Impact of QoS on Replica Placement in Tree Networks

Anne Benoit Veronika Rehn Yves Robert

December 2006

LIP Research Report RR-2006-48

Abstract

This paper discusses and compares several policies to place replicas in tree networks, subject to server capacity and QoS constraints. The client requests are known beforehand, while the number and location of the servers are to be determined. We study three strategies. The first two strategies assign each client to a unique server while the third allows requests of a client to be processed by multiple servers. The main contribution of this paper is to assess the impact of QoS constraints on the total replication cost. In this paper, we establish the NP-completeness of the problem on homogeneous networks when the requests of a given client can be processed by multiple servers. We provide several efficient polynomial heuristic algorithms for NP-complete instances of the problem. These heuristics are compared to the optimal solution provided by the formulation of the problem in terms of the solution of an integer linear program.

1 Introduction

This paper deals with the problem of replica placement in tree networks with Quality of Service (QoS) guarantees. Informally, there are clients issuing several requests per time-unit, to be satisfied by servers with a given QoS. The clients are known (both their position in the tree and their number of requests), while the number and location of the servers are to be determined. A client is a leaf node of the tree, and its requests can be served by one or several internal nodes. Initially, there are no replicas; when a node is equipped with a replica, it can process a number of requests, up to its capacity limit (number of requests served by time-unit). Nodes equipped with a replica, also called servers, can only serve clients located in their subtree (so that the root, if equipped with a replica, can serve any client); this restriction is usually adopted to enforce the hierarchical nature of the target application platforms, where a node has knowledge only of its parent and children in the tree. Every client has some QoS constraints: its requests must be served within a limited time, and thus the servers handling these requests must not be too far from the client.

The rule of the game is to assign replicas to nodes so that some optimization function is minimized and QoS constraints are respected. Typically, this optimization function is the total utilization cost of the servers. In this paper we study this optimization problem, called Replica

PLACEMENT, and we restrict the QoS in terms of number of hops (QoS = distance to server). This means for instance that the requests of a client who has a QoS range of $\mathsf{qos} = 5$ must be treated by one of the first five internal nodes on the path from the client up to the tree root.

We point out that the distribution tree (clients and nodes) is fixed in our approach. This key assumption is quite natural for a broad spectrum of applications, such as electronic, ISP, or VOD service delivery. The root server has the original copy of the database but cannot serve all clients directly, so a distribution tree is deployed to provide a hierarchical and distributed access to replicas of the original data. On the contrary, in other, more decentralized, applications (e.g. allocating Web mirrors in distributed networks), a two-step approach is used: first determine a "good" distribution tree in an arbitrary interconnection graph, and then determine a "good" placement of replicas among the tree nodes. Both steps are interdependent, and the problem is much more complex, due to the combinatorial solution space (the number of candidate distribution trees may well be exponential).

Many authors deal with the REPLICA PLACEMENT optimization problem. Most of the papers do not deal with QoS but instead consider average system performance such as total communication cost or total accessing cost. Please refer to [1] for a detailed description of related work with no QoS contraints.

Cidon et al [4] studied an instance of REPLICA PLACEMENT with multiple objects, where all requests of a client are served by the closest replica (*Closest* policy). In this work, the objective function integrates a communication cost, which can be seen as a substitute for QoS. Thus, they minimize the average communication cost for all the clients rather than ensuring a given QoS for each client. They target fully homogeneous platforms since there are no server capacity constraints in their approach. A similar instance of the problem has been studied by Liu et al [8], adding a QoS in terms of a range limit (QoS=distance), and whose objective is to minimize the number of replicas. In this latter approach, the servers are homogeneous, and their capacity is bounded. Both [4, 8] use a dynamic programming algorithm to find the optimal solution.

Some of the first authors to introduce actual QoS constraints in the problem were Tang and Xu [11]. In their approach, the QoS corresponds to the latency requirements of each client. Different access policies are considered. First, a replica-aware policy in a general graph with heterogeneous nodes is proven to be NP-complete. When the clients do not know where the replicas are (replica-blind policy), the graph is simplified to a tree (fixed routing scheme) with the *Closest* policy, and in this case again it is possible to find an optimal dynamic programming algorithm. In [12], Wang et al deal with the QoS aware replica placement problem on grid systems. In their general graph model, QoS is a parameter of communication cost. Their research includes heterogeneous nodes and communication links. A heuristic algorithm is proposed and compared to the results of Tang and Xu [11].

Another approach, this time for dynamic content distribution systems, is proposed by Chen et al [3]. They present a replica placement protocol to build a dissemination tree matching QoS and server capacity constraints. Their work focuses on Web content distribution built on top of peer-to-peer location services: QoS is defined by a latency within which the client has to be served, whereas server capacity is bounded by a fan-out-degree of direct children. Two placement algorithms (a native and a smart one) are proposed to build the dissemination tree over the physical structure.

In [1] we introduced two new access polices besides the *Closest* policy. In the first one, the restriction that all requests from a given client are processed by the same replica is kept, but client requests are allowed to "traverse" servers in order to be processed by other replicas located higher in

the path (closer to the root). This approach is called the *Upwards* policy. In the second approach, access constraints are further relaxed and the processing of a given client's requests can be split among several servers located in the tree path from the client to the root. This policy with multiple servers is called *Multiple*.

In this paper we study the impact of QoS constraints on these three policies. On the theoretical side we prove the NP-completeness of *Multiple*/Homogeneous instance with QoS constraints, while the same problem was shown to be polynomial without QoS [1]. This result shows the additional combinatorial difficulties which we face when enforcing QoS constraints. On the practical side, we propose several heuristics for all policies. We compare them through simulations conducted for problem instances with different ranges of QoS constraints. We are also able to assess the absolute performance of the heuristics, by comparing them to the optimal solution of the problem provided by a formulation of the Replica Placement problem in terms of a mixed integer linear program. The solution of this program allows us to build an optimal solution 3 for reasonably large problem instances.

The rest of the paper is organized as follows. Section 2 explains the framework and the different access policies in more details. Complexity results are presented in Section 4. Section 5 describes the proposed heuristics, whereas experimental results can be found in Section 6. Section 7 summarizes our contributions.

2 Framework and Placement Strategies

We consider a distribution tree $\mathcal{T}$ whose nodes are partitioned into a set of clients $\mathcal{C}$ and a set of nodes $\mathcal{N}$. The set of tree edges is denoted as $\mathcal{L}$. The clients are leaf nodes of the tree, while $\mathcal{N}$ is the set of internal nodes. A *client* $i \in \mathcal{C}$ is making r_i requests per time unit to a database, with a QoS qos_i: the database must be placed not further than qos_i hops on the path from the client to the root.

A node $j \in \mathcal{N}$ may or may not have been provided with a replica of the database. A node j equipped with a replica (*i.e.* j is a server) can process up to W_j requests per time unit from clients in its subtree. In other words, there is a unique path from a client i to the root of the tree, and each node in this path is eligible to process some or all the requests issued by i when provided with a replica. We denote by $R \subseteq \mathcal{N}$ the set of replicas, and $\mathsf{Servers}(i) \subseteq R$ is the set of nodes which are processing requests from client i. The number of requests from client i satisfied by server s is $r_{i,s}$, and the number of hops between i and $j \in \mathcal{N}$ is denoted by $d(i,j)$. Two constraints must be satisfied:

- Server capacity: $\forall s \in R, \quad \sum_{i \in \mathcal{C} | s \in \mathsf{Servers}(i)} r_{i,s} \leq \mathsf{W}_s$
- QoS constraint: $\forall i \in \mathcal{C}, \forall s \in \mathsf{Servers}(i), \quad d(i,s) \leq \mathsf{qos}_i$

Let r be the root of the tree. If $j \in \mathcal{N}$, then $\mathsf{children}(j)$ is the set of children of node j. If $k \neq r$ is any node in the tree (leaf or internal), $\mathsf{parent}(k)$ is its parent in the tree. If $l : k \rightarrow k' = \mathsf{parent}(k)$ is any link in the tree, then $\mathsf{succ}(l)$ is the link $k' \rightarrow \mathsf{parent}(k')$ (when it exists). Let $\mathsf{Ancestors}(k)$ denote the set of ancestors of node k, i.e. the nodes in the unique path that leads from k up to the root r (k excluded). If $k' \in \mathsf{Ancestors}(k)$, then $\mathsf{path}[k \rightarrow k']$ denotes the set of links in the path from k to k'; also, $\mathsf{subtree}(k)$ is the subtree rooted in k, including k.

The objective function for the Replica Placement problem is defined as: Min $\sum_{s \in R} \mathsf{W}_s$. When the servers are homogeneous, *i.e.* $\forall s \in \mathcal{N}, \mathsf{W}_s = \mathsf{W}$, the optimization problem reduces to finding a minimal number of replicas. This problem is called Replica Counting.

We consider three access policies in this paper. The first two are single server strategies, i.e. each client is assigned a single server responsible for processing all its requests. The *Closest* policy is the most restricted one: the server for client i is enforced to be the first server that can be found on the path from i upwards to the root. Relaxing this constraint leads to the *Upwards* policy. Clients are still assigned to a single server, but their requests are allowed to traverse one or several servers on the way up to the root, in order to be processed by another server closer to the root. The third policy is a multiple server strategy and hence a further relaxation: a client i may be assigned a set of several servers. Each server $s \in \mathsf{Servers}(i)$ will handle a fraction $r_{i,s}$ of requests. Of course $\sum_{s \in \mathsf{Servers}(i)} r_{i,s} = r_i$. This policy is referred to as the *Multiple* policy.

3 Linear programming formulation

In this section, we express the Replica Placement optimization problem in terms of an integer linear program. We deal with the most general instance of the problem on a heterogeneous tree, including QoS constraints, and bounds on resource usage (both server and link capacities). We derive a formulation for each of the three server access policies, namely *Closest*, *Upwards* and *Multiple*. This is an important extension to a previous formulation due to [7].

While there is no efficient algorithm to solve integer linear programs (unless P=NP), this formulation is extremely useful as it leads to an absolute lower bound: we solve the integer linear program over the rationals, using standard software packages [2, 6]. Of course the rational solution will not be feasible, as it assigns fractions of replicas to server nodes, but it will provide a lower bound on the storage cost of any solution.

3.1 Single server

We start with single server strategies, namely the *Upwards* and *Closest* access policies. We need to define a few variables:

Server assignment

- x_j is a boolean variable equal to 1 if j is a server (for one or several clients)
- $y_{i,j}$ is a boolean variable equal to 1 if $j = \mathsf{server}(i)$
- If $j \notin \mathsf{Ancestors}(i)$, we directly set $y_{i,j} = 0$.

Link assignment

- $z_{i,l}$ is a boolean variable equal to 1 if link $l \in \mathsf{path}[i \to r]$ is used when client i accesses its server $\mathsf{server}(i)$
- If $l \notin \mathsf{path}[i \to r]$ we directly set $z_{i,l} = 0$.

The objective function is the total storage cost, namely $\sum_{j \in \mathcal{N}} \mathsf{sc}_j x_j$. We list below the constraints common to the *Closest* and *Upwards* policies: First there are constraints for server and link usage:

- Every client is assigned a server: $\forall i \in \mathcal{C}, \sum_{j \in \mathsf{Ancestors}(i)} y_{i,j} = 1$.
- All requests from $i \in \mathcal{C}$ use the link to its parent: $z_{i,i \to \mathsf{parent}(i)} = 1$

- Let $i \in \mathcal{C}$, and consider any link $l : j \to j' = \mathsf{parent}(j) \in \mathsf{path}[i \to r]$. If $j' = \mathsf{server}(i)$ then link $\mathsf{succ}(l)$ is not used by i (if it exists). Otherwise $z_{i,\mathsf{succ}(l)} = z_{i,l}$. Thus:

$$\forall i \in \mathcal{C}, \forall l : j \to j' = \mathsf{parent}(j) \in \mathsf{path}[i \to r], z_{i,\mathsf{succ}(l)} = z_{i,l} - y_{i,j'}$$

Next there are constraints expressing that server capacities cannot be exceeded:

- The processing capacity of any server cannot be exceeded: $\forall j \in \mathcal{N}, \sum_{i \in \mathcal{C}} r_i y_{i,j} \leq \mathsf{W}_j x_j$. Note that this ensures that if j is the server of i, there is indeed a replica located in node j.

Finally there remains to express the QoS constraints:

$$\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i), \mathsf{dist}(i,j) y_{i,j} \leq \mathsf{qos}_i,$$

where $\mathsf{dist}(i,j) = \mathsf{path}[i \to k]$. As stated previously, we could take the computational time of a request into account by writing $(\mathsf{dist}(i,j) + \mathsf{comp}_j) y_{i,j} \leq \mathsf{qos}_i$, where comp_j would be the time to process a request on server j.

Altogether, we have fully characterized the linear program for the *Upwards* policy. We need additional constraints for the *Closest* policy, which is a particular case of the *Upwards* policy (hence all constraints and equations remain valid).

We need to express that if node j is the server of client i, then no ancestor of j can be the server of a client in the subtree rooted at j. Indeed, a client in this subtree would need to be served by j and not by one of its ancestors, according to the *Closest* policy. A direct way to write this constraint is

$$\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i), \forall i' \in \mathcal{C} \cap \mathsf{subtree}(j), \forall j' \in \mathsf{Ancestors}(j), y_{i,j} \leq 1 - y_{i',j'}.$$

Indeed, if $y_{i,j} = 1$, meaning that $j = \mathsf{server}(i)$, then any client i' in the subtree rooted in j must have its server in that subtree, not closer to the root than j. Hence $y_{i',j'} = 0$ for any ancestor j' of j.

There are $O(s^4)$ such constraints to write, where $s = |\mathcal{C}| + |\mathcal{N}|$ is the problem size. We can reduce this number down to $O(s^3)$ by writing

$$\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i) \setminus \{r\}, \forall i' \in \mathcal{C} \cap \mathsf{subtree}(j), y_{i,j} \leq 1 - z_{i',j \to \mathsf{parent}(j)}.$$

3.2 Multiple servers

We now proceed to the *Multiple* policy. We define the following variables:

Server assignment

- x_j is a boolean variable equal to 1 if j is a server (for one or several clients)
- $y_{i,j}$ is an integer variable equal to the number of requests from client i processed by node j
- If $j \notin \mathsf{Ancestors}(i)$, we directly set $y_{i,j} = 0$.

Link assignment

- $z_{i,l}$ is an integer variable equal to the number of requests flowing through link $l \in \mathsf{path}[i \to r]$ when client i accesses any of its servers in $\mathsf{Servers}(i)$
- If $l \notin \mathsf{path}[i \to r]$ we directly set $z_{i,l} = 0$.

The objective function is unchanged, as the total storage cost still writes $\sum_{j \in \mathcal{N}} \mathsf{sc}_j x_j$. But the constraints must be modified. First those for server and link usage:

- Every request is assigned a server: $\forall i \in \mathcal{C}, \sum_{j \in \mathsf{Ancestors}(i)} y_{i,j} = r_i$.
- All requests from $i \in \mathcal{C}$ use the link to its parent: $z_{i,i \to \mathsf{parent}(i)} = r_i$
- Let $i \in \mathcal{C}$, and consider any link $l : j \to j' = \mathsf{parent}(j) \in \mathsf{path}[i \to r]$. Some of the requests from i which flow through l will be processed by node j', and the remaining ones will flow upwards through link $\mathsf{succ}(l)$:

$$\forall i \in \mathcal{C}, \forall l : j \to j' = \mathsf{parent}(j) \in \mathsf{path}[i \to r], z_{i,\mathsf{succ}(l)} = z_{i,l} - y_{i,j'}$$

The other constraints on server capacities and QoS are slightly modified:

- Servers: $\forall j \in \mathcal{N}, \sum_{i \in \mathcal{C}} y_{i,j} \leq \mathsf{W}_j x_j$. Note that this ensure that if j is the server for one or more requests from i, there is indeed a replica located in node j.
- QoS: $\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i), \mathsf{dist}(i,j) y_{i,j} \leq \mathsf{qos}_i y_{i,j}$

Altogether, we have fully characterized the linear program for the *Multiple* policy.

3.3 An ILP-based lower bound

The previous linear programs contain boolean or integer variables, because it does not make sense to assign half a request or to place one third of a replica on a node. Thus, it must be solved in integer values if we wish to obtain an exact solution to an instance of the problem. This can be done for each access policy, but due to the large number of variables, the problem cannot be solved for platforms of size $s > 50$, where $s = |\mathcal{N}| + |\mathcal{C}|$. Thus we cannot use this approach for large-scale problems.

However, we can relax the constraints and solve the linear program assuming that all variables take rational values. The optimal solution of the relaxed program can be obtained in polynomial time (in theory using the ellipsoid method [9], in practice using standard software packages [2, 6]), and the value of its objective function provides an absolute lower bound on the cost of any valid (integer) solution. For all practical values of the problem size, the rational linear program returns a solution in a few minutes. We tested up to several thousands of nodes and clients, and we always found a solution within ten seconds. Of course the relaxation makes the most sense for the *Multiple* policy, because several fractions of servers are assigned by the rational program.

However, we can obtain a more precise lower bound for trees with up to $s = 400$ nodes and clients by using a rational solution of the *Multiple* instance of the linear program with fewer integer variables. We treat the $y_{i,j}$ and $z_{i,l}$ as rational variables, and only require the x_j to be integer variables. These variables are set to 1 if and only if there is a replica on the corresponding node. Thus, forbidding to set $0 < x_j < 1$ allows us to get a realistic value of the cost of a solution of the problem. For instance, a server might be used only at 50% of its capacity, thus setting $x = 0.5$

would be enough to ensure that all requests are processed; but in this case, the cost of placing the replica at this node is halved, which is incorrect: while we can place a replica or not but it is impossible to place half of a replica.

In practice, this lower bound provides a drastic improvement over the unreachable lower bound provided by the fully rational linear program. The good news is that we can compute the refined lower bound for problem sizes up to $s = 400$, using GLPK [6]. In the next section, we show that this refined bound is an achievable bound, and we provide an exact solution to the *Multiple* instance of the problem, based on the solution of this mixed integer linear program.

3.4 An exact MIP-based solution for *Multiple*

Theorem 1. *The solution of the linear program detailed in 3.2, when solved with all variables being rationals except of the x_i, is an achievable bound for the Multiple problem, and we can build an exact solution in polynomial time, based on the LP solution.*

Proof. Let us consider the solution of the LP program:

- $\forall i \in \mathcal{C},\ x_i \in \{0, 1\}$
- $\forall i \in \mathcal{C}, \forall j \in \mathcal{N},\ y_{i,j} \in \mathcal{R}$
- $\forall i \in \mathcal{C}, \forall l \in \mathcal{L},\ z_{i,l} \in \mathcal{R}$

To prove that the lower bound obtained by this program is achievable, we are building an integer solution where $y'_{i,j}$ and $z'_{i,l}$ are integer numbers, keeping the same x_i and without breaking any constraints.

In the following, for any variable y, $\lfloor y \rfloor$ is the integer part of y, and $\tilde{y}$ the fractional part: $y = \lfloor y \rfloor + \tilde{y}$, and $\tilde{y} < 1$.

Let us consider a client $i \in \mathcal{C}$ such that $\exists j \in \mathcal{N} \mid \tilde{y}_{i,j} > 0$, i.e. $y_{i,j}$ is not an integer. We consider j_1 being the closest server to i not serving an integer number of requests of client i, and more generally $j_k, k = 1..K$ the servers on the path from i to the root, such that $\tilde{y}_{i,j_k} > 0$. We want to move bits of requests in order to obtain an integer value for y_{i,j_1}. This elementary transformation is called $\mathsf{trans}(i, j_1)$. We consider the two following cases.

First case:

$$\sum_{i' \in \mathsf{subtree}(j_1) \cap \mathcal{C}} y_{i',j_1} \leq W_{j_1} - (1 - \tilde{y}_{i,j_1})$$

In this case, there is enough space at server j_1 to fulfill an integer number of requests from client i. Since the total number of requests of client i is an integer, $\sum_{k=1}^{K} \tilde{y}_{i,j_k}$ is a non null integer. Thus, $\sum_{k=2}^{K} \tilde{y}_{i,j_k} \geq 1 - \tilde{y}_{i,j_1}$, and we can move down $1 - \tilde{y}_{i,j_1}$ bits of requests from servers $j_k, k = 2..K$ to j_1. No constraints will be violated since there is enough space on the server, and we lower requests in the tree so the bandwidth is non increasing on all links. The move is done by changing the values of y_{i,j_k} and recalculating the $z_{i,l}$ for $l \in \mathsf{path}[i \to r]$. After such a transformation, y_{i,j_1} is an integer variable.

Second case: If server j_1 is already too full in order to add a fraction of requests from client i, we need to exchange some requests with other clients. First, if there is some free space on the server, we start by filling completely server j_1 with fractions of requests of client i from servers j_k, $k = 2..K$. We know there are such requests, otherwise y_{i,j_1} would be an integer. This transformation is similar as the one done in the first case. We now have

$\sum_{i' \in \mathsf{subtree}(j_1) \cap \mathcal{C}} y_{i',j_1} = W_{j_1}$. Let us denote by $i_t, t = 1..T$ the clients $i_t \in \mathsf{subtree}(j_1) \cap \mathcal{C} \setminus \{i\}$ such that $\tilde{y}_{i_t,j_1} > 0$. Since W_{j_1} is an integer and $\tilde{y}_{i,j_1} > 0$, we have $\sum_{t=1}^{T} \tilde{y}_{i_t,j_1} \geq 1 - \tilde{y}_{i,j_1}$, and also $\sum_{k=2}^{K} \tilde{y}_{i,j_k} \geq 1 - \tilde{y}_{i,j_1}$. We can select in both sets $1 - \tilde{y}_{i,j_1}$ bits of requests which will be exchanged, *i.e.* bits of requests from client i_t initially treated by j_1 will be moved on some servers j_k, which are in $\mathsf{Ancestors}(j_1)$, and the corresponding amount of requests of i will be moved back on server j_1.

In this case, we may break a QoS constraint since it is not sure that clients i_t can be served higher than j_1 in order to respect their QoS. However, we will see that in the general transformation process, we prevent such cases to happen. Note that all other constraints are still fulfilled, in particular the bandwidth one, since we do not change the amount of requests traversing each link, but just change the origin of these requests.

Once $\mathrm{trans}(i, j_1)$ has been done, y_{i,j_1} is an integer, and notice that only non-integer bits of requests have been moved, so we have not affected any integer part of the solution and we have decreased at least by one the number of non-integer variables in the solution.

Let us detail now the complete transformation algorithm, in order to obtain an integer solution. Particular attention must be paid to respect the QoS at all time.

for $j \in \mathcal{N}$ *taken in a bottom-up traversal order* **do**
  finish=1;
  while *(finish==1)* **do**
    $\mathcal{C}' = \{i' \in \mathcal{C} \cap \mathsf{subtree}(j) \mid \tilde{y}_{i',j} > 0\}$;
    if $\mathcal{C}' == \emptyset$ **then** finish=0; **else**
      $i = Min_{i' \in \mathcal{C}'} (\mathsf{qos}_{i'} - \mathsf{dist}(i', j))$;
      trans(i, j);
    end
  end
end

We consider each server in a bottom-up order, so that we are sure that each time we perform an elementary transformation, the server is the first one on the way from the client to the root having a non integer number of requests. In fact, when transforming server j, each server in $\mathsf{subtree}(j)$ has already been transformed, and thus have no fraction numbers of requests.

In order to transform server j, we look at the set $\mathcal{C}'$ of clients having a non-integer number of requests processed at j. If the set is empty, there is nothing to transform at j. Otherwise, we perform the elementary transformation with the client i which minimizes $(\mathsf{qos}_{i'} - \mathsf{dist}(i', j))$, for $i' \in \mathcal{C}'$. This ensures that when we perform an elementary transformation as in the second case above, the QoS constraint will be respected for all clients i_t, since we are moving their requests into servers at distance at most $d = \mathsf{qos}_i - \mathsf{dist}(i, j)$ from j, and their own QoS allows them to be processed at a distance $\mathsf{qos}_{i_t} - \mathsf{dist}(i_t, j) \geq d$. Figure 1 illustrates this phase of the algorithm.

At the end of the while loop, server j is processing only integer numbers of requests, and thus we will not modify its requests affectation any more in the following.

The constraints are all respected at all step of the transformation, and we do not add or remove any replica, so the solution has exactly the same cost than the initial LP-based solution, and the transformed solution is fully integer. Moreover, this transformation algorithm works in polynomial

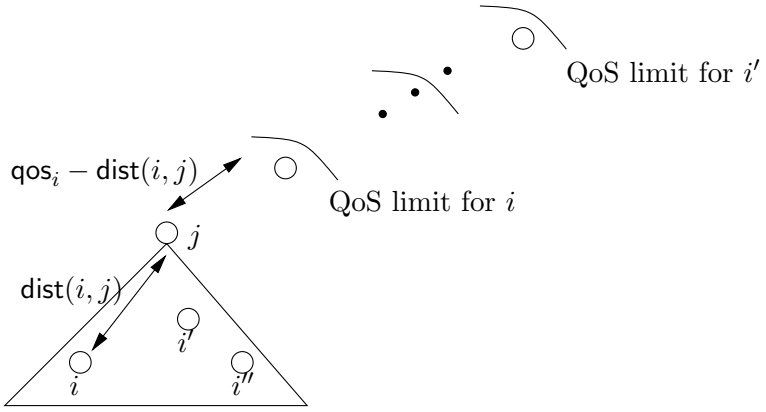

Figure 1: Illustration of the transformation algorithm

	Homogeneous	**Homogeneous/QoS**
Closest	polynomial [4, 8]	polynomial [8]
Upwards	NP-complete [1]	NP-complete [1]
Multiple	polynomial [1]	**NP-complete** (this paper)

Table 1: Complexity results for the different instances of the Replica Counting problem.

time, in the worst case in $|\mathcal{N}|+|\mathcal{C}|^2$ but most of the time it is much faster since the transformations do not concern all clients simultaneously but only a few of them.

□

4 Complexity Results

Table 1 gives an overview of complexity results of the different instances of the Replica Counting problem (homogeneous servers). Liu et al [8] provided a polynomial algorithm for the *Closest* policy with QoS constraints. In [1] we proved the NP-completeness of the *Upwards* policy without QoS. This was a surprising result, to be contrasted with the fact that the *Multiple* policy is polynomial under the same conditions [1].

An important contribution of this paper is the NP-completeness of the *Multiple* policy with QoS constraints. As stated above, the same problem was polynomial without QoS, which gives a clear insight on the additional complexity introduced by QoS constraints. We point out that all three instances of the Replica Placement problem (heterogeneous servers with the *Closest*, *Upwards* and *Multiple* policies) are already NP-complete without QoS constraints [1].

Theorem 2. *The instance of the* Replica Counting *problem with QoS constraints and the Multiple strategy is NP-complete.*

Proof. The problem clearly belongs to the class NP: given a solution, it is easy to verify in polynomial time that all requests are served, that all QoS constraints are satisfied and that no server capacity is exceeded.

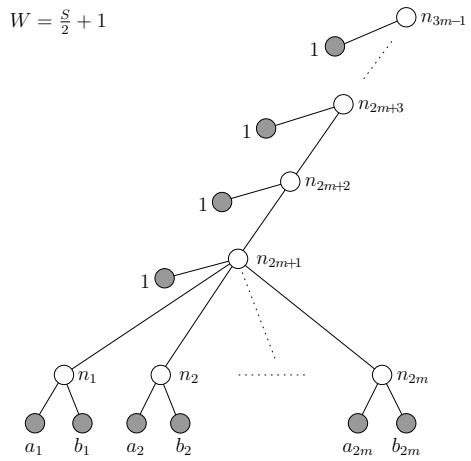

Figure 2: The platform used in the reduction for Theorem 2.

To establish the completeness, we use a reduction from 2-PARTITION-EQUAL [5]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION-EQUAL: given $2m$ positive integers $a_1, a_2, \ldots, a_{2m}$, does there exist a subset $I \subset \{1, \ldots, 2m\}$ of cardinal m such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. Let $S = \sum_{i=1}^{2m} a_i$, $W = \frac{S}{2}$ and $b_i = \frac{S}{2} - 2a_i$ for $1 \leq i \leq 2m$. We build the following instance $\mathcal{I}_2$ of our problem (see Figure 2):

- Problem size: there are $5m - 1$ clients c_i and $3m - 1$ internal nodes n_j:
- Nodes: for $1 \leq j \leq 2m$, node n_j has capacity $\mathsf{W}_j = W$
  - For $1 \leq j \leq 2m$, the parent of node n_j is node n_{2m+1}
  - For $2m + 1 \leq j \leq 3m - 2$, the parent of node n_j is node n_{j+1}
  - Node n_{3m-1} is the root r of the tree.
- Clients:
  - For $1 \leq i \leq 2m$, client c_i has $r_i = a_i$ requests of QoS $\mathsf{qos}_i = 2$, and its parent is node n_i
  - For $2m + 1 \leq i \leq 4m$, client c_i has $r_i = b_{i-2m}$ requests of QoS $\mathsf{qos}_i = m$, and its parent is node n_{i-2m}
  - For $4m + 1 \leq i \leq 5m - 1$, client c_i has $r_i = 1$ request of QoS $\mathsf{qos}_i = 1$ and its parent is node n_{i-2m}.

Finally, we ask whether there exists a solution with total storage cost $(2m - 1)W$, i.e. with $2m - 1$ servers. Clearly, the size of $\mathcal{I}_2$ is polynomial (and even linear) in the size of $\mathcal{I}_1$. We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. We assign a replica to each node n_i, $i \in I$ (by hypothesis there are m of them), and one in each of the $m - 1$ top nodes n_{2m+1} to n_{3m-1}. All $m - 1$ clients with QoS 1 are served by their parent.

For $1 \leq i \leq 2m$ there are tow cases:
- If $i \in \mathcal{I}$, both clients c_i and c_{i+2m} are served by their parent n_i. Node n_i serves a total of $a_i + b_i = \frac{S}{2} - a_i \leq W$ requests.
- If $i \notin \mathcal{I}$, client c_i is served by node n_{2m+1} and client c_{i+2m} is served by one or several ancestors of n_{2m+1}, i.e. nodes n_{2m+2} to n_{3m-1}. Node n_{2m+1}, which also serves the unique request of client

c_{2m+1}, serves a total of $\sum_{i \notin I} a_i + 1 = W$ requests. The $m-2$ ancestors of n_{2m+1} receive the load $\sum_{i \notin I} b_i = mS - 2S$. They also serve $m-2$ clients with a single request, hence a a total load of $(m-2)S + m - 2 = (m-2)W$ requests to distribute among them. This is precisely the sum of their capacities, and any assignment will do the job.
Note that the allocation of requests to servers is compatible with all QoS constraints. All requests with QoS 1 are served by the parent node. All requests with QoS 2, i.e. with value a_i, are served either by the parent node (if $i \in I$) or by the grandparent node (if $i \notin I$). Altogether, we have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution with $2m-1$ servers. Necessarily, there is a replica located in each of the top $m-1$ nodes n_{2m+1} to n_{3m-1}, otherwise some request with QoS 1 would not be served satisfactorily. Each of these nodes serves one of these requests, hence has remaining capacity $W - 1 = \frac{S}{2}$.

There remain m servers which are placed among nodes n_1 to n_{2m}. Let I be the set of indices of those m nodes which have not received a replica. Necessarily, requests a_i, with $i \in I$, are served by node n_{2m+1}, because of their QoS constraint. Hence $\sum_{i \in I} a_i \leq \frac{S}{2}$. Next, all requests a_i and b_i, with $i \in I$, are served by nodes n_{2m+1} to n_{3m-1}, whose total remaining capacity is $(m-1)\frac{S}{2}$. There are $(\sum_{i \in I} a_i) + (m\frac{S}{2} - 2\sum_{i \in I} a_i)$ such requests, hence

$$m\frac{S}{2} - \sum_{i \in I} a_i \leq (m-1)\frac{S}{2}.$$

From this equation we derive that $\sum_{i \in I} a_i \geq \frac{S}{2}$. Finally we have $\sum_{i \in I} a_i = \frac{S}{2}$, with $|I| = m$, hence a solution to $\mathcal{I}_2$.

□

5 Heuristics for the Replica Placement Problem

In this section several heuristics for the *Closest*, *Upwards* and *Multiple* policies are presented. As already pointed out, the quality of service is the number of hops that requests of a client are allowed to traverse until they have to reach their server (QoS=distance). The code of all heuristics can be found on the web [10]. All heuristics described below have polynomial, and even worst-case quadratic, complexity $O(s^2)$, where $s = |\mathcal{C}| + |\mathcal{N}|$ is the problem size.

In the following, we denote by $\mathsf{inreqQoS}_i$ the amount of requests that reach an inner node i within their QoS constraints, and by inreq_i the total amount of requests that reach i (including requests whose QoS constraints are violated).

Closest Big Subtree First - CBS. In this heuristic we traverse the tree in top-down manner. We place a replica on an inner node i if $\mathsf{inreqQoS}_i \leq \mathsf{W}_i$. When the condition holds, we do not process any other subtree of i. If this condition does not hold, we process the subtrees of i in non-increasing order of inreq_i. Once no further replica can be added, we repeat the procedure. We stop when no new replica is added during a pass.

Closest Small QoS First - CSQoS. This heuristic uses a different approach. We do not execute a tree traversal. Instead, we sort all clients by non-decreasing order of qos_i. In case of tie, clients are sorted by non-increasing order of r_i. For each client, we look for the server that can process its subtree ($\mathsf{inreqQoS}_i \leq \mathsf{W}_i$) and which is the nearest to the root. If no server is found for a client, we continue with the next client in the list. Once we reach a client in the list that

is already treated by an earlier chosen server, we delete all treated clients from the to-do list and restart at the beginning of the remaining client list. The procedure stops either when the list is empty or when the end of the list is reached.

Upwards Small QoS Started Servers First - USQoSS. Clients are sorted by non-decreasing order of qos_i (and non-increasing order of r_i in case of tie). For each client i in the list we search for an appropriate server: we take the next server on the way up to the root (i.e., an inner node that is already equipped with a replica) which has enough remaining capacity to treat all the client's requests. Of course the QoS-constraints of the client have to be respected. If there is no server, we take the first inner node j that satisfies $\mathsf{W}_j \geq r_i$ within the QoS-range and we place a replica in j. If we still find no appropriate node, this heuristic has no feasible solution.

Upwards Small QoS Minimal Requests - USQoSM. This heuristic processes the clients in the same order as the previous one, but the choice of the appropriate server differs. Among the nodes in the QoS-range of client i, the node j with minimal $(\mathsf{W}_j - \mathsf{inreqQoS}_j)$-value is chosen as a server if it can satisfy r_i requests. Again it may happen that the heuristic cannot find a feasible solution, whenever no inner node can be found for a client.

Upwards Minimal Distance - UMD. This heuristic requires two steps. In the first step, so-called indispensable servers are chosen, i.e. inner nodes which have a client that must be treated by this very node. At the beginning, all servers that have a child client with $\mathsf{qos} = 1$ will be chosen. This step guarantees that in each loop of the algorithm, we do not forget any client. The criterion for indispensable servers is the following: for each client check the number of nodes eligible as servers; if there is only one, this node is indispensable and chosen. The second step of UMD chooses the inner node with minimal $(\mathsf{W}_j - \mathsf{inreqQoS}_j)$-value as server (if $\mathsf{inreqQoS}_j > 0$). Note that this value can be negative. Then clients are associated to this server in order of distance, i.e. clients that are next to the server are chosen first, until the server capacity W_j is reached or no further client can be found.

Multiple Small QoS Close Servers First - MSQoSC. The main idea of this heuristic is the same as for USQoSS, but with two differences. Searching for an appropriate server, we take the next inner node on the way up to the root which has some remaining capacity. Note that this makes the difference between *close* and *started* servers. If this capacity W_i is not sufficient (client c has more requests, $\mathsf{W}_i < r_c$), we choose other inner nodes going upwards to the root until all requests of the client can be processed (this is possible owing to the multiple-server relaxation). If we cannot find enough inner nodes for a client, this heuristic will not return a feasible solution.

Multiple Small QoS Minimal Requests - MSQoSM. This heuristic is a mix of USQoSM and MSQoSC. Clients are treated in non-decreasing order of qos_i, and the appropriate servers i are chosen by minimal $(\mathsf{W}_i - \mathsf{inreqQoS}_i)$-value until all requests of clients can be processed.

Multiple Minimal Requests - MMR. This heuristic is the counterpart of UMD for the *Multiple* policy and requires two steps. Step one is the same as in UMD, with extension to the multiple-server policy: servers are added in the "indispensable" step, either when they are the only possible server for a client, or when the total capacity of all possible inner nodes for a client i is exactly r_i. The server chosen in the second step is also the inner node with minimal $(\mathsf{W}_j - \mathsf{inreqQoS}_j)$-value, but this time clients are associated in non-decreasing order of $\min(\mathsf{qos}_i, d(i,r))$, where $d(i,r)$ is the number of hops between i and the root of the tree. Note that the last client that is associated to a server, might not be processed entirely by this server.

Mixed Best - MB. This heuristic unifies all previous ones. For each tree, we select the best cost returned by the previous eight heuristics. Since each solution of *Closest* is also a solution for

Upwards, which in turn is a valid solution for *Multiple*, this heuristic provides a solution for the *Multiple* policy.

6 Experimental Plan

In this section we evaluate the performance of our heuristics on tree platforms with varying parameters. Through these experiments we want to assess the different access policies, and the impact of QoS constraints on the performance of the heuristics. We obtain an optimal solution for each tree platform with the help of a mixed integer linear program, see [1] for further details. We can compute the latter optimal solution for problem sizes up to 400 nodes and clients, using GLPK [6]. This optimal solution gives us a feasible lower bound. We used this bound for all our experiments.

An important parameter in our tree networks is the load, i.e. the total number of requests compared to the total processing power: $\lambda = \frac{\sum_{i \in \mathcal{C}} r_i}{\sum_{j \in \mathcal{N}} \mathsf{W}_j}$, where $\mathcal{C}$ is the set of clients in the tree and $\mathcal{N}$ the set of inner nodes. We tested our heuristics for $\lambda = 0.1, 0.2, ..., 0.9$, each on 30 randomly generated trees of two heights: we made a first series of experiments where trees have a height between 4 and 7 (in the following we call them small trees). In the second series, tree heights vary between 16 and 21 (big trees). All trees have s nodes, where $15 \leq s \leq 400$. To assess the impact of QoS on the performance, we study the behavior (i) when QoS constraints are very tight ($\mathsf{qos} \in \{1, 2\}$); (ii) when QoS constraints are more relaxed (the average value is set to half of the tree height height); and (iii) without any QoS constraint at all ($\mathsf{qos} = \mathsf{height} + 1$).

We have computed the number of solutions for each lambda and each heuristic. The number of solutions obtained by the linear program indicates which problems are solvable. Of course we cannot expect a result with our heuristics for intractable problems. To assess the performance of our heuristics, we have studied the relative performance of each heuristic compared to the optimal solution. This allows to compare the cost of the different heuristics, and thus to compare the different access policies. For each λ, the cost is computed on the trees for which the linear program has a solution. Let T_λ be the subset of trees with a LP solution. Then, the relative performance for the heuristic h is obtained by $\frac{1}{|T_\lambda|} \sum_{t \in T_\lambda} \frac{cost_{LP(t)}}{cost_h(t)}$, where $cost_{LP(t)}$ is the optimal solution cost returned by the linear program on tree t, and $cost_h(t)$ is the cost involved by the solution proposed by heuristic h. In order to be fair versus heuristics that have a higher success rate, we set $cost_h(t) = +\infty$, if the heuristic did not find any solution.

6.1 Success

Figures 3, 5 and 7 show the percentage of success of each heuristic for small trees, while the percentage of success for big trees is shown in Figures 4, 6 and 8. A general overview of all figures shows that, as expected, the *Closest* policy has the poorest success rate for all its heuristics, whereas the *Multiple* heuristics almost always find a solution when the LP finds one. In fact, MB and MSQoSC always find a solution when the LP does with the exception of the configuration (small trees, $\lambda \geq 0.5$, $\mathsf{qos} \in \{1, 2\}$). In this case the success rate is slightly inferior. Examining the *Closest* heuristics, CBSF finds in almost all configurations more solutions than CSQoS (exception: configuration (small trees, $\lambda = 0.4$, $\mathsf{average_{qos}} = \mathsf{height}/2$)). The *Upwards* heuristic that finds the most solutions is UDS, followed by USQoSS. In the case of no QoS constraints (see Figures 7 and 8), the *Closest* heuristics outperform USQoSM and MSQoSM for small values of λ. In general MSQoSM finds fewer solutions than other *Multiple* heuristics.

6.2 Relative performance

Figures 9 to 14 represent the relative performance of the heuristics, compared to the LP-based optimal solution, where Figures 9, 11 and 13 deal with small trees, and Figures 10, 12 and 14 consider big ones. As expected, the hierarchy between the policies is respected, i.e. *Multiple* is better than *Upwards* which in turn is better than *Closest*. There is an exception: on small trees with no QoS and $\lambda \in \{1, 2\}$ the best results are achieved with the *Closest* heuristics. This is due to the fact that these heuristics assign servers to process their whole subtrees, whereas in the other policies there is the risk of choosing servers "too early". Altogether, the use of the MixedBest heuristic MB allows to always pick up the best result, thereby providing a very satisfying cost for the *Multiple* instances of the problem. The comparison of the results on small trees and those on big trees shows that QoS constraints are better supported by big trees: on big trees MB always achieves a relative performance of at least 85% (even 95% when $\mathsf{qos} \in \{1, 2\}$) while its relative performance on small trees has a strong dependence on QoS constraints: the tighter the QoS constraints, the better the results (70% without QoS up to 90% with $\mathsf{qos} \in \{1, 2\}$). The influence of the QoS constraints is also perceivable on some particular heuristics: MSQoSM performs poorly when QoS constraints are not tight, but achieves the best relative cost when $\mathsf{qos} \in \{1, 2\}$. This is also true for USQoSM in comparison with USQoSS and UMD.

6.3 Hierarchy of the policies

An another point of interest of this paper is the following question: Have QoS constraints an influence on the hierarchical behaviour of the three policies that we stated in our precedent studies [1]? For this purpose we color the different heuristics of a policy x whith one single color (see Figures 15 to 20). This makes it possible to visualize the general behaviour of the different access policies.

We state that the QoS constraints do not have an impact on the hierarchical behaviour of the policies: we still have *Closest* $\leq$ *Upwards* $\leq$ *Multiple*. When QoS constraints are restricting ($\mathsf{qos} \in \{1, 2\}$), there is a big gap between the best *Closest* heuristic CBS and the others, particularly when λ is small. For small λ *Upwards* and *Multiple* policies perform nearly the same, but when $\lambda > 0.4$, *Multiple* outperforms *Upwards*. "Outperform" in this case means, that there exists a *Multiple* heuristic that has a better relative performance than the best *Upwards* heuristic. When QoS is less restricting, i.e. $\mathsf{average}_{\mathsf{qos}} = \mathsf{height}/2$, we can observe similar behaviours. *Upwards* has still the poorest relative performance, whereas the gap to the other policies is less. The difference of *Upwards* and *Multiple* once again grows with increasing λ. In the configuration (small trees, $\lambda \in \{0.1, 0.2\}$, no QoS) *Closest* misbehaves in the hierarchical predictions: it shows the best performance. This effect can be reasonned with the following: In this work we tried to propose heuristics that are well adapted to QoS constraints. As in the actual configuration there are not any QoS restrictions, the "advantages" of *Upwards* or *Multiple* heuristics can not be exploited in the same efficience. For $\lambda \geq 0.3$ we once again state the hierarchical behaviour.

6.4 Impact of QoS constraints on the relative performance

As you may have already remarked, one of the *Multiple* heuristics, MSQoSM, sometimes has a poor relative performance in comparison to the other heuristics. So we study in Figures 21 to 26 the impact of QoS constraints on the relative performance. MSQoSM has indeed a poor relative performance, when $\mathsf{average}_{\mathsf{qos}} = \mathsf{height}/2$ and also for $\mathsf{qos} = \mathsf{height} + 1$. But by restricting QoS constraints it achieves the best results. We can observe the same dependance on QoS for USQoSS.

6.5 Summary

Globally, all the results show that QoS constraints do not modify the relative performance of the three policies: with or without QoS, *Multiple* is better than *Upwards*, which in turn is better than *Closest*, and their difference in performance is not sensitive to QoS tightness. This is an enjoyable result, that could not be predicted a priori. Altogether we conclude, when QoS is very restricting and λ small, that MSQoSM is the best choice. For big λ, MSQoSC is to prefer. In the case of less restricting QoS values, we choose MMR for λ up to 0.4 and then MSQoSC. Generally, when λ is high, MSQoSC never performs poorly. Concerning the *Upwards* policy, USQoSS behaves the best for tight QoS, in the other cases UMD achieves better results. Finally, CBS always outperforms CSQoS.

7 Conclusion

In this paper we dealt with the Replica Placement optimization problem with QoS constraints. We have proved NP-completeness for *Multiple*/Homogeneous/QoS instances, and we have proposed a set of efficient heuristics for the *Closest*, *Upwards* and *Multiple* access policies. To evaluate the absolute performance of our algorithms, we have compared the experimental results to the optimal solution of an integer linear program, and these results turned out quite satisfactory. In our experiments we have assessed the impact of QoS constraints on the different policies, and we have discussed which heuristic performed best depending upon problem instances, platform parameters and QoS tightness. We have also showed the impact of platform size on the performances.

There remains much work to extend the results of this paper. Bandwidth and communication costs could be included in the experimental plan. Also the structure of the tree networks has to be studied more precisely. In this paper we have restricted ourselves to different tree heights, but it would be interesting to study the impact of the average degree of the nodes onto the performance. In a longer term, the extension of the Replica Placement optimization problem to various object types should be considered, which would call for the design and evaluation of new efficient heuristics.

References

[1] A. Benoit, V. Rehn, and Y. Robert. Strategies for Replica Placement in Tree Networks. Research Report 2006-30, LIP, ENS Lyon, France, Oct. 2006. Available at `graal.ens-lyon.fr/~yrobert/`.

[2] B. W. Char, K. O. Geddes, G. H. Gonnet, M. B. Monagan, and S. M. Watt. *Maple Reference Manual*, 1988.

[3] Y. Chen, R. H. Katz, and J. D. Kubiatowicz. Dynamic Replica Placement for Scalable Content Delivery. In *Peer-to-Peer Systems: First International Workshop, IPTPS 2002*, pages 306–318, Cambridge, MA, USA, Mar. 2002.

[4] I. Cidon, S. Kutten, and R. Soffer. Optimal allocation of electronic content. *Computer Networks*, 40:205–218, 2002.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[6] GLPK: GNU Linear Programming Kit. `http://www.gnu.org/software/glpk/`.

[7] M. Karlsson, C. Karamanolis, and M. Mahalingam. A framework for evaluating replica placement algorithms. Research Report HPL-2002-219, HP Laboratories, Palo Alto, CA, 2002.

[8] P. Liu, Y.-F. Lin, and J.-J. Wu. Optimal placement of replicas in data grid environments with locality assurance. In *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society Press, 2006.

[9] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1986.

[10] Source Code for the Heuristics. `http://graal.ens-lyon.fr/~vrehn/code/replicaQoS/`.

[11] X. Tang and J. Xu. QoS-Aware Replica Placement for Content Distribution. *IEEE Trans. Parallel Distributed Systems*, 16(10):921–932, 2005.

[12] H. Wang, P. Liu, , and J.-J. Wu. A QoS-aware Heuristic Algorithm for Replica Placement. In *Proceedings of the 7th International Conference on Grid Computing, GRID2006*, pages 96–103. IEEE Computer Society, 2006.

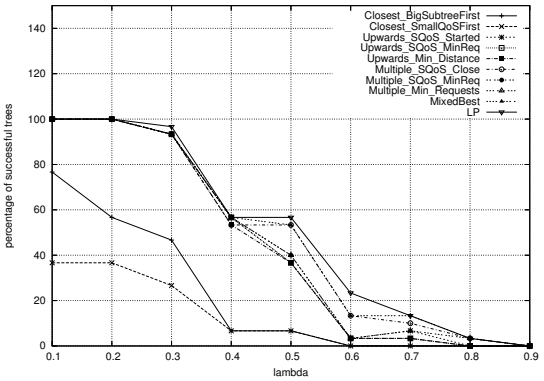

Figure 3: Success for small trees, $\mathsf{qos} \in \{1, 2\}$.

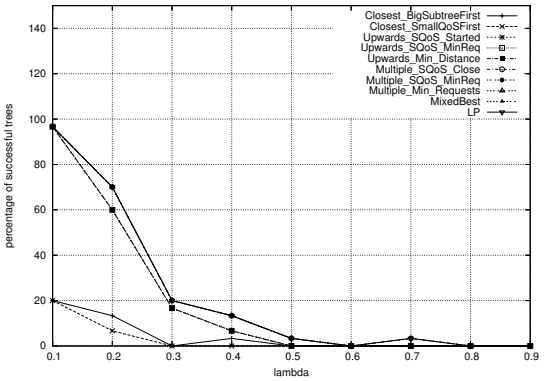

Figure 4: Success for big trees, $\mathsf{qos} \in \{1, 2\}$.

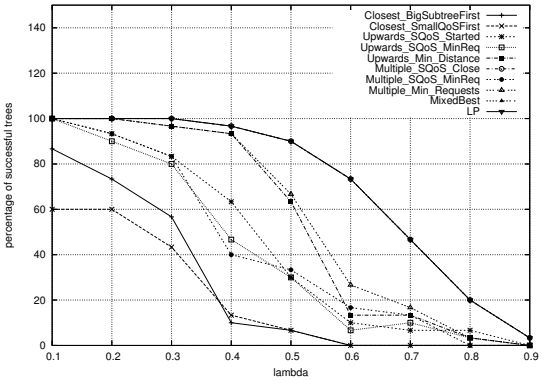

Figure 5: Success for small trees, $\mathsf{average_{qos}} = \mathsf{height}/2$.

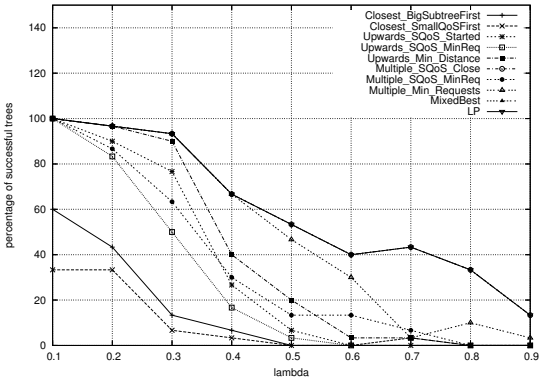

Figure 6: Success for big trees, $\mathsf{average_{qos}} = \mathsf{height}/2$.

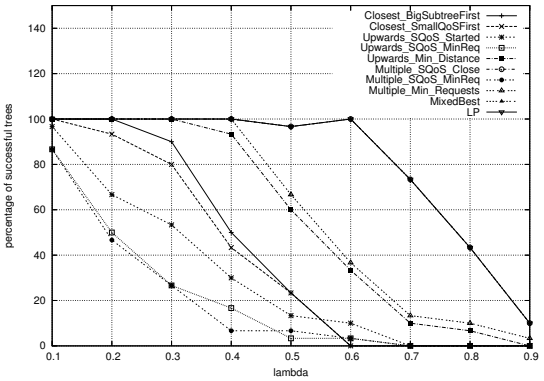

Figure 7: Success for small trees, $\mathsf{qos} = \mathsf{height} + 1 \longrightarrow$ no QoS.

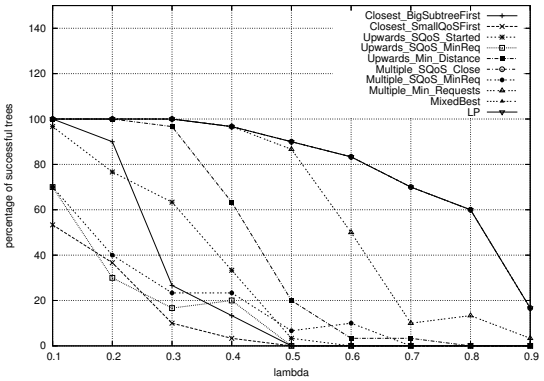

Figure 8: Success for big trees, $\mathsf{qos} = \mathsf{height} + 1 \longrightarrow$ no QoS.

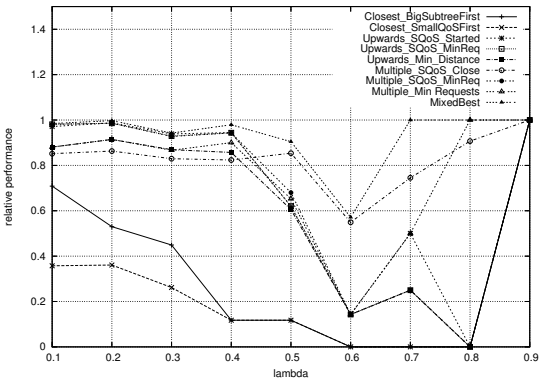

Figure 9: Relative performance for small trees, $\mathsf{qos} \in \{1, 2\}$.

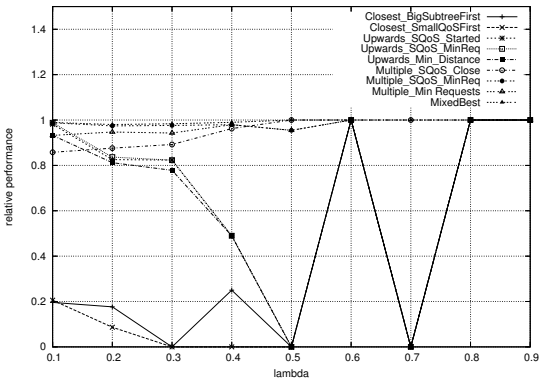

Figure 10: Relative performance for big trees, $\mathsf{qos} \in \{1, 2\}$.

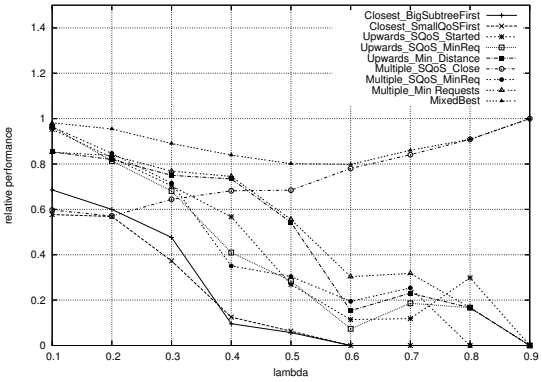

Figure 11: Relative performance for small trees, $\mathsf{average}_{\mathsf{qos}} = \mathsf{height}/2$.

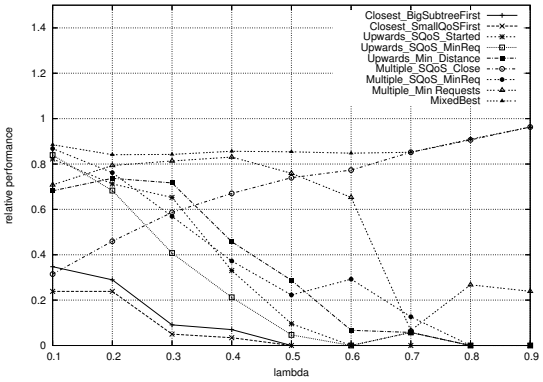

Figure 12: Relative performance for big trees, $\mathsf{average}_{\mathsf{qos}} = \mathsf{height}/2$.

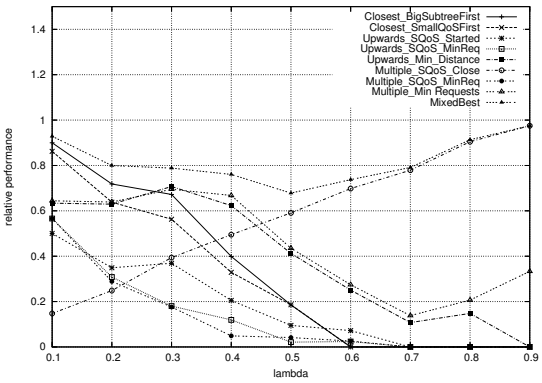

Figure 13: Relative performance for small trees, $\mathsf{qos} = \mathsf{height} + 1 \longrightarrow$ no QoS.

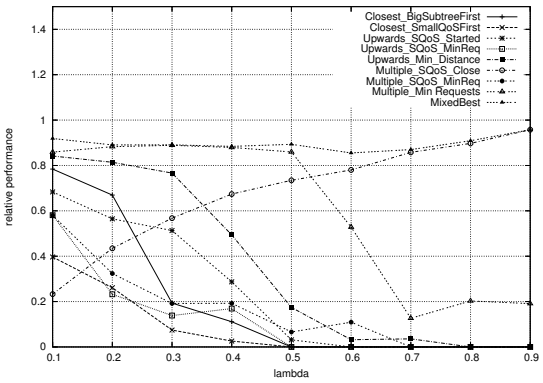

Figure 14: Relative performance for big trees, $\mathsf{qos} = \mathsf{height} + 1 \longrightarrow$ no QoS.

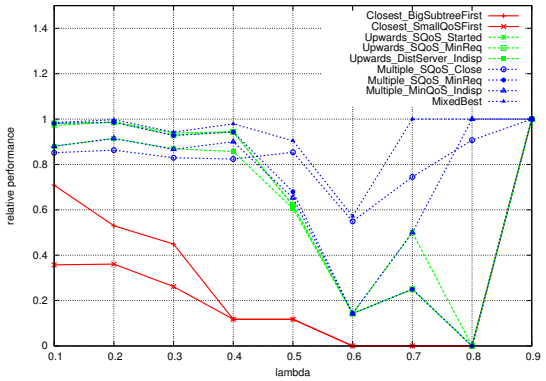

Figure 15: Hierarchy in small trees, $\mathsf{qos} \in \{1, 2\}$.

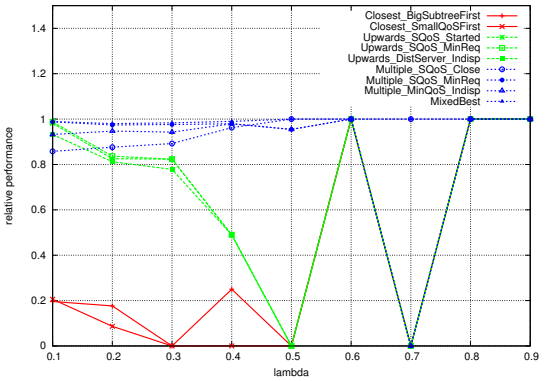

Figure 16: Hierarchy in big trees, $\mathsf{qos} \in \{1, 2\}$.

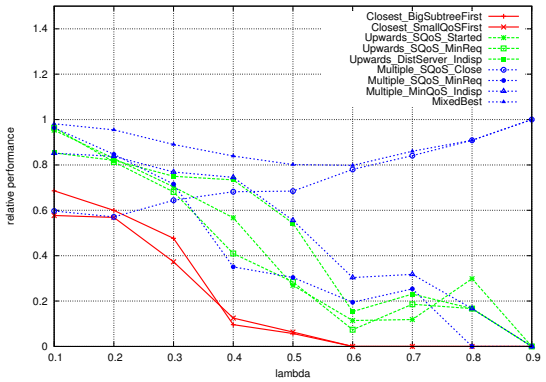

Figure 17: Hierarchy in small trees, $\mathsf{average}_{\mathsf{qos}} = \mathsf{height}/2$.

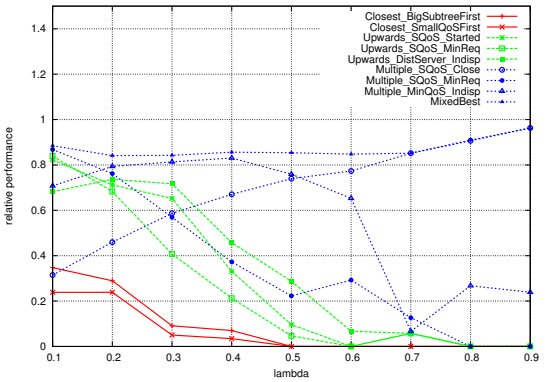

Figure 18: Hierarchy in big trees, $\mathsf{average}_{\mathsf{qos}} = \mathsf{height}/2$.

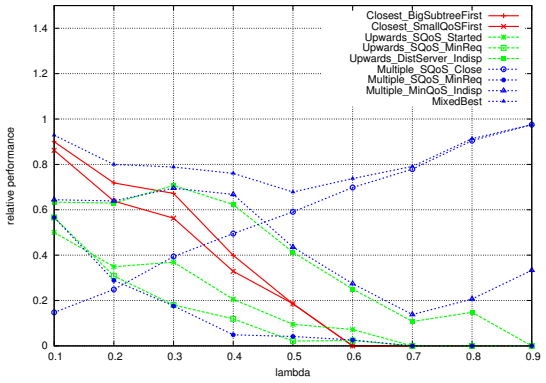

Figure 19: Hierarchy in small trees, $\mathsf{qos} = \mathsf{height} + 1 \longrightarrow$ no QoS.

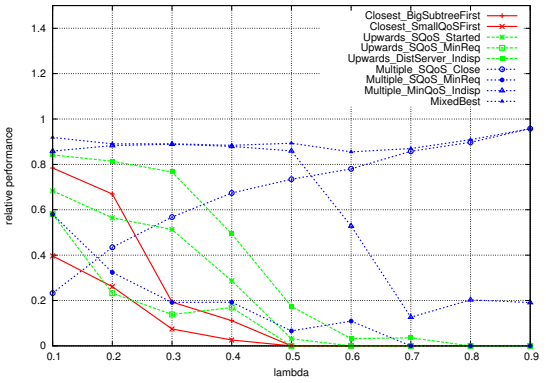

Figure 20: Hierarchy in big trees, $\mathsf{qos} = \mathsf{height} + 1 \longrightarrow$ no QoS.

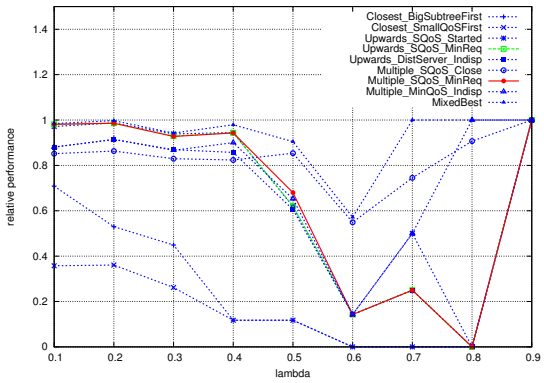

Figure 21: Impact of QoS constraints in small trees, $\mathsf{qos} \in \{1, 2\}$.

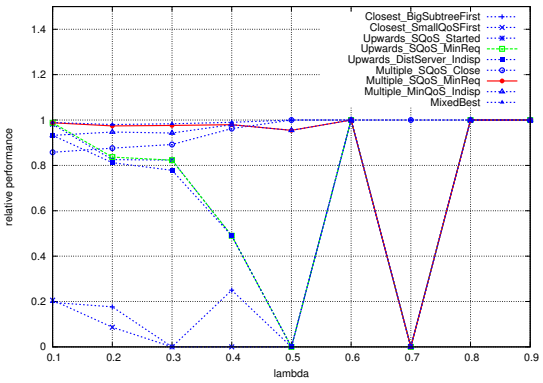

Figure 22: Impact of QoS constraints in big trees, $\mathsf{qos} \in \{1, 2\}$.

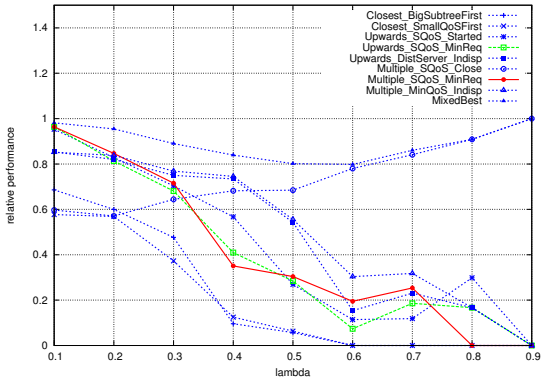

Figure 23: Impact of QoS constraints in small trees, $\mathsf{qos} = \mathsf{average}_{\mathsf{qos}} = \mathsf{height}/2$.

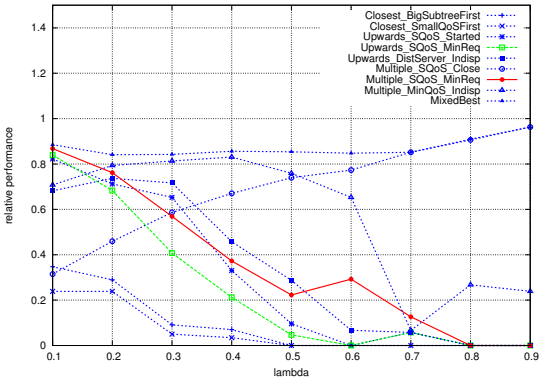

Figure 24: Impact of QoS constraints in big trees, $\mathsf{qos} = \mathsf{average}_{\mathsf{qos}} = \mathsf{height}/2$.

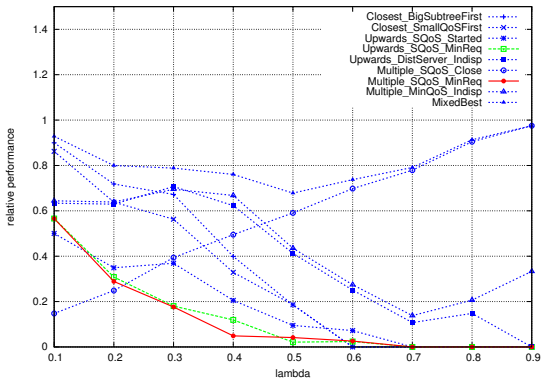

Figure 25: Impact of QoS constraints in small trees, $\mathsf{qos} = \mathsf{height} + 1 \longrightarrow$ no QoS.

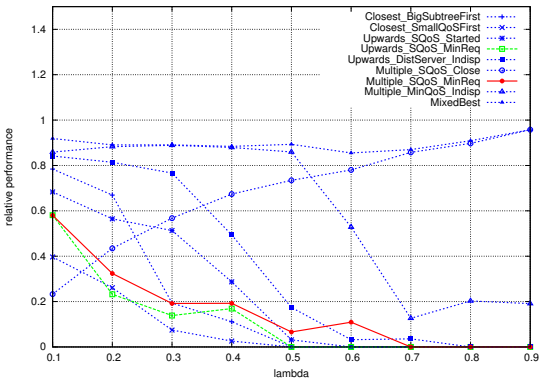

Figure 26: Impact of QoS constraints in big trees, $\mathsf{qos} = \mathsf{height} + 1 \longrightarrow$ no QoS.