason is that even though w1 has a heavy-tailed object popularity distribution, due to this constraint, unpopular objects have as many replicas as popular ones. On the other hand, for w2, replica constraints have minimal impact to cost—their lower bound nearly overlaps with the general lower bound. This is because w2 includes only objects that are accessed often. Thus, creating the same number of replicas for every single object works well.

In the case of w1, *storage-constrained heuristics* have at most 5 times the cost of the general lower bound, as all nodes generate many requests and there are few popular objects. If there was a large skew in the distribution of requests in the system, then the lower bound for storage-constrained heuristics would increase considerably. For w2, the storage constraint results in costs that are always more than 5 times the general lower bound, as there is a lot to gain by storing just a subset of the objects, in this case.

Local routing knowledge has only a small impact on cost, in the case of w1, as every node stores roughly the same objects (there are only few popular objects). This is not the case in w2—accessing objects from other nodes would provide much lower cost. Using local routing knowledge in *decentralized heuristics* (see Section 4.2) results in considerably higher costs for w1. On the other hand, *reactive placement* and *single interval window* constraints have very small impact on cost, for either of the two workloads, when activated in decentralized heuristics. The workloads are

relatively static and the intervals are long. Thus, those two primitives do not constraint the solution substantially. It is the global routing knowledge that makes a difference for decentralized heuristics.

Both classes of *caching heuristics* (local and cooperative)[2] have by far the poorest cost for high QoS levels, in the case of workload w1. Their lower bounds are 10 to 14 times the cost of the general lower bound. The reason is that the combination of all the heuristic properties involved in caching are too restrictive for w1. In the case of workload w2, the lower bounds for caching and cooperative caching overlap with the bound for storage constrained heuristics, at 5 times the cost of the general bound. The limiting factor for this workload is the storage constraint property of caching.

7 Conclusions

The paper proposes solving the linear relaxation of MC-QOS to obtain lower cost bounds and introduces a domain-specific rounding algorithm to check the tightness of those bounds. The method is applied to obtain lower bounds for constraint versions of MC-QOS that capture a range of properties of placement heuristics. Based on experimental evaluations, we argue that the cost of heuristics depends heavily on the workload and QoS goal. We provide examples of the effects on cost of the heuristic properties we identified, for two different workloads and a range of QoS goals.

References

[1] R. Braynard, D. Kostic, A. Rodriguez, J. Chase, and A. Vahdat. Opus: an Overlay Peer Utility Service. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 167–178, June 2002.

[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 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.

[2] We consider here delayed caching algorithms [5] with an evaluation interval of 1 hour.

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

[7] 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*, pages 190–201, November 2000.

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

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] H. Yu and A. Vahdat. The Costs and Limits of Availability for Replicated Services. In *18th Symposium on Operating Systems Principles (SOSP)*, pages 29–42, October 2001.

[19] 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

Note to the reviewers: The contents of this appendix will be published as a technical report. It is included here for your convenience.

In section A of the Appendix, we prove that the MC-QOS 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-QOS.

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

Proof for Theorem 1: We show this by a polynomial time reduction of SET-COVER [15] to MC-QOS. Let each candidate set in SET-COVER correspond to a unique node in MC-QOS; this set of nodes is denoted C. Let each element in SET-COVER correspond to a unique node in MC-QOS; this set of nodes is denoted E. Sets C and E are disjoint. Thus, the set of nodes considered in MC-QOS 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-QOS, 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-QOS 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-QOS *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.

$$store_{nik} \leq 1 \quad \forall n \in N, i \in I, k \in K \tag{6a}$$

$$store_{nik} \geq 0 \quad \forall n \in N, i \in I, k \in K \tag{6b}$$

$$covered_{nik} \leq 1 \quad \forall n \in N, i \in I, k \in K \tag{6c}$$

$$covered_{nik} \geq 0 \quad \forall n \in N, i \in I, k \in K \tag{6d}$$

$$create_{nik} \leq 1 \quad \forall n \in N, i \in I, k \in K \tag{6e}$$

$$create_{nik} \geq 0 \quad \forall n \in N, i \in I, k \in K \tag{6f}$$

$$remove_{nik} \leq 1 \quad \forall n \in N, i \in I, k \in K \tag{10a}$$

$$remove_{nik} \geq 0 \quad \forall n \in N, i \in I, k \in K \tag{10b}$$

As discussed in Section 5, the approach used to obtain lower bounds for MC-QOS and its extensions is a combination of linear relaxation and a rounding algorithm. We propose a greedy rounding

```
function calculate_round_up_benefit(v)
    if (v.succ ≥ v.value ≥ v.prev)
        v.cost = β · (1 − v.succ)
    else if (v.succ < v.value < v.prev)
        v.cost = β · (1 − v.prev)
    else if (v.succ ≥ v.value ∧ v.prev > v.value)
        v.cost = β · (1 − v.prev − (v.succ − v.value))
    else if (v.succ ≤ v.value ∧ v.prev < v.value)
        v.cost = β · (1 − v.value)
    v.cost = v.cost + α · (1 − v.value)
    // reward only from nodes within the latency threshold
    // that are not fully covered by another node
    M ← {n ∈ N : dist_{n,v.node} = 1 ∧
            Σ_{m∈N} ⌊store_{m,v.interval,v.object}⌋ · dist_{nm} = 0}
    v.reward = Σ_{n∈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 ← {n ∈ N : dist_{n,v.node} = 1}
    if Σ_{n∈M} store_{n,v.interval,v.object} < 1
        v.qos = v.reward · (1 − Σ_{n∈M} store_{v.node,v.interval,v.object})
    else
        v.qos = 0
```

Figure 5: The rounding up benefit function.

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 5, 6 and 7. The intuition behind the algorithm and how it works have already been described in Sections 5.2. In this appendix, we focus only on features of the algorithm not discussed there.

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 main body of the rounding algorithm is described in Figure 7. The algorithm consists of a loop during which one value is chosen to be rounded up and then zero or more other values are rounded down. The values to be rounded up and down are chosen on the basis of the $cost/reward$ ratio of all fractional values in the solution. The algorithm is augmented with two additional modules that extensions due to the storage and replica constraints in the extended version of MC-QOS.

Figures 5 and 6 show the two functions that calculate the cost ($v.cost$), QoS impact ($v.qos$) and QoS reward ($v.reward$) of rounding a value up or down, respectively. The rounding up case is explained in Section 5.2. Thus, we focus here on the symmetric case of rounding down. There are

```
function calculate_round_down_benefit(v)
    if (v.succ ≥ v.value ≥ v.prev)
        v.cost = −β · v.prev
    else if (v.succ < v.value < v.prev)
        v.cost = −β · v.succ
    else if (v.succ ≥ v.value ∧ v.prev > v.value)
        v.cost = −β · v.value
    else if (v.succ ≤ v.value ∧ v.prev < v.value)
        v.cost = −β · (v.succ − (v.value − v.prev))
    v.cost = v.cost − α · v.value
    // reward only from nodes within the latency threshold
    // that are not fully covered by another node
    M ← {n ∈ N : dist_{n,v.node} = 1 ∧
            Σ_{m∈N} ⌊store_{m,v.interval,v.object}⌋ · dist_{nm} = 0}
    v.reward = −Σ_{n∈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 ← {n ∈ N : dist_{n,v.node} = 1}
    if Σ_{n∈M} store_{n,v.interval,v.object} − store_{v.node,v.interval,v.object} < 1
        v.qos = −v.reward · (1 − (Σ_{n∈M} store_{n,v.interval,v.object}
                − store_{v.node,v.interval,v.object}))
    else
        v.qos = 0
```

Figure 6: The rounding down benefit function.

the following cases to consider (reflected in "if" statements in Figure 6): 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*.

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 storage. For nodes that never (in no interval) store this many objects, we also have to add $\beta(c_{max} - \overline{c}_n)$ creation cost, where $\overline{c}_n$ 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 decision knowledge, the decision window and the reactive property constraints are never violated by the rounding algorithm as proven below.

Proposition 1 *The rounding algorithm presented in Figure 7 produces a feasible solution to* MC-QOS *even when the the decision knowledge, interval window and/or reactive properties are included in the LP-problem.*

Proof for Proposition 1: The decision knowledge, reactive, and the interval window 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%.

$cost =$ 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 $\overline{c}_n$, the max no. of objects stored on node n
        during any interval
    $\forall n \in N; cost = cost + \beta \cdot (c_{max} - \overline{c}_n)$
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 $\overline{c}_k$, the max no. of replicas of object k
        during any interval
    $\forall k \in K; cost = cost + \beta \cdot (c_{max} - \overline{c}_k)$
output *cost*

Figure 7: The rounding algorithm used to find the tightness of the lower bound.