Replica Placement and Access Policies in Tree Networks

Anne Benoit, *Member*, *IEEE*, Veronika Rehn-Sonigo, *Student Member*, *IEEE*, and Yves Robert, *Fellow*, *IEEE*

Abstract—In this paper, we discuss and compare several policies to place replicas in tree networks, subject to server capacity and Quality-of-Service (QoS) 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 one to the other, and their absolute performance is assessed by comparison with the optimal solution provided by an integer linear program.

Index Terms—Replica placement, tree networks, access policy, scheduling, complexity results, heuristics, linear program, heterogeneous clusters.

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 is 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 a knowledge only of its parent and children in the tree.

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 per time unit is equivalent to two replicas on nodes of capacity 100 each).

• *The authors are with the Laboratoire de l'Informatique du Parallellsme (UMR 5668, ENS Lyon—CNRS—INRIA—UCBL), Ecole Normale Supérieure Lyon, University of Lyon, 69364 Lyon, France. E-mail: {Anne.Benoit, Veronika.Sonigo, Yves.Robert}@ens-lyon.fr.*

Manuscript received 13 July 2007; revised 22 Nov. 2007; accepted 21 Jan. 2008; published online 1 Feb. 2008.
Recommended for acceptance by R. Eigenmann.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2007-07-0233.
Digital Object Identifier no. 10.1109/TPDS.2008.25.

The core of the paper is devoted to the study of the previous optimization problem, called REPLICA PLACEMENT in the following. Additional constraints are introduced, in order to guarantee some Quality of Service (QoS): requests must be served in limited time, thereby prohibiting remote or hard-to-reach replica locations. We focus on optimizing the total utilization cost (or replica number in the homogeneous case). There is a bunch of possible extensions: dealing with several object types rather than one, including communication time into the objective function, enforcing additional bandwidth constraints on each link, taking into account an update cost of the replicas, and so on. For the sake of clarity, we devote a special section (Section 8) to formulate these extensions and to describe which situations our results and algorithms can still apply to.

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 [1], [2], [3], [4]. 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, and we survey related work in Section 9. The objective of this paper is threefold: 1) introducing two new access policies and comparing them with the standard approach, 2) assessing the impact

1045-9219/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

of server heterogeneity on the problem, and 3) assessing the impact of QoS constraints.

In most, if not all, papers from the literature, 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 approaches: 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 *Upward* policy. The trade-off to explore is the following: the *Closest* policy assigns replicas at a proximity of the clients but may need to allocate too many of them if some local subtree issues a great number of requests. The *Upward* policy will ensure better resource usage, load-balancing the process of requests on a larger scale; the possible drawback is that requests will be served by remote servers, likely to take longer time to process them. Taking QoS constraints into account would typically be more important for the *Upward* 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. The only drawback is the (modest) additional complexity induced by the fact that requests must now be tagged with the replica server ID in addition to the client ID. As already stated, one major objective of this paper is to compare these three access policies, *Closest*, *Upward*, and *Multiple*.

The second major contribution of the paper is to assess the impact of server heterogeneity and QoS constraints, 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 [2], [3], [4]. In this paper, we establish several new complexity results. Those for the homogeneous case are surprising: for the simplest instance without QoS, the *Multiple* policy is polynomial (as *Closest*), while *Upward* is NP-hard. With the addition of QoS constraints, the *Multiple* policy becomes NP-complete for the homogeneous case, and only *Closest* remains polynomial. The three policies turn out to be NP-complete for heterogeneous nodes, even without QoS, which provides yet another example of the additional difficulties induced by resource heterogeneity.

On the more practical side, we provide an optimal algorithm for the *Multiple* problem with homogeneous nodes and several heuristics for all three policies in the heterogeneous case. We compare these heuristics through simulations conducted for problem instances without or with QoS constraints. Another contribution is that we are able to assess the absolute performance of the heuristics, not just comparing one to the other, owing to an optimal solution for *Multiple* provided by a new formulation of the REPLICA PLACEMENT problem in terms of an integer linear program. The solution of this program allows us to build an optimal solution for reasonably large problem instances.

The rest of the paper is organized as follows: Section 2 is devoted to a detailed presentation of the target optimization problems. In Section 3, we introduce the three access policies, and we give a few motivating examples. Next, in Section 4, we proceed to the complexity results for the REPLICA PLACEMENT problem. Section 5 deals with the formulation of the REPLICA PLACEMENT problem in terms of an integer linear program. In Section 6, we introduce several polynomial heuristics to solve the REPLICA PLACEMENT problem with the different access policies. These heuristics are compared through simulations, whose results are analyzed in Section 7. Section 8 discusses various extensions to the REPLICA PLACEMENT problem, while Section 9 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 10.

2 Framework

This section is devoted to a precise statement of the REPLICA PLACEMENT optimization problem. We start with some definitions and notations. Next, we outline different problem instances that we study in this paper.

2.1 Definitions and Notations

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 requests to database objects. For the sake of clarity, we restrict the presentation to a single object type, hence, a single database. We deal with several object types in Section 8.

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 requests 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.

Let r be the root of the tree. If $j \in \mathcal{N}$, then $\mathrm{children}(j)$ is the set of children of node j. If $k \neq r$ is any node in the tree (leaf or internal), $\mathrm{parent}(k)$ is its parent in the tree. If $l : k \rightarrow k' = \mathrm{parent}(k)$ is any link in the tree, then $\mathrm{succ}(l)$ is the link $k' \rightarrow \mathrm{parent}(k')$ (when it exists). Let $\mathrm{Ancestors}(k)$ 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 $k' \in \mathrm{Ancestors}(k)$, then $\mathrm{path}[k \rightarrow k']$ denotes the set of links in the path from k to k'; also, $\mathrm{subtree}(k)$ is the subtree rooted in k, including k. We introduce more notations to describe our system in the following:

- **Clients** $i \in \mathcal{C}$. Each client i (leaf of the tree) is sending r_i requests per time unit. For such requests, the required QoS (typically, a response time) is denoted q_i, and we need to ensure that this QoS will be satisfied for each client.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

- **Nodes** $j \in \mathcal{N}$. Each node j (internal node of the tree) has a processing capacity W_j, which is the total number of requests that it can process per time unit when it has a replica. A cost is also associated to each node, sc_j, which represents the price to pay to place a replica at this node. With a single object type, it is quite natural to assume that sc_j is proportional to W_j: the more powerful a server, the more costly. However, with several objects we may use non-related values of capacity and cost.
- **Communication links** $l \in \mathcal{L}$. The edges of the tree represent the communication links between nodes (leaf and internal). We assign a communication time comm_l on link l, which is the time required to send a request through the link.

2.2 Problem Instances

For each client $i \in \mathcal{C}$, let $\mathrm{Servers}(i) \subseteq \mathcal{N}$ be the set of servers responsible for processing at least one of its requests. We do not specify here that access policy is enforced (e.g., one or multiple servers), we defer this to Section 3. Instead, we let $r_{i,s}$ be the number of requests from client i processed by server s (of course, $\sum_{s \in \mathrm{Servers}(i)} r_{i,s} = r_i$). In the following, R is the set of replicas:

$$R = \{s \in \mathcal{N} \mid \exists i \in \mathcal{C},\ s \in \mathrm{Servers}(i)\}.$$

2.2.1 Constraints

- **Server capacity.** The constraint that no server capacity can be exceeded is present in all variants of the problem:

$$\forall s \in R, \sum_{i \in \mathcal{C} | s \in \mathrm{Servers}(i)} r_{i,s} \leq \mathrm{W}_s.$$

- **QoS.** Some problem instances enforce a quality of service: the time to transfer a request from a client to a replica server is bounded by a quantity q_i. This translates into

$$\forall i \in \mathcal{C}, \forall s \in \mathrm{Servers}(i), \sum_{l \in \mathrm{path}[i \to s]} \mathrm{comm}_l \leq \mathrm{q}_i.$$

  Note that it would be easy to extend the QoS constraint so as to take the computation cost of a request in addition to its communication cost. This former cost is directly related to the computational speed of the server and the amount of computation (in flops) required for each request.

2.2.2 Objective Function

The objective function for the REPLICA PLACEMENT problem is defined as

$$\mathrm{Min} \sum_{s \in R} \mathrm{sc}_s.$$

As already pointed out, it is frequently assumed that the cost of a server is proportional to its capacity, so, in most problem instances, we let $\mathrm{sc}_s = \mathrm{W}_s$.

2.2.3 Simplified Problems

We define a few simplified problem instances in the following:

- **QoS=distance.** We can simplify the expression of the communication time in the QoS constraint and only consider the distance (in number of hops) between a client and its server(s). The QoS constraint is then

$$\forall i \in \mathcal{C}, \forall s \in \mathrm{Servers}(i),\ d(i,s) \leq \mathrm{q}_i,$$

  where the distance $d(i,s) = |\mathrm{path}[i \to s]|$ is the number of communication links between i and s.
- **No QoS.** We may further simplify the problem by completely suppressing the QoS constraints. In this case, the servers can be anywhere in the tree, their location is indifferent to the client. The problem reduces to finding a valid solution of minimal cost, where "valid" means that no server capacity is exceeded. We name REPLICA COST this fundamental problem.
- **Replica counting.** We can further simplify the previous REPLICA COST 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 $\mathrm{server}(i)$, that is responsible for processing all its requests.
- **Multiple servers.** A client i may be assigned several servers in a set $\mathrm{Servers}(i)$. Each server $s \in \mathrm{Servers}(i)$ will handle a fraction $r_{i,s}$ of the requests.

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 nonoptimal 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 $\mathrm{server}(i)$. Then, any other client node i' residing in the subtree rooted in $\mathrm{server}(i)$ will be assigned a server in that subtree. This forbids requests from i' to "traverse" $\mathrm{server}(i)$ and be served higher (closer to the root) in the tree.

We relax this constraint in the *Upward* policy, which is the general single server policy. Notice that a solution to *Closest* always is a solution to *Upward*, thus *Upward* is always better than *Closest* in terms of the objective function.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

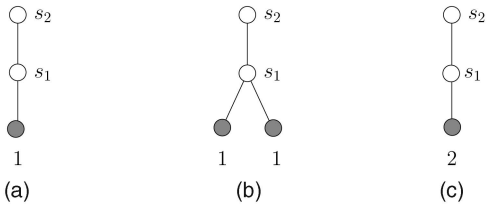

Fig. 1. Access policies.

Similarly, the *Multiple* policy is always better than *Upward*, 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 *Upward* can be arbitrarily better than *Closest*, while Section 3.3 shows that *Multiple* can be arbitrarily better than *Upward*. 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 Fig. 1). Each node can process $\mathrm{W} = 1$ request.

- If s_1 has one client child making one request, the problem has a solution with all three policies, placing a replica on s_1 or on s_2 indifferently (Fig. 1a).
- If s_1 has two client children, each making one request, the problem has no more solution with *Closest*. However, we have a solution with both *Upward* and *Multiple* if we place replicas on both nodes. Each server will process the request of one of the clients (Fig. 1b).
- Finally, if s_1 has only one client child making two 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 (Fig. 1c).

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

3.2 Upward versus Closest

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

With the *Upward* 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:

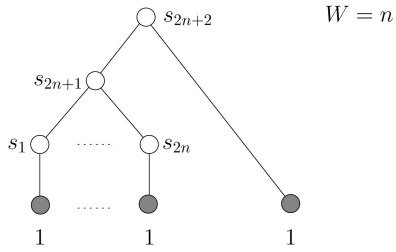

Fig. 2. *Upward* versus *Closest*.

- Either we place a replica on s_{2n+1}. In this case, this replica is handling n requests, but there remain 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 $s_1 \dots s_{2n}$.
- 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 $s_1, s_2, \dots, s_{2n}$.

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

3.3 Multiple versus Upward

In this section, we build an instance of the REPLICA COUNTING problem, where *Multiple* is twice better than *Upward*. We do not know whether there exist instances of REPLICA COUNTING where the performance ratio of *Multiple* versus *Upward* 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 *Upward*.

We start with the homogeneous case. Consider the instance of REPLICA COUNTING represented in Fig. 3, with $3n+1$ nodes of capacity $\mathrm{W}_j = W = 2n$. 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 $1 \le j \le n$. 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.

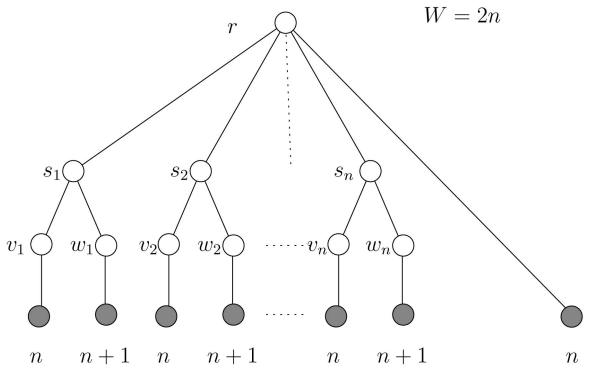

Fig. 3. *Multiple* versus *Upward*, homogeneous platforms.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

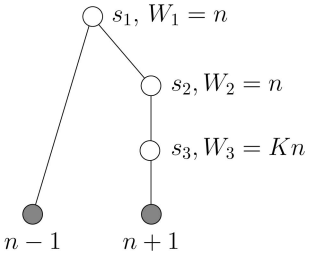

Fig. 4. *Multiple* versus *Upward*, heterogeneous platforms.

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 *Upward* 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.

We now proceed to the heterogeneous case. Consider the instance of REPLICA COST represented in Fig. 4, with three nodes s_1, s_2, and s_3, and two clients. The capacity of s_1 and s_2 is $\mathrm{W}_1 = \mathrm{W}_2 = n$ while that of s_3 is $\mathrm{W}_3 = Kn$, where K is arbitrarily large. Recall that in the REPLICA COST problem, we let $\mathrm{sc}_j = \mathrm{W}_j$ for each node. *Multiple* assigns two replicas, in s_1 and s_2 and hence has cost $2n$. The *Upward* policy assigns a replica to s_1 to cover its child, and then, it 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 Fig. 5, with $n+1$ nodes of capacity $\mathrm{W}_j = W$. 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 $r_i = W/n$ (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*, *Upward*, 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, $1 \leq j \leq n$.

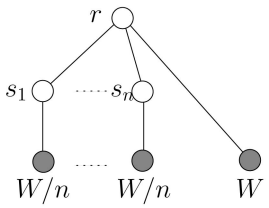

Fig. 5. The lower bound cannot be approximated for REPLICA COUNTING.

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 versus heterogeneous servers and without QoS versus with QoS. We restrict to the simplest problems, namely, the REPLICA COST and REPLICA COUNTING problems introduced in Section 2.2.3, and with or without QoS constraints, limited to the **QoS=distance** problem. We consider a tree $\mathcal{T} = \mathcal{C} \cup \mathcal{N}$ with or without QoS constraints. Each client $i \in \mathcal{C}$ has r_i requests; each node $j \in \mathcal{N}$ has processing capacity W_j and storage cost $\mathrm{sc}_j = W_j$. This problem comes in two flavors, either with homogeneous nodes ($\mathrm{W}_j = \mathrm{W}$, for all $j \in \mathcal{N}$; REPLICA COUNTING) or with heterogeneous nodes (servers with different capacities/costs; REPLICA COST).

In the single server version of the problem, we need to find a server $\mathrm{server}(i)$ 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 constraint is that server capacities cannot be exceeded: this translates into

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

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 *Upward* 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.

For each of those problems, we can add the QoS=distance constraint as specified in Section 2.2.3.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

TABLE 1
Complexity Results

	Replica Counting Homogeneous		Replica Cost Heterogeneous
	No QoS	**With QoS**	**No/With QoS**
Closest	polynomial [2], [4]	polynomial [4]	NP-complete
Upward	NP-complete	NP-complete	NP-complete
Multiple	polynomial	NP-complete	NP-complete

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?

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

The NP-completeness of the *Upward*/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 [2] or QoS constraints [4]. We provide an elegant algorithm to show the polynomial complexity of the *Multiple*/Homogeneous problem. Another important contribution of this paper is the NP-completeness of the *Multiple* policy with QoS constraints for the homogeneous case, which gives a clear insight on the additional complexity introduced by QoS constraints. In addition, all problems become NP-complete when dealing with resource heterogeneity (REPLICA COST problem).

Note that 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 QoS or resource heterogeneity.

4.1 Multiple/Homogeneous/No-QoS

Theorem 1. *The instance of the REPLICA COUNTING problem without QoS and with the* Multiple *strategy can be solved in polynomial time.*

Proof. An algorithm is provided to solve the problem, together with the proof of its optimality (which is quite technical). Please refer to [5] or to the supplemental material on the Web, where we also outline a detailed example to illustrate the step-by-step execution of the algorithm. □

4.2 NP-Completeness Results

Theorem 2. *The instance of the REPLICA COUNTING problem with the* Upward *strategy is NP-complete in the strong sense.*

Theorem 3. *The instance of the REPLICA COUNTING problem with QoS constraints and the* Multiple *strategy is NP-complete.*

This last result is interesting since the same problem with no QoS was polynomial. Less surprisingly, the following theorem assesses the complexity of the problem with heterogeneous resources.

Theorem 4. *All three instances of the REPLICA COST problem with heterogeneous nodes are NP-complete.*

Detailed proofs of these theorems are available in [5] and [6] and in the supplemental material on the Web.

5 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 server capacities. We derive a formulation for each of the three server access policies, namely, *Closest*, *Upward*, and *Multiple*. This is an important extension to a previous formulation due to [7].

5.1 Single Server

We start with single server strategies, namely, the *Upward* 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 $j = \text{server}(i)$.
- If $j \notin \text{Ancestors}(i)$, we directly set $y_{i,j} = 0$.

Link assignment

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

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

- Every client is assigned a server:

$$\forall i \in \mathcal{C}, \sum_{j \in \text{Ancestors}(i)} y_{i,j} = 1.$$

- All requests from $i \in \mathcal{C}$ use the link to its parent: $z_{i,i \to \text{parent}(i)} = 1$.
- Let $i \in \mathcal{C}$ and consider any link $l : j \to j' = \text{parent}(j) \in \text{path}[i \to r]$. If $j' = \text{server}(i)$, then link $\text{succ}(l)$ is not used by i (if it exists). Otherwise, $z_{i,\text{succ}(l)} = z_{i,l}$. Thus, $\forall i \in \mathcal{C}$, $\forall l : j \to j' = \text{parent}(j) \in \text{path}[i \to r]$, $z_{i,\text{succ}(l)} = z_{i,l} - y_{i,j'}$.

Next, there are constraints expressing that the processing capacity of any server cannot be exceeded:

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

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:

$$\forall i \in \mathcal{C}, \forall j \in \text{Ancests}(i), \text{dist}(i,j) y_{i,j} \leq \text{q}_i,$$

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

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

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

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 $\forall i \in \mathcal{C}$, $\forall j \in \text{Ancestors}(i)$, $\forall i' \in \mathcal{C} \cap \text{subtree}(j)$, $\forall j' \in \text{Ancestors}(j)$, and $y_{i,j} \leq 1 - y_{i',j'}$. Indeed, if $y_{i,j} = 1$, meaning that $j = \text{server}(i)$, 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 $\forall i \in \mathcal{C}$, $\forall j \in \text{Ancestors}(i) \setminus \{r\}$, $\forall i' \in \mathcal{C} \cap \text{subtree}(j)$, $y_{i,j} \leq 1 - z_{i',j \to \text{parent}(j)}$.

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 $j \notin \text{Ancestors}(i)$, 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 $l \in \text{path}[i \to r]$ when client i accesses any of its servers in $\text{Servers}(i)$.
- If $l \notin \text{path}[i \to r]$, we directly set $z_{i,l} = 0$.

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

- Every request is assigned a server:

$$\forall i \in \mathcal{C}, \sum_{j \in \text{Ancestors}(i)} y_{i,j} = r_i.$$

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

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

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

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

5.3 An ILP-Based Lower Bound

The previous linear program contains 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, and there is no efficient algorithm to solve integer linear programs (unless P = NP). For each access policy, there is a large number of variables, and 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, this formulation is extremely useful as it leads to an absolute lower bound: we can solve the integer linear program over the rationals. In this case, all constraints are relaxed, and we assume that all variables can take rational values. The optimal solution of the relaxed program can be obtained in polynomial time (in theory, using the ellipsoid method [8] and, in practice, using standard software packages [9], [10]), 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 10 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 percent 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, 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 [10]. 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.

5.4 An Exact MIP-Based Solution for Multiple

Theorem 5. *The solution of the linear program detailed in Section 5.2, when solved with all variables being rational except 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.*

Detailed proof of this theorem is available in [6] and in the supplemental material on the Web.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

6 Heuristics for the Replica Cost Problem

In this section, several heuristics for the *Closest*, *Upward*, and *Multiple* policies are presented. As previously stated, our main objective is to provide an experimental assessment of the relative performance of these three access policies.

In a first step, we target heterogeneous trees without QoS constraints, thus considering the REPLICA COST problem. Then, we analyze the impact of QoS constraints on the replica costs achieved by each policy. In this case, we consider the simplified problem QoS=distance in which the quality of service of a client is the number of hops that requests from this client are allowed to traverse until they reach their server(s).

All the heuristics described below have a worst case quadratic complexity $O(s^2)$, where $s = |\mathcal{C}| + |\mathcal{N}|$ is the problem size. Indeed, most of the 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).

We assume that each node $k \in \mathcal{N} \cup \mathcal{C} \setminus \{root\}$ knows its $\mathrm{parent}(k)$. Additionally, an internal node $j \in \mathcal{N}$ knows its $\mathrm{children}(j)$ and the set $\mathrm{clients}(j)$ of the clients in its subtree $\mathrm{subtree}(j)$. At any step of the heuristics, we denote by inreq_j the number of requests in $\mathrm{subtree}(j)$, reaching j with the current replicas already placed (initially, with no replica, $\mathrm{inreq}_j = \sum_{i \in \mathrm{clients}(j)} r_i$). We use a Boolean variable $\mathrm{treated}_j$ to mark if a node j has been treated during a tree traversal. The set of replicas is initialized by $\mathrm{replica} = \emptyset$.

The pseudocode of all heuristics is available in [5] and in the supplemental material on the Web. Moreover, the code of all heuristics can be found on the Web [11].

6.1 Without QoS Constraints

The first set of heuristics is not taking QoS constraints into account. In the following, inreq_j denotes the amount of requests that reach an inner node j.

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 further explore that subtree. The procedure is called until no more servers are added in a tree traversal.

Closest Top Down Largest First (CTDLF). The tree is traversed in breadth-first manner as in CTDA. However, we treat the subtree, which contains the most requests first when considering the children of the tree. In addition, instead of adding all possible servers in a single step, the tree traversal is stopped as soon as a server that can process all the requests in its subtree has been found, and it starts from the root again. Thus, CTDLF is called exactly $|R|$ times, where R is the final set of replica.

Closest Bottom Up (CBU). The last heuristic for the *Closest* policy performs a bottom-up traversal of the tree. A node is chosen as a server if it can process all the requests of the clients in its subtree. The procedure is initially called with the root of the tree; while we do not reach the bottom of the tree, we go down. Once arrived at the bottom, i.e., when the current node s has only clients as children or when all its children have already been treated, the node is marked as treated and added to the set replica if $W_s \geq \mathrm{inreq}_s$. Then, we go up in the tree until all nodes are treated, performing recursive calls.

Each of these three heuristics is placing a number of replicas, but none is ensuring whether a valid solution has been found or not. We need to check the final value of inreq_{root}. If there still are some pending requests at the root, there is no valid solution. However, if $\mathrm{inreq}_{root} = 0$, the heuristic has found a solution.

Upward Top Down (UTD). The top down approach works in two passes. In the first pass, each node $s \in \mathcal{N}$ whose capacity is exhausted by the number of requests in its subtree $(W_s \leq \mathrm{inreq}_s)$ is chosen by traversing the tree in depth-first manner. When a server is chosen, we delete as much clients as possible in nonincreasing order of their number of requests r_i, until the server capacity is reached or no other client can be deleted. If not all requests can be treated by the chosen servers, a second pass is started. In this *UTDSecondPass*-procedure, servers with remaining requests are added. Note that all these servers are nonexhausted by the remaining requests $(\mathrm{inreq}_s < W_s)$. These two procedures are each called only once, with $s = root$ as a parameter. Similar to the *Closest* heuristics, we need to check that $\mathrm{inreq}_{root} = 0$ at the end of UTD to find out whether a valid solution has been found.

Upward Big Client First (UBCF). The second heuristic for the *Upward* policy works in a completely different way than all the other heuristics in this section. The basic idea here is to treat all clients in nonincreasing order of their r_i values. For each client, we identify the server with minimal current capacity (in the path from the client to the root) that can treat all its requests. The capacity of a server is decreased each time it is assigned some requests to process. If there is no valid server to assign to a given client, the heuristic has failed to find a valid solution.

Multiple Top Down (MTD). The top-down approach for the *Multiple* policy is similar to the top-down approach for *Upward*, with one significant difference: the delete procedure. For *Upward*, requests of a client have to be treated by a single server, and it may occur that after the delete procedure a server still has some capacity left to treat more requests, but all remaining clients have a higher amount of requests than this leftover capacity. For *Multiple*, requests of a client can be treated by multiple servers. Therefore, 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.

Multiple Bottom Up (MBU). The first pass of this heuristic performs a bottom-up traversal of the tree, as in CBU. During this traversal, nodes $s \in \mathcal{N}$ are added to the set replica if their capacity is exhausted $(W_s \leq \mathrm{inreq}_s)$, similar to the first pass of the MTD procedure. The delete procedure is identical to the MTD delete procedure, except that clients are deleted in a nondecreasing order of their r_i values (instead of the nonincreasing order). Intuitively, we aim at deleting many small clients rather than fewer demanding ones.

Multiple Greedy (MG). The last heuristic performs a greedy bottom-up assignment of requests. We add a replica whenever there are some requests affected to a server. For heterogeneous platforms, we may often return a cost far

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

from the optimal, but we ensure that we always find a solution to the problem if there exists one. It might be particularly interesting to use MG only for problem instances for which MBU or MTD fail to find a solution.

Mixed Best (MB). This heuristic unifies all previous ones: for each tree, we select the best cost returned by the previous heuristics. In fact, since each solution for *Closest* is also a solution for *Upward*, which in turn is a valid solution for *Multiple*, this heuristic provides a solution for the *Multiple* policy.

6.2 With QoS Constraints

We now add QoS constraints to the clients. In the following, we denote by $\mathrm{inreqQoS}_j$ the amount of requests that reach an inner node j within their QoS constraints and by inreq_j the total amount of requests that reach j (including requests whose QoS constraints are violated). In this set of heuristics, the difficulty is to find a good trade-off between favoring clients with a large number of requests and clients with a very constrained QoS.

Closest Big Subtree First (CBS). Here, we traverse the tree in top-down manner. We place a replica on an inner node j if $\mathrm{inreqQoS}_j \leq \mathrm{W}_j$. When the condition holds, we do not process any other subtree of j. If this condition does not hold, we process the subtrees of j in nonincreasing order of inreq_j. 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 a nondecreasing order of q_i. In case of a tie, clients are sorted by a nonincreasing order of r_i. For each client, we look for the server that can process its subtree $(\mathrm{inreqQoS}_j \leq \mathrm{W}_j)$ and that 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.

Upward Small QoS Started Servers First (USQoSS). Clients are sorted by a nondecreasing order of q_i (and a nonincreasing 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 $\mathrm{W}_j \geq r_i$ within the QoS-range, and we place a replica in j. If we still do not find any appropriate node, then this heuristic has no feasible solution.

Upward 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 $(\mathrm{W}_j - \mathrm{inreqQoS}_j)$-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.

Upward Minimal Distance (UMD). This heuristic requires two steps. In the first step, so-called indispensable servers are chosen, i.e., inner nodes that have a client that must be treated by this very node. At the beginning, all servers that have a child client with $\mathrm{q} = 1$ 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 $(\mathrm{W}_j - \mathrm{inreqQoS}_j)$-value as server (if $\mathrm{inreqQoS}_j > 0$). Note that this value can be negative. Then, clients are associated to this server in order of distance, i.e., clients that are close 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 first inner node on the way up to the root that 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, $\mathrm{W}_i < r_c$), 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 a nondecreasing order of q_i, and the appropriate servers j are chosen by minimal $(\mathrm{W}_j - \mathrm{inreqQoS}_j)$-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 $(\mathrm{W}_j - \mathrm{inreqQoS}_j)$-value, but this time, clients are associated in a nondecreasing order of $\min(\mathrm{q}_i, d(i,r))$, 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 as for the case without QoS.

7 Experiments

We have done several experiments to assess the impact of the different access policies, the impact of QoS constraints, and the performance of the polynomial heuristics described in Section 6. We outline the experimental plan in Section 7.1. Results without QoS constraints are given and analyzed in Section 7.2, while results with QoS constraints are provided in Section 7.3. In the following, we denote by s the problem size: $s = |\mathcal{C}| + |\mathcal{N}|$.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

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:

$$\lambda = \frac{\sum_{i \in \mathcal{C}} r_i}{\sum_{j \in \mathcal{N}} \mathrm{W}_j}.$$

We have performed experiments on 30 trees for each of the nine values of λ selected ($\lambda = 0.1, 0.2, \ldots, 0.9$). The trees have been randomly generated, with a problem size $15 \leq s \leq 400$. Each node may have up to five children, where the exact number of children is also randomly determined. The value for the server capacity of the inner nodes is chosen between $50 \leq \mathrm{W}_j \leq 150$. Depending on these former values, the requests r_i of a client node i obey the following law: $r = \frac{|\mathcal{N}|}{|\mathcal{C}|} \times 50 \times \lambda$ and $r \leq r_i \leq 2r$. When λ is small, the tree has a light request load, while large values of λ imply a heavy load on the servers. In the latter case, we expect the problem to have a solution less frequently.

For QoS-aware heuristics, we differentiate two types of trees: 1) trees with a height between 4 and 7, called *small trees*, and 2) trees with a height between 16 and 21, called *big trees*. We study the behavior 1) when QoS constraints[1] are very tight ($\mathrm{q} \in \{1, 2\}$), 2) when QoS constraints are more relaxed (the average value is set to half of the tree height height), and 3) without any QoS constraints ($\mathrm{q} = \mathrm{height} + 1$).

We have computed the number of solutions for each lambda and each heuristic. Each instance of the problem has been solved with the linear program, thus obtaining an optimal solution for *Multiple*.

To assess performances, 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 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 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 without QoS Constraints

Fig. 6b 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 UBCF 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 fewer solutions than these two heuristics, since it is further constrained by the *Upward* policy. As expected, all the

1. Recall that q is the distance from the client to its server(s).

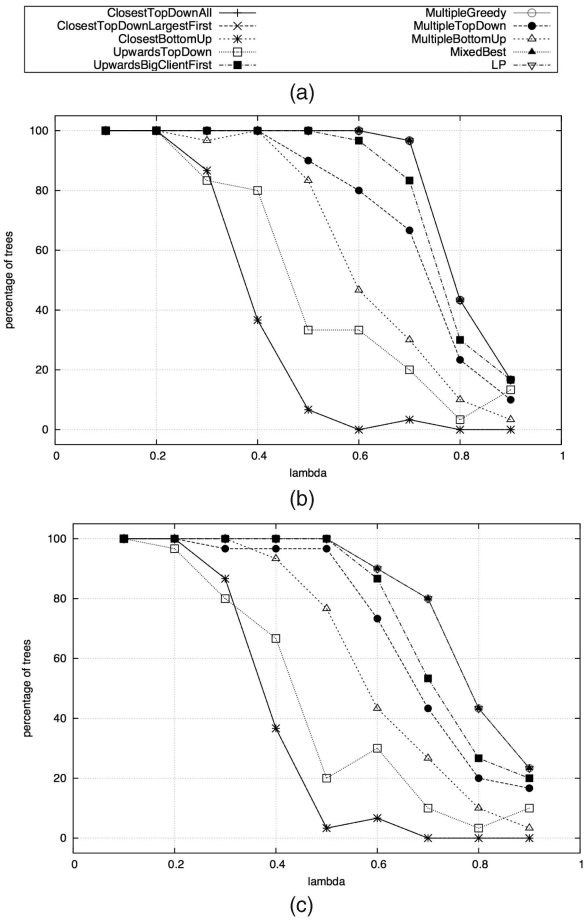

Fig. 6. Percentage of success. (a) Legend. (b) Homogeneous case. (c) Heterogeneous case.

Closest heuristics find a smaller number of solutions as soon as λ reaches higher values: the bottom curve of the plot corresponds to CTDA, CTDLF, and CBU, which find identical 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.

Fig. 7b 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 *Upward*, which in turn is better than *Closest*. For small values of λ, it happens that some *Closest* heuristics give a better solution than those for *Upward* or *Multiple* due to the fact that the latter heuristics are not well optimized for small values of λ. In addition, UBCF is better than all the *Multiple* heuristics for $\lambda = 0.6$. Altogether, the use of the MixedBest heuristic MB allows to always pick up the best result, thereby resulting in a very satisfying relative performance for the *Multiple* instance of the problem. The greedy MG should not be used for small values of λ but proves 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

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

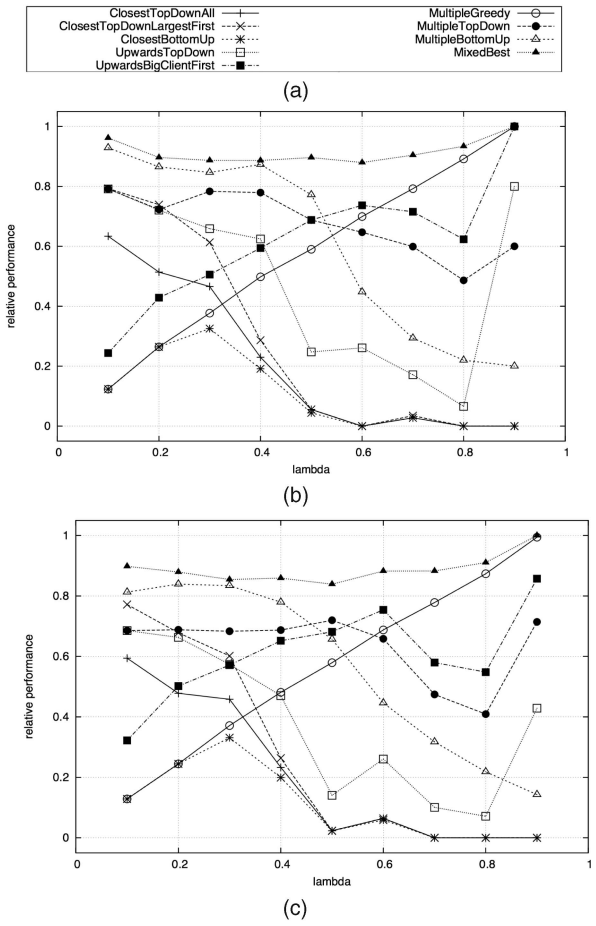

Fig. 7. Relative performance. (a) Legend. (b) Homogeneous case. (c) Heterogeneous case.

performance of at least 85 percent, thus returning a replica cost within 17 percent 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 Figs. 6c and 7c) are very similar to the homogeneous ones, which clearly shows that our heuristics are not much sensitive to the heterogeneity of the platform. Altogether, we have provided an efficient polynomial-time approach to find a satisfactory solution to all the NP-hard problems stated in Section 4.

7.3 Results with QoS Constraints

Due to lack of space, we do not present exhaustive results (they can be found in [6]). Rather, we point out the influence of QoS constraints on the three policies.

When QoS constraints are tight $(\mathrm{q} \in \{1, 2\})$, there is a big gap between the best *Closest* heuristic CBS and the heuristics of the other two policies, particularly when λ is small (see Fig. 8b). For small λ, the *Upward* and *Multiple* policies perform nearly the same, but when $\lambda > 0.4$, *Multiple* outperforms *Upward*. "Outperform" in this case means that there exists a *Multiple* heuristic that has a better relative performance than the best *Upward* heuristic.

When QoS is less tight, i.e., in the case $\mathrm{average_q} = \mathrm{height}/2$, we can observe similar behaviors (see Fig. 8c). *Closest* has still the poorest relative performance, whereas the gap to the other policies is less important. The difference of *Upward* and *Multiple* once again grows with increasing λ. In the

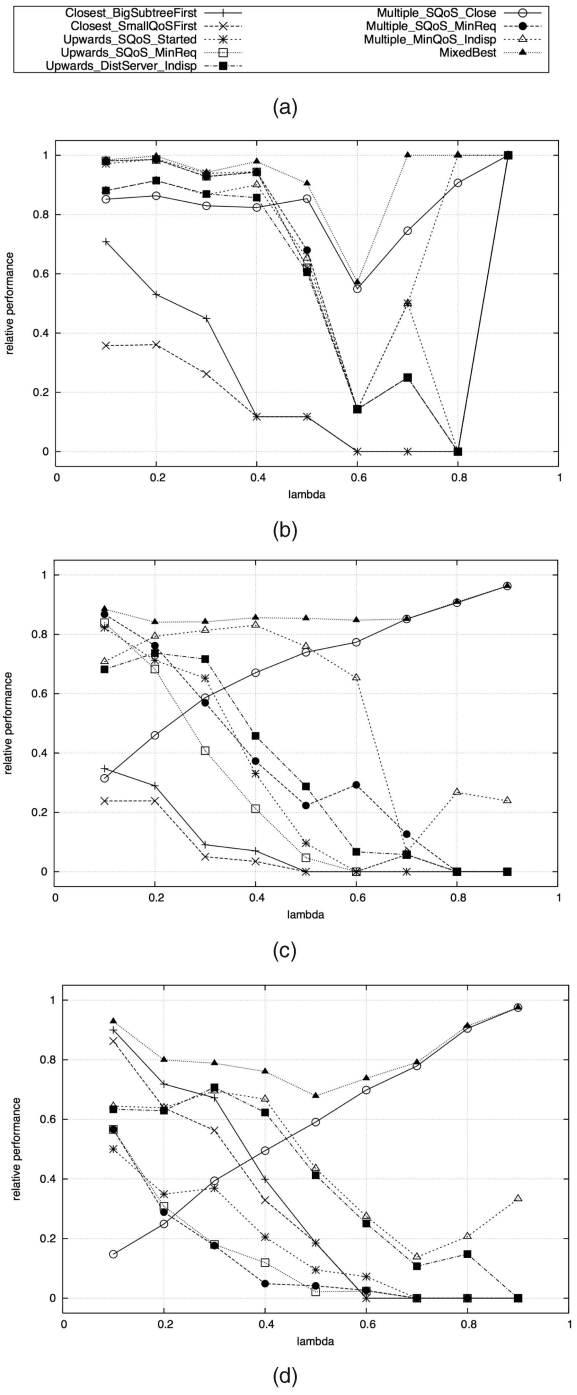

Fig. 8. Relative performance. (a) Legend. (b) Small trees, $\mathrm{q} \in \{1, 2\}$. (c) Big trees, $\mathrm{average_q} = \mathrm{height}/2$. (d) Small trees, $\mathrm{q} = \mathrm{height} + 1 \longrightarrow \mathrm{no\ QoS}$.

configuration with small trees, $\lambda \in \{0.1, 0.2\}$, and no QoS (see Fig. 8d), *Closest* contradicts our expectations, since it shows the best performance. This is an artifact, which can be explained as follows: the heuristics for *Upward* and *Multiple* used in this experiment have been tuned so that their first priority is to take QoS constraints into account, and their counterparts tuned for the case without QoS constraints (see Section 6.1) should be used in this case. Finally, for $\lambda \geq 0.3$, we once again retrieve the previous hierarchical behavior.

Globally, all the results show that QoS constraints do not modify the relative performance of the three policies: with

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

or without QoS, *Multiple* is better than *Upward*, which in turn is better than *Closest*, and their difference in performance is not sensitive to QoS tightness. This is an enjoyable result, which could not be predicted a priori. Altogether, when QoS is tight, we conclude that MSQoSM is the best choice for small values of λ and that MSQoSC is to prefer for larger values. In the case of less tight 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 *Upward* policy, USQoSS behaves the best for tight QoS; in the other cases, UMD achieves better results. Finally, CBS always outperforms CSQoS.

8 Extensions

In this paper, we have considered a simplified instance of the replica problem. We outline here some important generalizations such as dealing with bandwidth constraints and several objects. We also discuss changing the objective function.

8.1 With Bandwidth Constraints and Several Objects

First, we point out that it is not difficult to add link bandwidth constraints in addition to server capacity and QoS constraints. For each communication link, $l \in \mathcal{L}$ such a constraint would write (with the notations of Section 2.2):

$$\sum_{i \in \mathcal{C}, s \in \text{Servers}(i) | l \in \text{path}[i \to s]} r_{i,s} \leq \text{BW}_l.$$

For each of the three policies, such constraints can directly be integrated into the linear program [5]. It is important to point out that the *Closest* policy for the REPLICA COUNTING problem remains of polynomial complexity with both QoS and bandwidth constraints [12].[2]

In this paper, we have restricted the study of the problem to a single object, which means that all replicas are identical (of the same type). We can envision a system in which different types of objects need to be accessed. The clients are then having requests of different types, which can be served only by an appropriate replica. Thus, for an object of type k, client $i \in \mathcal{C}$ issues $r_i^{(k)}$ requests for this object. To serve a request of type k, a node must be provided with a replica of that type. Nodes can be provided with several replica types. A given client is likely to have different servers for different objects. The QoS may also be object-dependent ($\text{q}_i^{(k)}$).

To refine further, new parameters can be introduced such as the size of object k and the computation time involved for this object. Node parameters become object-dependent too, in particular the storage cost and the time required to answer a request. The server capacity constraint must then be a sum on all the object types, while the QoS must be satisfied for each object type. The link capacity also is a sum on the different object types, taking into account the size of each object. There remains to modify the objective function: we simply aim at minimizing the cost of all replicas of different types that have been assigned to the nodes in the solution to get the extended *replica cost* for several objects. Because the constraints add up linearly for different objects, it is not difficult to extend the linear programming formulation of Section 5 to deal with several objects. In addition, the three access policies *Closest*, *Upward*, and *Multiple* could naturally be extended to handle several objects. However, designing efficient heuristics for various object types, especially with different communication to computation ratios and different QoS constraints for each type, is a challenging algorithmic problem.

2. Recall that *Upward* and *Multiple* are already NP-hard with QoS but without bandwidth constraints.

8.2 More Complex Objective Functions

Several important extensions of the problem consist in having a more complex objective function. In fact, either with one or with several objects, we have restricted so far to minimizing the cost of the replicas (and even their number in the homogeneous case). However, several other factors can be introduced in the objective function:

- **Communication cost.** This cost is the *read* cost, i.e., the communication cost required to access the replicas to answer requests. It is thus a sum on all objects and all clients of the communication time required to access the replica. If we take this criteria into account in the objective function, we may prefer a solution in which replicas are close to the clients.
- **Update cost.** The *write* cost is the extra cost due to an update of the replicas. An update must be performed when one of the clients is modifying (writing) some of the data. In this case, to ensure the consistency of the data, we need to propagate the modification to all other replicas of the modified object. Usually, this cost is directly related to the communication costs on the minimum spanning tree of the replica, since the replica that has been modified sends the information to all the other replicas.
- **Linear combination.** A quite general objective function can be obtained by a linear combination of the three different costs, namely, replica cost, read cost, and write cost. Informally, such an objective function would write $\alpha \sum_{\text{servers,objects}} \textit{replica cost} + \beta \sum_{\text{requests}} \textit{read cost} + \gamma \sum_{\text{updates}} \textit{write cost}$, where the application-dependent parameters α, β, and γ would be used to give priorities to the different costs.

Again, designing efficient heuristics for such general objective functions, especially in the context of heterogeneous resources, is a challenging algorithmic problem.

9 Related Work

In the literature, there are two main approaches for the replica placement problem. A first set of papers deals with general graphs and aims in a first step at extracting a "good" spanning tree, i.e., a spanning tree that will optimize some global objective function. In a second step, replicas are placed along the spanning tree, typically in order to optimize a more refined function. However, the process of extracting a spanning tree is of combinatorial nature, as it generalizes the well-known NP-hard K-center problem [13]. Therefore, several authors propose sophisticated heuristics whose goal is to solve both steps simultaneously.

The second set of papers considers that the spanning tree is given [1], [2], [3], [4] and aims at optimizing replica placement on that tree. Because this paper falls into this latter line of research, we refer the reader to the survey paper [14] for more references to the former two-step approach and merely point out that proposed heuristics are

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

of widely different nature, including greedy strategies [15], graph-theoretic algorithms [16], linear-programming techniques [17], [18], and game-related strategies [19].[3]

Early work on replica placement by Wolfson and Milo [20] 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. [2] 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. [4], adding a QoS in terms of a range limit (QoS=distance), and the objective being the REPLICA COUNTING problem. In this latter approach, the servers are homogeneous, and their capacity is bounded.

Cidon et al. [2] and Liu et al. [4] 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. [1]. Indeed, in [1], 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 or QoS constraints, directly reduces to the classical bin-packing problem.

Kalpakis et al. [1] 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 [20]. Different methods can, however, be used such as a minimum cost Steiner tree, in order to further optimize the write strategy [21].

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. [7], [17] 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 [2]. Integrating the communication cost into the objective function can be viewed as a Lagrangian relaxation of QoS constraints.

Tang and Xu [18] 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.

In [3], 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 [18].

Another approach, this time for dynamic content distribution systems, is proposed by Chen et al. [22]. 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.

To the best of our knowledge, there is no related work comparing different access policies, neither on tree networks nor on general graphs. Most previous works impose the *Closest* policy. The *Multiple* policy is enforced by Rodolakis et al. [23] 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. In addition, 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.

10 Conclusion

In this paper, we have introduced and extensively analyzed two important new policies for the replica placement problem. The *Upward* 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*, *Upward*, and *Multiple* policies, both for homogeneous and heterogeneous platforms, and with or without QoS constraints. The polynomial complexity of the *Multiple* policy in the homogeneous case without QoS constraints is quite unexpected, and we have provided an elegant algorithm to compute the optimal cost for this policy. When adding QoS constraints, the same problem becomes NP-complete, which illustrates the additional complexity induced by such constraints. 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*, *Upward*, and *Multiple* policies, and we have compared their performance for several problem instances with or without QoS constraints. In the experiments, the total cost was the sum of the server capacities (or

3. This small list of references is not intended to be comprehensive.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.

their number in the homogeneous case). The impact of the new policies is impressive: the number of trees that admit a solution is much higher with the *Upward* and *Multiple* policies than with the *Closest* policy. Finally, we point out that the absolute performance of the heuristics is quite good, since their cost is close to the optimal solution based upon the solution of the integer linear program.

There remains much work to extend the results of this paper, in several important directions. 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 bandwidth constraints into account. Including bandwidth constraints may require a better global load balancing along the tree, thereby favoring *Multiple* over *Upward*.

In the longer term, designing efficient heuristics for the problem with various object types, all with different communication to computation ratios and different QoS constraints is a demanding algorithmic problem. In addition, 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, especially in the context of heterogeneous resources.

ACKNOWLEDGMENT

The authors thank the reviewers for their numerous comments and suggestions, which greatly improved the final version of the paper.

REFERENCES

[1] 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,* vol. 12, no. 6, pp. 628-637, June 2001.

[2] I. Cidon, S. Kutten, and R. Soffer, "Optimal Allocation of Electronic Content," *Computer Networks,* vol. 40, pp. 205-218, 2002.

[3] H. Wang, P. Liu, and J.-J. Wu, "A QoS-Aware Heuristic Algorithm for Replica Placement," *Proc. Seventh Int'l Conf. Grid Computing (GRID '06),* pp. 96-103, 2006.

[4] P. Liu, Y.-F. Lin, and J.-J. Wu, "Optimal Placement of Replicas in Data Grid Environments with Locality Assurance," *Proc. Int'l Conf. Parallel and Distributed Systems (ICPADS),* 2006.

[5] A. Benoit, V. Rehn, and Y. Robert, "Strategies for Replica Placement in Tree Networks," Research Report 2006-30, LIP, ENS Lyon, Oct. 2006.

[6] A. Benoit, V. Rehn, and Y. Robert, "Impact of QoS on Replica Placement in Tree Networks," Research Report 2006-48, LIP, ENS Lyon, Dec. 2006.

[7] M. Karlsson, C. Karamanolis, and M. Mahalingam, "A Framework for Evaluating Replica Placement Algorithms," Research Report HPL-2002-219, HP Laboratories, 2002.

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

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

[10] *GLPK: GNU Linear Programming Kit,* http://www.gnu.org/software/glpk/, 2008.

[11] *Source Code for the Heuristics,* http://graal.ens-lyon.fr/~vrehn/code/replicaQoS/, 2008.

[12] V. Rehn, "Optimal Closest Policy with QoS and Bandwidth Constraints for Placing Replicas in Tree Networks," Research Report 2007-10, LIP, ENS Lyon, Mar. 2007.

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

[14] T. Loukopoulos, I. Ahmad, and D. Papadias, "An Overview of Data Replication on the Internet," *Proc. Int'l Symp. Parallel Architectures, Algorithms and Networks (ISPAN),* 2002.

[15] C.-M. Wang, C.-C. Hsu, P. Liu, H.-M. Chen, and J.-J. Wu, "Optimizing Server Placement in Hierarchical Grid Environments," *Proc. First Int'l Conf. Grid and Pervasive Computing (GPC '07),* pp. 1-11, 2007.

[16] L. Qiu, V.N. Padmanabhan, and G.M. Voelker, "On the Placement of Web Server Replicas," *Proc. INFOCOM '01,* pp. 1587-1596, 2001.

[17] M. Karlsson and C. Karamanolis, "Choosing Replica Placement Heuristics for Wide-Area Systems," *Proc. 24th Int'l Conf. Distributed Computing Systems (ICDCS '04),* pp. 350-359, 2004.

[18] X. Tang and J. Xu, "QoS-Aware Replica Placement for Content Distribution," *IEEE Trans. Parallel Distributed Systems,* vol. 16, no. 10, pp. 921-932, Oct. 2005.

[19] S.U. Khan and I. Ahmad, "RAMM: A Game Theoretical Replica Allocation and Management Mechanism," *Proc. Int'l Symp. Parallel Architectures, Algorithms and Networks (ISPAN),* 2005.

[20] O. Wolfson and A. Milo, "The Multicast Policy and Its Relationship to Replicated Data Placement," *ACM Trans. Database Systems,* vol. 16, no. 1, pp. 181-205, 1991.

[21] K. Kalpakis, K. Dasgupta, and O. Wolfson, "Steiner-Optimal Data Replication in Tree Networks with Storage Costs," *Proc. Int'l Symp. Database Eng. and Applications (IDEAS '01),* pp. 285-293, 2001.

[22] Y. Chen, R.H. Katz, and J.D. Kubiatowicz, "Dynamic Replica Placement for Scalable Content Delivery," *Proc. First Int'l Workshop Peer-to-Peer Systems (IPTPS '02),* pp. 306-318, Mar. 2002.

[23] G. Rodolakis, S. Siachalou, and L. Georgiadis, "Replicated Server Placement with QoS Constraints," *IEEE Trans. Parallel Distributed Systems,* vol. 17, no. 10, pp. 1151-1162, Oct. 2006.

Anne Benoit received the PhD degree from the Polytechnical Institute of Grenoble (INPG) in 2003. From 2003 to 2005, she was a research associate in the School of Informatics, University of Edinburgh, United Kingdom. She is currently an associate professor at the Laboratoire de l'Informatique du Parallellsme, Ecole Normale Supérieure, Lyon, France. Her research interests include performance evaluation, high-level parallel programming, and algorithms and scheduling for distributed heterogeneous platforms. She is a member of the IEEE and the IEEE Computer Society.

Veronika Rehn-Sonigo is currently working toward the PhD degree at the LIP Laboratory, ENS Lyon. She is mainly interested in parallel algorithms and replica placement, as well as scheduling for distributed platforms. 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 in 1986. He is currently a full professor with the Laboratoire de l'Informatique du Parallellsme, ENS Lyon. His main research interests are scheduling techniques and parallel algorithms for clusters and grids. He served on many editorial boards, including *IEEE Transactions on Parallel AND Distributed Systems (TPDS)*. He was the program chair of HiPC '06 in Bangalore and of IPDPS '08 in Miami. He is a fellow of the IEEE and the IEEE Computer Society. He has been elected a senior member of Institut Universitaire de France in 2007.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 12:25 from IEEE Xplore. Restrictions apply.