Strategies for Replica Placement in Tree Networks

Anne Benoit, Veronika Rehn, and Yves Robert

Laboratoire LIP, UMR CNRS - ENS Lyon - INRIA - UCB Lyon 5668
École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France

E-mail: {Anne.Benoit|Veronika.Rehn|Yves.Robert}@ens-lyon.fr

Abstract

In this paper, we discuss and compare several policies to place replicas in tree networks, subject to server capacity constraints. The client requests are known beforehand, while the number and location of the servers are to be determined. The standard approach in the literature is to enforce that all requests of a client be served by the closest server in the tree. We introduce and study two new policies. In the first policy, all requests from a given client are still processed by the same server, but this server can be located anywhere in the path from the client to the root. In the second policy, the requests of a given client can be processed by multiple servers.

One major contribution of this paper is to assess the impact of these new policies on the total replication cost. Another important goal is to assess the impact of server heterogeneity, both from a theoretical and a practical perspective. In this paper, we establish several new complexity results, and provide several efficient polynomial heuristics for NP-complete instances of the problem. These heuristics are compared to an absolute lower bound provided by the formulation of the problem in terms of the solution of an integer linear program.

1. Introduction

In this paper, we consider the general problem of replica placement in tree networks. Informally, there are clients issuing requests to be satisfied by servers. 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 replica; when a node is equipped with a replica, it can process a number of requests, up to its capacity limit. 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. 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 video on demand (VOD) service delivery.

The rule of the game is to assign replicas to nodes so that some optimization function is minimized. Typically, this optimization function is the total utilization cost of the servers. If all the nodes are identical, this reduces to minimizing the number of replicas. If the nodes are heterogeneous, it is natural to assign a cost proportional to their capacity (so that one replica on a node capable of handling 200 requests is equivalent to two replicas on nodes of capacity 100 each). The core of the paper is devoted to the study of the previous optimization problem, called Replica Placement in the following. The objective of this paper is twofold: (i) introducing two new access policies and comparing them with the standard approach; (ii) assessing the impact of server heterogeneity on the problem.

In most papers from the literature (see Section 8 for a survey of related work), all requests of a client are served by the closest replica, i.e the first replica found in the unique path from the client to the root in the distribution tree. This *Closest* policy is simple and natural, but may be unduly restrictive, leading to a waste of resources. We introduce and study two different ap-

1-4244-0910-1/07/$20.00 ©2007 IEEE.

proaches: in the first one, we keep the restriction that all requests from a given client are processed by the same replica, but we allow client requests to "traverse" servers so as to be processed by other replicas located higher in the path (closer to the root). We call this approach the *Upwards* policy. In the second approach, we further relax access constraints and grant the possibility for a client to be assigned several replicas. With this *Multiple* policy, the processing of a given client's requests will be split among several servers located in the tree path from the client to the root. Obviously, this policy is the most flexible, and likely to achieve the best resource usage. As already stated, one major objective of this paper is to compare these three access policies, *Closest*, *Upwards* and *Multiple*.

The second major contribution of the paper is to assess the impact of server heterogeneity, both from a theoretical and a practical perspective. Recently, several variants of the REPLICA PLACEMENT optimization problem with the *Closest* policy have been shown to have polynomial complexity. In this paper, we establish several new complexity results. Those for the homogeneous case are surprising: the *Multiple* policy is polynomial (as *Closest*) while *Upwards* is NP-hard. The three policies turn out to be NP-complete for heterogeneous nodes, which provides yet another example of the additional difficulties induced by resource heterogeneity. On the more practical side, we provide several heuristics for all three policies. We are able to assess the absolute performance of the heuristics, owing to a lower bound provided by a new formulation of the REPLICA PLACEMENT problem in terms of an mixed integer linear program.

2. Framework

This section is devoted to a precise statement of the REPLICA PLACEMENT optimization problem. 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. It would be easy to allow *client-server* nodes which play both the rule of a client and of an internal node (possibly a server), by dividing such a node into two distinct nodes in the tree, connected by an edge with zero communication cost.

A *client* $i \in \mathcal{C}$ is making r_i requests per time unit to a database. A *node* $j \in \mathcal{N}$ may or may not have been provided with a replica of the database. Nodes equipped with a replica (*i.e.* servers) can process up to W_j requests per time unit from clients in their 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 $\mathsf{Servers}(i) \subseteq \mathcal{N}$ this set of nodes. The price to pay to place a replica at node j is sc_j.

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 $l : k \to k' = \mathsf{parent}(k)$ is any link in the tree, then $\mathsf{succ}(l)$ is the link $k' \to \mathsf{parent}(k')$ (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 $\mathsf{path}[k \to k']$ denotes the set of links in the path from k to k'; also, $\mathsf{subtree}(k)$ is the subtree rooted in k, including k.

Let $r_{i,j}$ be the number of requests from client i processed by server j (of course, $\sum_{j \in \mathsf{Servers}(i)} r_{i,j} = r_i$). In the following, R is the set of replicas: $R = \{j \in \mathcal{N} | \ \exists i \in C \ , \ j \in \mathsf{Servers}(i)\}$. The problem is constrained by the fact that no server capacity can be exceeded: $\forall j \in R, \sum_{i \in \mathcal{C} | j \in \mathsf{Servers}(i)} r_{i,j} \leq \mathsf{W}_j$.

The objective function for the REPLICA PLACEMENT problem is defined as: Min $\sum_{s \in R} \mathsf{sc}_s$.

As already pointed out, it is frequently assumed that the cost of a server is proportional to its capacity, so in some problem instances we let $\mathsf{sc}_s = \mathsf{W}_s$. We name REPLICA COST this fundamental problem. We can further simplify this problem in the homogeneous case: with identical servers, the REPLICA COST problem amounts to minimize the number of replicas needed to solve the problem. In this case, the storage cost sc_j is set to 1 for each node. We call this problem REPLICA COUNTING.

3 Access policies

In this section we review the usual policies enforcing which replica is accessed by a given client. Consider that each client i is making r_i requests per time-unit. There are two scenarios for the number of servers assigned to each client:

Single server – Each client i is assigned a single server $\mathsf{server}(i)$, that is responsible for processing all its requests.

Multiple servers – A client i may be assigned several servers in a set $\mathsf{Servers}(i)$. Each server $s \in \mathsf{Servers}(i)$ will handle a fraction $r_{i,s}$ of the requests. Of course $\sum_{s \in \mathsf{Servers}(i)} r_{i,s} = r_i$.

To the best of our knowledge, the single server policy has been enforced in all previous approaches. One

objective of this paper is to assess the impact of this restriction on the performance of data replication algorithms. The single server policy may prove a useful simplification, but may come at the price of a non-optimal resource usage.

In the literature, the single server strategy is further constrained to the *Closest* policy. Here, the server of client i is constrained to be the first server found on the path that goes from i upwards to the root of the tree. In particular, consider a client i and its server $\mathsf{server}(i)$. Then any other client node i' residing in the subtree rooted in $\mathsf{server}(i)$ will be assigned a server in that subtree. This forbids requests from i' to "traverse" $\mathsf{server}(i)$ and be served higher (closer to the root in the tree).

We relax this constraint in the *Upwards* policy which is the general single server policy. Notice that a solution to *Closest* always is a solution to *Upwards*, thus *Upwards* is always better than *Closest* in terms of the objective function. Similarly, the *Multiple* policy is always better than *Upwards*, because it is not constrained by the single server restriction.

The following sections illustrate the three policies. Section 3.1 provides simple examples where there is a valid solution for a given policy, but none for a more constrained one. Section 3.2 shows that *Upwards* can be arbitrarily better than *Closest*, while Section 3.3 shows that *Multiple* can be arbitrarily better than *Upwards*. We conclude with an example showing that the cost of an optimal solution of the REPLICA COUNTING problem (for any policy) can be arbitrarily higher than the obvious lower bound

$$\left\lceil \frac{\sum_{i \in \mathcal{C}} r_i}{W} \right\rceil,$$

where W is the server capacity.

3.1 Impact of the access policy on the existence of a solution

We consider here a very simple instance of the REPLICA COUNTING problem. In this example there are two nodes, s_1 being the unique child of s_2, the tree root (see Figure 1). Each node can process $\mathsf{W} = 1$ request.

- If s_1 has one client child making 1 request, the problem has a solution with all three policies, placing a replica on s_1 or on s_2 indifferently (Figure 1(a)).
- If s_1 has two client children, each making 1 request, the problem has no more solution with *Closest*. However, we have a solution with both *Upwards* and *Multiple* if we place replicas on both nodes. Each server will process the request of one of the clients (Figure 1(b)).
- Finally, if s_1 has only one client child making 2 requests, only *Multiple* has a solution since we need to process one request on s_1 and the other on s_2, thus requesting multiple servers (Figure 1(c)).

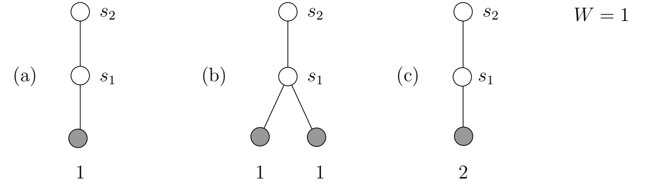

Figure 1. Access policies.

This example demonstrates the usefulness of the new policies. The *Upwards* policy allows to find solutions when the classical *Closest* policy does not. The same holds true for *Multiple* versus *Upwards*. In the following, we compare the cost of solutions obtained with different strategies.

3.2 *Upwards* versus *Closest*

In the following example, we construct an instance of REPLICA COUNTING where the cost of the *Upwards* policy is arbitrarily lower than the cost of the *Closest* policy. We consider the tree network of Figure 2, where there are $2n+2$ internal nodes, each with $\mathsf{W}_j = W = n$, and $2n + 1$ clients, each with $r_i = r = 1$.

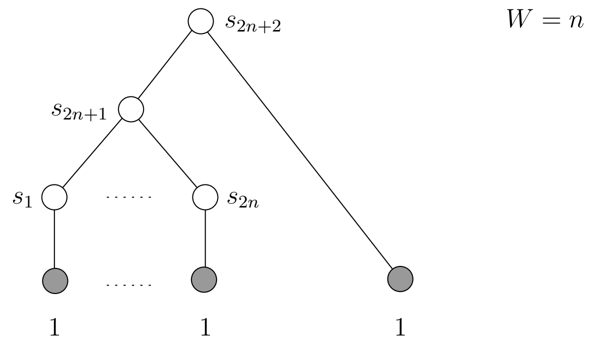

Figure 2. *Upwards* **versus** *Closest*

With the *Upwards* policy, we place three replicas in s_{2n}, s_{2n+1} and s_{2n+2}. All requests can be satisfied with these three replicas.

When considering the *Closest* policy, first we need to place a replica in s_{2n+2} to cover its client. Then,

- Either we place a replica on s_{2n+1}. In this case, this replica is handling n requests, but there re-

main n other requests from the $2n$ clients in its subtree that cannot be processed by s_{2n+2}. Thus, we need to add n replicas between $s_1..s_{2n}$.
- Otherwise, $n-1$ requests of the $2n$ clients in the subtree of s_{2n+1} can be processed by s_{2n+2} in addition to its own client. We need to add $n+1$ extra replicas among $s_1, s_2, \ldots, s_{2n}$.

In both cases, we are placing $n+2$ replicas, instead of the 3 replicas needed with the *Upwards* policy. This proves that *Upwards* can be arbitrary better than *Closest* on some Replica Counting instances.

3.3 *Multiple* versus *Upwards*

In this section we build an instance of the Replica Counting problem where *Multiple* is twice better than *Upwards*. We do not know whether there exist instances of Replica Counting where the performance ratio of *Multiple* versus *Upwards* is higher than 2 (and we conjecture that this is not the case). However, we also build an instance of the Replica Cost problem (with heterogeneous nodes) where *Multiple* is arbitrarily better than *Upwards*.

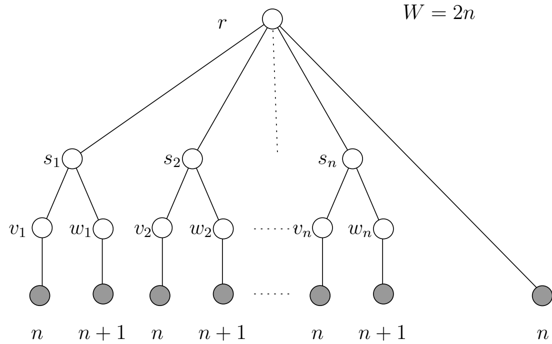

Figure 3. *Multiple* **versus** *Upwards***, homogeneous platforms.**

We start with the homogeneous case. Consider the instance of Replica Counting represented in Figure 3, with $3n+1$ nodes of capacity $\mathsf{W}_j = W = 2n$. The root r has $n+1$ children, n nodes labeled s_1 to s_n and a client with $r_i = n$. Each node s_j has two children nodes, labeled v_j and w_j for $1 \leq j \leq n$. Each node v_j has a unique child, a client with $r_i = n$ requests; each node w_j has a unique child, a client with $r_i = n+1$ requests.

The *Multiple* policy assigns $n+1$ replicas, one to the root r and one to each node s_j. The replica in s_j can process all the $2n+1$ requests in its subtree except one, which is processed by the root.

For the *Upwards* policy, we need to assign one replica to r, to cover its client. This replica can process n other requests, for instance those from the client child of v_1. We need to place at least a replica in s_1 or in w_1, and $2(n-1)$ replicas in v_j and w_j for $2 \leq j \leq n$. This leads to a total of $2n$ replicas, hence a performance factor $\frac{2n}{n+1}$ whose limit is to 2 when n tends to infinity.

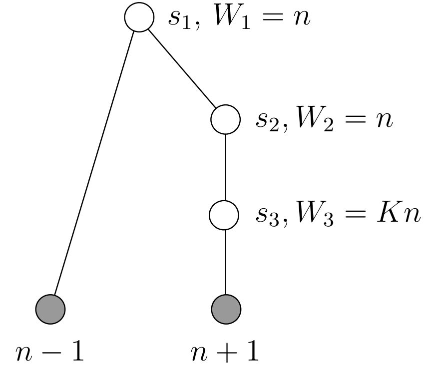

Figure 4. *Multiple* **versus** *Upwards***, heterogeneous platforms.**

We now proceed to the heterogeneous case. Consider the instance of Replica Cost represented in Figure 4, with 3 nodes s_1, s_2 and s_3, and 2 clients. The capacity of s_1 and s_2 is $\mathsf{W}_1 = \mathsf{W}_2 = n$ while that of s_3 is $\mathsf{W}_3 = Kn$, where K is arbitrarily large. Recall that in the Replica Cost problem, we let $\mathsf{sc}_j = \mathsf{W}_j$ for each node. *Multiple* assigns 2 replicas, in s_1 and s_2, hence has cost $2n$. The *Upwards* policy assigns a replica to s_1 to cover its child, and then cannot use s_2 to process the requests of the child in its subtree. It must place a replica in s_3, hence a final cost $n + Kn = (K+1)n$ arbitrarily higher than *Multiple*.

3.4 Lower bound for the Replica Counting problem

Obviously, the cost of an optimal solution of the Replica Counting problem (for any policy) cannot be lower than the obvious lower bound $\left\lceil \frac{\sum_{i \in \mathcal{C}} r_i}{W} \right\rceil$, where W is the server capacity. Indeed, this corresponds to a solution where the total request load is shared as evenly as possible among the replicas.

The following instance of Replica Counting shows that the optimal cost can be arbitrarily higher than this lower bound. Consider Figure 5, with $n+1$ nodes of capacity $\mathsf{W}_j = W$. The root r has $n+1$ children, n nodes labeled s_1 to s_n, and a client with $r_i = W$. Each node s_j has a unique child, a client with

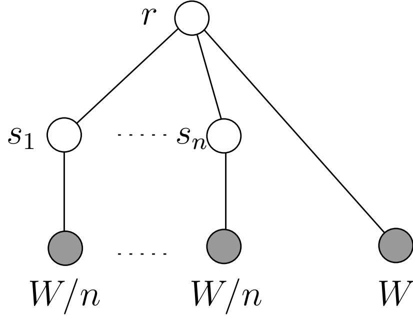

Figure 5. The lower bound cannot be approximated for Replica Counting**.**

$r_i = W/n$ (assume without loss of generality that W is divisible by n). The lower bound is $\left\lceil \frac{\sum_{i \in \mathcal{C}} r_i}{W} \right\rceil = \frac{2W}{W} = 2$. However, each of the three policies *Closest*, *Upwards* and *Multiple* will assign a replica to the root to cover its client, and will then need n extra replicas, one per client of s_j, $1 \leq j \leq n$. The total cost is thus $n+1$ replicas, arbitrarily higher than the lower bound.

All the examples in Sections 3.1 to 3.4 give an insight of the combinatorial nature of the Replica Placement optimization problem, even in its simplest variants Replica Cost and Replica Counting. The following section corroborates this insight: most problems are shown NP-hard, even though some variants have polynomial complexity.

4 Complexity results

One major goal of this paper is to assess the impact of the access policy on the problem with homogeneous vs heterogeneous servers. We consider a tree $\mathcal{T} = \mathcal{C} \cup \mathcal{N}$. Each client $i \in \mathcal{C}$ has r_i requests; each node $j \in \mathcal{N}$ has processing capacity W_j and storage cost $\mathsf{sc}_j = W_j$. The problem comes in two flavors, either the Replica Counting problem with homogeneous nodes ($\mathsf{W}_j = \mathsf{W}$ for all $j \in \mathcal{N}$), or the Replica Cost problem with heterogeneous nodes (servers with different capacities/costs).

In the single server version of the problem, we need to find a server $\mathsf{server}(i)$ for each client $i \in \mathcal{C}$. R is the set of replica, *i.e.* the servers chosen among the nodes in $\mathcal{N}$. The only constraint is that server capacities cannot be exceeded: this translates into

$$\sum_{i \in \mathcal{C}, \mathsf{server}(i)=j} r_i \leq W_j \quad \text{for all } j \in \mathcal{N}.$$

The objective is to find a valid solution of minimal storage cost $\sum_{j \in R} W_j$. As outlined in Section 3, there are two variants of the single server version of the problem, namely the *Closest* and the *Upwards* strategies.

In the *Multiple* policy with multiple servers per client, for any client $i \in \mathcal{C}$ and any node $j \in \mathcal{N}$, $r_{i,j}$ is the number of requests from i that are processed by j ($r_{i,j} = 0$ if $j \notin R$, and $\sum_{j \in \mathcal{N}} r_{i,j} = r_i$ for all $i \in \mathcal{C}$). The capacity constraint now writes

$$\sum_{i \in \mathcal{C}} r_{i,j} \leq W_j \quad \text{for all } j \in R,$$

while the objective function is the same as for the single server version.

The decision problems associated with the previous optimization problems are easy to formulate: given a bound on the number of servers (homogeneous version) or on the total storage cost (heterogeneous version), is there a valid solution that meets the bound?

	Homogeneous (Replica Counting)	**Heterogeneous** (Replica Cost)
Closest	polynomial [3, 9]	NP-complete
Upwards	NP-complete	NP-complete
Multiple	polynomial	NP-complete

Table 1. Complexity results for the different instances of the problem.

Table 1 captures the complexity results. These complexity results are all new, except for the *Closest*/Homogeneous combination.

The NP-completeness of the *Upwards*/Homogeneous case comes as a surprise, since all previously known instances were shown to be polynomial, using dynamic programming algorithms. In particular, the *Closest*/Homogeneous variant remains polynomial when adding communication costs [3] or QoS constraints [9]. We provide an elegant algorithm to show the polynomial complexity of the *Multiple*/Homogeneous problem.

Previous NP-completeness results involved general graphs rather than trees, and the combinatorial nature of the problem came from the difficulty to extract a good replica tree out of an arbitrary communication graph. Here the tree is fixed, but the problem remains combinatorial due to resource heterogeneity.

The proofs of these complexity results can be found in [1], together with the optimal polynomial algorithm for the *Multiple*/Homogeneous problem.

5. Linear programming formulation

In this section, we express the REPLICA PLACEMENT optimization problem in terms of an integer linear program. 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 [8].

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, 4]. 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. This bound will be very helpful to assess the performance of the polynomial heuristics that are introduced in Section 6.

5.1 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: $\forall j \in \mathcal{N}, \sum_{i \in \mathcal{C}} r_i y_{i,j} \le \mathsf{W}_j x_j$. Note that this ensures that if j is the server of i, there is indeed a replica located in node j. 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
$$\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i),$$
$$\forall i' \in \mathcal{C} \cap \mathsf{subtree}(j), \forall j' \in \mathsf{Ancestors}(j),$$
$$y_{i,j} \le 1 - y_{i',j'}$$
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
$$\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i) \setminus \{r\},$$
$$\forall i' \in \mathcal{C} \cap \mathsf{subtree}(j),$$
$$y_{i,j} \le 1 - z_{i',j \to \mathsf{parent}(j)}$$

5.2 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 are slightly modified:

$$\forall j \in \mathcal{N}, \sum_{i \in \mathcal{C}} y_{i,j} \leq \mathsf{W}_j x_j$$

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

5.3 A mixed integer LP-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. It has to 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 $s = |\mathcal{C}| + |\mathcal{N}| > 50$. Thus we cannot use this approach for large-scale problems.

However, we can still 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 [11], in practice using standard software packages [2, 4]), and the value of its objective function provides an absolute lower bound on the cost of any valid (integer) solution. Of course the relaxation makes the most sense for the *Multiple* policy, because several fractions of servers are assigned by the rational program. 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.

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 [4]. We used the refined bound for all our experiments.

6. Heuristics

In this section several heuristics for the *Closest*, *Upwards* and *Multiple* policies are presented. As previously stated, our main objective is to provide an experimental assessment of the relative performance of the three policies. All the eight heuristics described below have a worst case quadratic complexity $O(s^2)$, where $s = |\mathcal{C}| + |\mathcal{N}|$ is the problem size. Indeed, all heuristics proceed by traversing the tree, and the number of traversals is bounded by the number of internal nodes (and is much lower in practice).

1. Closest Top Down All (CTDA) – The basic idea is to perform a breadth-first traversal of the tree. Every time a node is able to process the requests of all the clients in its subtree, the node is chosen as a server,

and we do not explore further that subtree. The corresponding procedure (Algorithm 1) is called until no more servers are added in a tree traversal.

2. Closest Top Down Largest First (CTDLF) – The tree is traversed in breadth-first manner similarly to CTDA, but we treat the subtree which contains the most requests first. Also, the tree traversal is stopped each time a replica has been placed (and then the procedure is called again).

3. Closest Bottom Up (CBU) – Still dealing with the *Closest* policy, this heuristic performs a bottom-up traversal of the tree. A node is chosen as server if it can process all the requests of the clients in its subtree. Algorithm 2 describes the recursive implementation.

4. Upwards Top Down (UTD) – The top down approach works in two passes. In the first pass (see Algorithm 4), each node whose capacity is exhausted by the number of requests in its subtree is chosen by traversing the tree in depth-first manner. When a server is chosen, we delete as much clients as possible in non-increasing order of their r_i-values, until the server capacity is reached or no other client can be deleted. The delete-procedure is described in Algorithm 3. If not all requests can be treated by the chosen servers, a second pass is started. In this procedure (see Algorithm 5) servers with remaining requests are added.

5. Upwards Big Client First (UBCF) – The second heuristic for the *Upwards* policy works in a completely different way than all the other heuristics. The basic idea here is to treat all clients in non-increasing order of their r_i values. For each client we identify the server with minimal available capacity that can treat all its requests (see Algorithm 6).

6. Multiple Top Down (MTD) – The top-down approach for the *Multiple* policy is similar to UTD, with one significant difference: the delete-procedure (see Algorithm 7). For *Upwards*, requests of a client have to be treated by a single server, and it may occur that after the delete-procedure a server has still some capacity left to treat more requests, but all remaining clients have a higher amount of requests than this left-over capacity. For *Multiple*, requests of a client can be treated by multiple servers. So if at the end of the delete-procedure the server still has some capacity, we delete this amount of requests from the client with the largest r_i.

7. Multiple Bottom Up (MBU) – This heuristic is similar to MTD, except that we perform a bottom-up traversal of the tree in the first pass, and that the clients are deleted in non-decreasing order of their r_i-values. Algorithm 8 describes the first pass (servers with exhausted capacity). The second pass which adds extra servers if required is described in Algorithm 9.

8. Multiple Greedy (MG) – This last heuristic greedily allocates requests to servers in a bottom-up traversal of the tree, thus always finding a solution if there is one, but possibly at an expensive cost.

The pseudo-code for the algorithms is provided at the end of the paper.

7 Experiments

We have done some experiments to assess the impact of the different access policies, and the performance of these heuristics.

7.1 Experimental plan

The important parameter in our tree networks is the load, i.e. the total number of requests compared to the total processing power: $\lambda = \frac{\sum_{i\in\mathcal{C}} r_i}{\sum_{j\in\mathcal{N}} \mathsf{W}_i}$. We have performed experiments on 30 trees for each of the nine values of λ selected ($\lambda = 0.1, 0.2, ..., 0.9$). The trees have been randomly generated, with a problem size $15 \leq s \leq 400$. When λ is small, the tree has a light request load, while large values of λ implies a heavy load on the servers. We then expect the problem to have a solution less frequently.

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 the intractable problems.

To assess the performance of our heuristics, we have studied the relative performance of each heuristic compared to the lower bound. For each λ, results are computed on the trees for which the linear program has a solution. Let T_λ be the subset of trees with a 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 lower bound 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 who have a higher success rate, we set $cost_h(t) = +\infty$ if the heuristic did not find any solution.

Experiments have been conducted both on homogeneous networks (Replica Counting problem) and on heterogeneous ones (Replica Cost problem).

7.2 Results

A solution computed by a *Closest* or *Upwards* heuristic always is a solution for the *Multiple* policy,

since the latter is less constrained. Therefore, we can mix results into a new heuristic for the *Multiple* policy, called MixedBest (**MB**), which selects for each tree the best cost returned by the previous eight heuristics. Since MG never fails to find a solution if there is one, MB will neither fail either.

Experimental results are summarized in the figures provided at the end of the paper. Figure 6 shows the percentage of success of each heuristic for homogeneous platforms. The upper curve corresponds to the result of the linear program, and to the cost of the MG and MB heuristics, which confirms that they always find a solution when there is one. The UBU heuristic seems very efficient, since it finds a solution more often than MTD and MBU, the other two *Multiple* policies. On the contrary, UTD, which works in a similar way to MTD and MBU, finds less solutions than these two heuristics, since it is further constrained by the *Upwards* policy. As expected, all the *Closest* heuristics find fewer solutions as soon as λ reaches higher values: the bottom curve of the plot corresponds to CTDA, CTDLF and CBU, which all find the same solutions. This is inherent to the limitation of the *Closest* policy: when the number of requests is high compared to the total processing power in the tree, there is little chance that a server can process all the requests coming from its subtree, and requests cannot traverse this server to be served by a server located higher in the tree. These results confirm that the new policies have a striking impact on the existence of a solution to the Replica Counting problem.

Figure 7 represents the relative performance of the heuristics compared to the LP-based lower bound. As expected, the hierarchy between the policies is respected, i.e. *Multiple* is better than *Upwards* which in turn is better than *Closest*. Altogether, the use of the MixedBest heuristic MB allows to always pick up the best result, thereby resulting in a very satisfying relative cost for the *Multiple* instance of the problem. The greedy MG should not be used for small values of λ, but proves to be very efficient for large values, since it is the only heuristic to find a solution for such instances.

To conclude, we point out that MB always achieves a relative performance of at least 85%, thus returning a replica cost within 17% of that of the LP-based lower bound. This is a very satisfactory result for the absolute performance of our heuristics.

The heterogeneous results (see Figure 8 and Figure 9) are very similar to the homogeneous ones, which clearly shows that our heuristics are not much sensitive to the heterogeneity of the platform. Therefore, we have an efficient way to find in polynomial time a good solution to all the NP-hard problems stated in Section 4.

8. Related work

Early work on replica placement by Wolfson and Milo [13] has shown the impact of the write cost and motivated the use of a minimum spanning tree to perform updates between the replicas. In this work, they prove that the replica placement problem in a general graph is NP-complete, even without taking into account storage costs. Thus they address the case of special topologies, and in particular tree networks. They give a polynomial solution in a fully homogeneous case and a simple model with no QoS and no server capacity. Their work uses the closest server access policy (single server) to access the data.

Using this *Closest* policy, Cidon et al [3] studied an instance of the problem with multiple objects. In this work, the objective function has no update cost, but integrates a communication cost. Communication cost in the objective function 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 [9], adding a QoS in terms of a range limit, and the objective being the Replica Counting problem. In this latter approach, the servers are homogeneous, and their capacity is bounded.

Cidon et al [3] and Liu et al [9] both use the *Closest* access policy. In each case, the optimization problems are shown to have polynomial complexity. However, the variant with bidirectional links is shown NP-complete by Kalpakis et al [5]. Indeed in [5], requests can be served by any node in the tree, not just the nodes located in the path from the client to the root. The simple problem of minimizing the number of replicas with identical servers of fixed capacity, without any communication cost nor QoS contraints, directly reduces to the clasical bin packing problem.

Kalpakis et al [5] show that a special instance of the problem is polynomial, when considering no server capacities, but with a general objective function taking into account read, write and storage costs. In their work, a minimum spanning tree is used to propagate the writes, as was done in [13]. Different methods can however be used, such as a minimum cost Steiner tree, in order to further optimize the write strategy [6].

All papers listed above consider the *Closest* access policy. As already stated, most problems are NP-

complete, except for some very simplified instances. Karlsson et al [8, 7] compare different objective functions and several heuristics to solve these complex problems. They do not take QoS constraints into account, but instead integrate a communication cost in the objective function as was done in [3]. Integrating the communication cost into the objective function can be viewed as a Lagrangian relaxation of QoS constraints.

Tang and Xu [12] have been one of the first authors to introduce actual QoS constraints in the problem formalization. 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 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 a polynomial algorithm using dynamic programming.

To the best of our knowledge, there is no related work comparing different access policies, either on tree networks or on general graphs. Most previous works impose the *Closest* policy. The *Multiple* policy is enforced by Rodolakis et al [10] but in a very different context. In fact, they consider general graphs instead of trees, so they face the combinatorial complexity of finding good routing paths. Also, they assume an unlimited capacity at each node, since they can add numerous servers of different kinds on a single node. Finally, they include some QoS constraints in their problem formulation, based on the round trip time (in the graph) required to serve the client requests. In such a context, this (very particular) instance of the *Multiple* problem is shown to be NP-hard.

9. Conclusion

In this paper, we have introduced and extensively analyzed two important new policies for the replica placement problem. The *Upwards* and *Multiple* policies are natural variants of the standard *Closest* approach, and it may seem surprising that they have not already been considered in the published literature. On the theoretical side, we have fully assessed the complexity of the *Closest*, *Upwards* and *Multiple* policies, both for homogeneous and heterogeneous platforms. Not surprisingly, all three policies turn out to be NP-complete for heterogeneous nodes, which provides yet another example of the additional difficulties induced by resource heterogeneity. On the practical side, we have designed several heuristics for the *Closest*, *Upwards* and *Multiple* policies, and we have compared their performance. In the experiments, the constraints were only related to server capacities, and the total cost was the sum of the server capacities (or their number in the homogeneous case). Even in this simple setting, the impact of the new policies is impressive: (i) the number of trees which admit a solution is much higher with the *Upwards* and *Multiple* policies than with the *Closest* policy. (ii) for those problems which have a solution with the *Closest* policy, the replica cost is much lower for the other two policies. Finally, we point out that the absolute performance of the heuristics is quite good, since their cost is close to the lower bound based upon the solution of an integer linear program.

There remains much work to extend the results of this paper. In the short term, we need to conduct more simulations for the Replica Cost problem, varying the shape of the trees, the distribution law of the requests and the degree of heterogeneity of the platforms. We also aim at designing efficient heuristics for more general instances of the Replica Placement problem, taking more constraints into account. It will be instructive to see whether the superiority of the new *Upwards* and *Multiple* policies over *Closest* remains so important in the presence of QoS constraints. Also, including bandwidth constraints may require a better global load-balancing along the tree, thereby favoring *Multiple* over *Upwards*. In the longer term, designing efficient heuristics for the problem with various object types is a demanding algorithmic problem. Also, we would like to extend this work so as to handle more complex objective functions, including communication costs and update costs as well as replica costs; this seems to be a very difficult challenge to tackle.

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

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

[5] K. Kalpakis, K. Dasgupta, and O. Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. Parallel and Distributed Systems*, 12(6):628–637, 2001.

[6] K. Kalpakis, K. Dasgupta, and O. Wolfson. Steiner-Optimal Data Replication in Tree Networks with Storage Costs. In *IDEAS '01: Proceedings of the 2001*

International Symposium on Database Engineering & Applications, pages 285–293. IEEE Computer Society Press, 2001.

[7] M. Karlsson and C. Karamanolis. Choosing Replica Placement Heuristics for Wide-Area Systems. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.

[8] 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.

[9] 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.

[10] G. Rodolakis, S. Siachalou, and L. Georgiadis. Replicated server placement with QoS constraints. *IEEE Trans. Parallel Distributed Systems*, 17(10):1151–1162, 2006.

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

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

[13] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.*, 16(1):181–205, 1991.

Biographies

Anne Benoit is an ENSIMAG engineer (Applied Mathematics and Computer Science), she got her PhD in 2003 at the Polytechnical Institute of Grenoble (INPG). From 2003 to 2005, she was a Research Associate in the Institute for Computing Systems Architecture and Laboratory for Fundations of Computer Science of the School of Informatics at the University of Edinburgh, UK. She currently holds a position of Assistant Professor in the LIP laboratory at Ecole Normale Superieure in Lyon, France. Her research interests include algorithms design and scheduling techniques for parallel and distributed platforms, and also the performance evaluation of parallel systems and applications.

Veronika Rehn Veronika Rehn is currently a PhD student in the LIP laboratory at ENS Lyon. She is mainly interested in parallel algorithm and replica placement. She is a student member of the IEEE and the IEEE Computer Society.

Yves Robert received the PhD degree from Institut National Polytechnique de Grenoble. He is currently a full professor in the Computer Science Laboratory LIP at ENS Lyon. He is the author of four books, 95 papers published in international journals, and 120 papers published in international conferences. His main research interests are scheduling techniques and parallel algorithms for clusters and grids. He is a Fellow of the IEEE, and a Senior Member of the Institut Universitaire de France.

Algorithms

```
procedure CTDA (root, replica)
Fifo fifo;
fifo.push(root);
while fifo ≠ ∅ do
    s = fifo.pop();
    if s ∉ replica then
        if W_s ≥ inreq_s & inreq_s > 0 then
            replica = replica ∪ {s};
            foreach a ∈ Ancestors(s) do
            inreq_a = inreq_a − inreq_s;
        else
            foreach i ∈ children(s) do
                if i ∈ N then  fifo.push(i);
            end
        end
    end
end
```

Algorithm 1: Procedure CTDA

```
procedure CBU (s ∈ 𝒩, replica)
if atBottom(s) || allChildrenTreated(s) then
    treated_s = true;
    if W_s ≥ inreq_s & inreq_s > 0 then
        /* node can treat all children's requests */
        replica = replica ∪ {s};
        foreach a ∈ Ancestors(s) do
        inreq_a = inreq_a − inreq_s;
    else
        /* node cannot treat all children's
        requests, go up in the tree */
        if Ancestors(s) ≠ ∅ then  call CBU
        (parent(s), replica);
    end
else
    foreach i ∈ children(s) do
        /* not yet at the bottom of the tree, go
        down */
        if i ∈ 𝒩 & treated_i then  call CBU (i,
        replica);
    end
end
```

Algorithm 2: Procedure CBU

```
procedure deleteRequests (s ∈ 𝒩,
numToDelete)
clientList = sortDecreasing(clients(s));
foreach i ∈ clientList do
    if r_i ≤ numToDelete then
        numToDelete = numToDelete - r_i;
        foreach a ∈ Ancestors(i) do
        inreq_a = inreq_a − r_i;
        children(parent(i)) =
        children(parent(i)) \ {i};
        if numToDelete == 0 then  return;
    end
end
```

Algorithm 3: Procedure deleteRequests

```
procedure UTDFirstPass (s ∈ 𝒩, replica)
if inreq_s ≥ W_s & inreq_s > 0 then
    replica = replica ∪ {s};
    treated_s = true;
    deleteRequests(s, W_s);
end
foreach i ∈ children(s) do
    if i ∈ 𝒩 then  UTDFirstPass (i, replica);
end
```

Algorithm 4: Procedure UTDFirstPass

```
procedure UTDSecondPass (s ∈ 𝒩, replica)
if s ∉ replica & inreq_s > 0 then
    replica = replica ∪ {s};
    deleteRequests(s, inreq_s);
else
    foreach i ∈ children(s) do
        if i ∈ 𝒩 & inreq_i > 0 then
        UTDSecondPass (i, replica);
    end
end
```

Algorithm 5: Procedure UTDSecondPass

```
procedure UBU (s ∈ 𝒩, replica)
clientList = sortDecreasing(clients(s));
foreach i ∈ clientList do
    ValidAncests = {a ∈ Ancestors(i)|W_a ≥ r_i};
    if ValidAncests ≠ ∅ then
        a = Min_{W_j}{j ∈ ValidAncests};
        if a ∉ replica then
        replica = replica ∪ {a};
        W_a = W_a − r_i;
    end
    else return no solution;
end
```

Algorithm 6: Procedure UBU

```
procedure deleteRequestsInMTD (s ∈ 𝒩,
numToDelete)
clientList = sortDecreasing(clients(s));
foreach i ∈ clientList do
    if r_i ≤ numToDelete then
        numToDelete = numToDelete - r_i;
        foreach a ∈ Ancestors(i) do
        inreq_a = inreq_a − r_i;
        children(parent(i)) =
        children(parent(i)) \ {i};
    else
        r_i = r_i - numToDelete;
        foreach a ∈ Ancestors(i) do
        inreq_a = inreq_a − r_i;
        return;
    end
end
```

Algorithm 7: Procedure deleteRequestsInMTD

```
procedure MBUFirstPass (s ∈ N, replica)
if atBottom(s) || allChildrenTreated(s) then
    treated_s = true;
    if W_s ≤ inreq_s & inreq_s > 0 then
        /* node is exhausted by the requests of its
        clients */
        replica = replica ∪ {s};
        deleteRequestsInMBU(s, W_s);
    else
        /* node is not exhausted, go up the tree */
        if Ancestors(s) ≠ ∅ then call MBU
        (parent(s), replica);
    end
else
    /* not yet at the bottom of the tree, go down
    */
    foreach i ∈ children(s) do
        if i ∈ N & treated_i then call MBU (i,
        replica);
    end
end
```

Algorithm 8: Procedure MBUFirstPass

```
procedure MBUSecondPass (s ∈ N, replica)
if s ∉ replica & inreq_s > 0 then
    replica = replica ∪ {s};
    deleteRequestsInMBU(s, inreq_s);
else
    foreach i ∈ children(s) do
        if i ∈ N & inreq_i > 0 then
        UTDSecondPass (i, replica);
    end
end
```

Algorithm 9: Procedure MBUSecondPass

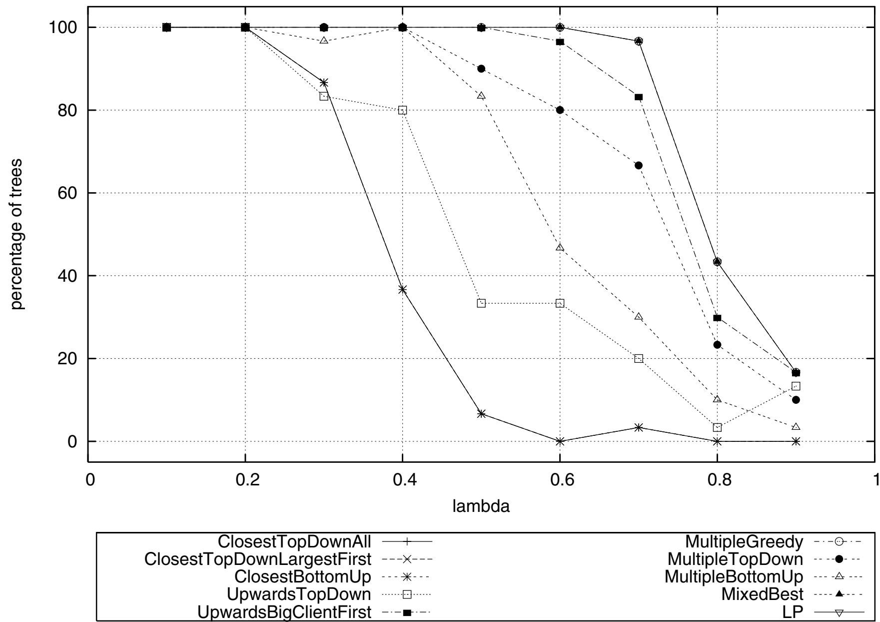

Figure 6. Homogeneous case - Percentage of success.

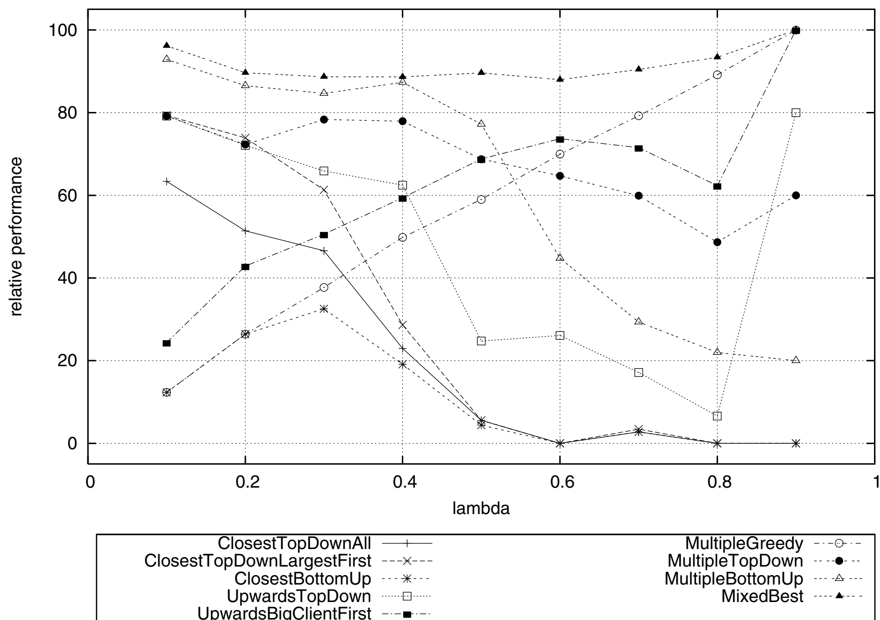

Figure 7. Homogeneous case - Relative performance.

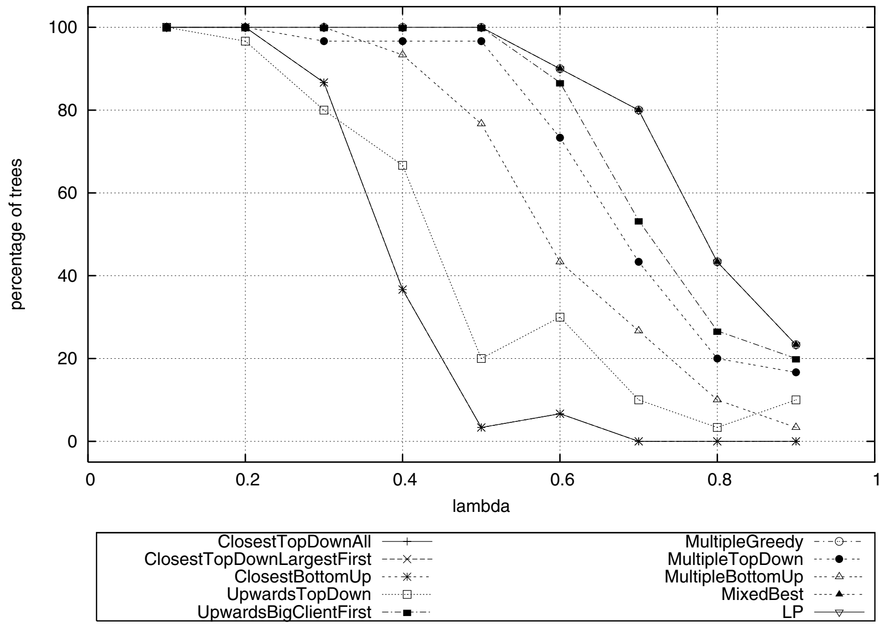

Figure 8. Heterogeneous case - Percentage of success.

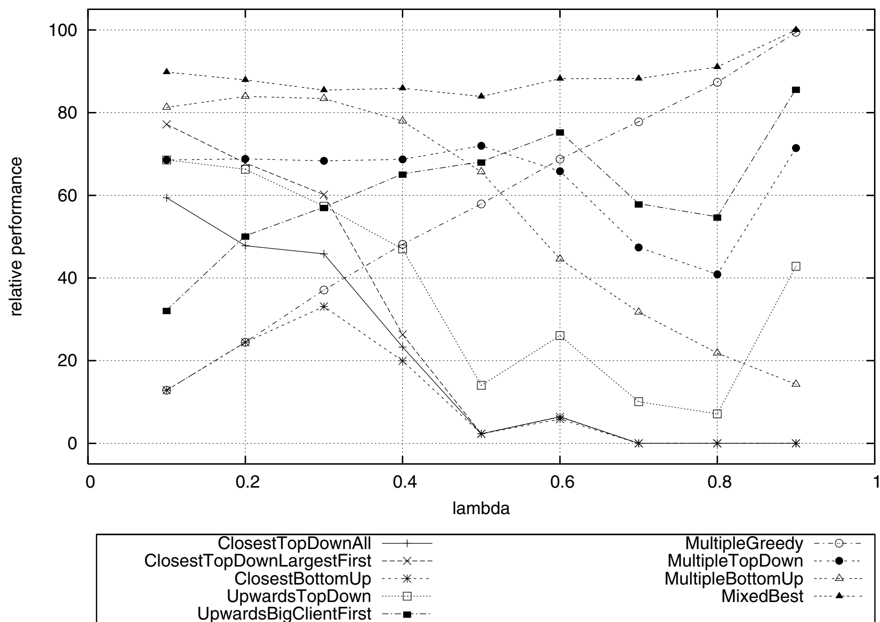

Figure 9. Heterogeneous case - Relative performance.