Comparison of Access Policies for Replica Placement in Tree Networks

Anne Benoit

LIP Laboratory, ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France
Université de Lyon, UMR 5668 ENS Lyon-CNRS-INRIA-UCBL
`Anne.Benoit@ens-lyon.fr`

LIP Research Report RR-2009-12

Abstract. In this paper, we discuss and compare several policies to place replicas in tree networks subject to server capacity. 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 a single server in the tree (*Single*). One major contribution of this paper is to assess the impact of a new policy in which requests of a given client can be processed by multiple servers (*Multiple*), thus distributing the processing of requests over the platform. We characterize problem instances for which *Multiple* cannot be more than two times better than the optimal *Single* solution, if this latter exists. For such instances, we provide a procedure which builds a *Single* solution with a guarantee on its cost. This is a very interesting result for applications which do not accept multiple servers for a given client, since it might be more difficult to implement such a complex strategy.

Keywords. replica placement, access policies, heterogeneous platforms, hierarchical platforms, VOD applications.

1 Introduction

In this paper, we discuss and compare several policies to place replicas in tree networks subject to server capacity. The client requests are known beforehand, 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.

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

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 nodes are identical (homogeneous version), 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 standard approach in the literature is to enforce that all requests of a client be served by a single server in the tree; this policy is called *Single*. Following the hierarchical structure of the platform, the server must be on the path from the client to the root of the tree. We introduced in [3] a new policy, *Multiple*, in which the requests of a given client can be processed by multiple servers on the path, thus distributing the processing of requests over the platform. All problems were shown to be NP-complete, except the multiple/homogeneous combination, in which the optimal solution can be found in polynomial time, using a multi-pass greedy algorithm.

One major contribution of this paper is to assess the impact of the new *Multiple* policy on the total replication cost, and the impact of server heterogeneity, both from a theoretical and a practical perspective. Thus we demonstrate the usefulness of the new policy, even in the case of identical servers. The first result is that multiple allows to find solutions when the classical single policy does not. More generally, a single server policy will never have a solution if one client sends more requests than the largest server capacity. Then we build an instance of the problem where both access policies have a solution, but the solution of *Multiple* is arbitrarily better than the solution of *Single*. This is quite evident for heterogeneous platforms [3], but it is a new result for the homogeneous case.

If we focus on homogeneous platforms, there are many problem instances for which *Multiple* cannot be more than two times better than the optimal *Single* solution, if this latter exists. In our work, we thoroughly characterize such cases and we provide a procedure which builds a *Single* allocation with a guarantee on the cost. The idea consists in having a single server allocation in which each server is processing a number of requests being at least equal to half of its processing capacity. The servers may be fully used in the *Multiple* allocation, thus the solution will be up to two times better, but not arbitrarily better. This is a very interesting result for applications which do not accept multiple servers for a given client, since it might be more difficult to implement such a complex strategy.

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 compare the different access policies. Next in Section 4 we proceed to the complexity results, both in the homogeneous and heterogeneous cases. Section 5 introduces the procedure to build efficient *Single* solutions. Section 6 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 7.

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 the simplest instance of the problem. Then we describe several types of constraints that can be added to the formulation.

2.1 Definitions and notations

We consider a distribution tree whose nodes are partitioned into a set of clients $\mathcal{C}$ and a set of nodes $\mathcal{N}$. 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. Each client $i \in \mathcal{C}$ (leaf of the tree) is sending r_i requests per time unit to a database object. A node $j \in \mathcal{N}$ (internal node of the tree) may or may not have been provided with a replica of this 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. Node j 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. It is quite natural to assume that sc_j is proportional to W_j: the more powerful a server, the more costly.

Let r be the root of the tree. If $j \in \mathcal{N}$, then $\mathsf{children}(j) \subseteq \mathcal{N} \cup \mathcal{C}$ is the set of children of node j. If $k \neq r$ is any node in the tree (leaf or internal), $\mathsf{parent}(k) \in \mathcal{N}$ is its parent in the tree. Let $\mathsf{Ancestors}(k) \subseteq \mathcal{N}$ 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). Finally, $\mathsf{subtree}(k) \subseteq \mathcal{N} \cup \mathcal{C}$ is the subtree rooted in k, including k.

2.2 Problem instances

There are two scenarios for the number of servers assigned to each client:

Single server – Each client i is assigned a single server that is responsible for processing all its requests.

Multiple servers– A client i may be assigned several servers in a set $\mathsf{Servers}(i)$.

To the best of our knowledge, the single server policy (*Single*) 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.

For each client $i \in \mathcal{C}$, let $\mathsf{Servers}(i) \subseteq \mathsf{Ancestors}(i)$ be the set of servers responsible for processing at least one of its requests. In a single server access policy, this set is reduced to only one server: $\forall i \in \mathcal{C} \ \ |\mathsf{Servers}(i)| = 1$. Let R be the set of replicas: $R = \{s \in \mathcal{N} \mid \exists i \in C\ ,\ s \in \mathsf{Servers}(i)\}$. Also, we let $r_{i,s}$ be the number of requests from client i processed by server s. All requests must be processed, thus $\sum_{s \in \mathsf{Servers}(i)} r_{i,s} = r_i$. In the single server case, this means that a unique server is handling all r_i requests. The problem is constrained by the server capacities: no server capacity can be exceeded, thus $\forall s \in R,\ \sum_{i \in \mathcal{C} | s \in \mathsf{Servers}(i)} r_{i,s} \leq W_s$. Finally, the objective function is defined as $\min \sum_{s \in R} \mathsf{sc}_s$.

In addition to the two access policies *Single* or *Multiple*, we consider different platform types, with different or identical servers:

Different servers – As already pointed out, it is frequently assumed that the cost of a server is proportional to its capacity: $\mathsf{sc}_s = W_s$. The problem thus reduces to finding a valid solution of minimal cost, where "valid" means that no server capacity is exceeded. We name Replica Cost this problem.

Identical servers – We can further simplify the previous problem in the homogeneous case: with identical servers ($\forall s \in \mathcal{N},\ W_s = W$), 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.

2.3 Note about the *Closest* policy

In the literature, the *Single* strategy is further constrained to the *Closest* policy: 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 s. Then any other client i' residing in $\mathsf{subtree}(s)$ will be assigned a server in that subtree. This forbids requests from i' to "traverse" s and be served higher (closer to the root in the tree).

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

3 Access policies comparison

In this section, we compare the general *Single* policy with the *Multiple* one for the Replica Counting problem. Section 3.1 illustrates the impact of policies on the existence of a solution. Then Section 3.2 shows that *Multiple* can be arbitrarily better than *Single*. Finally, Section 3.3 presents lower bounds and show some cases in which the optimal solution is arbitrarily higher than this bound. This comparison is done on the Replica Counting problem, thus the results can be generalized to the Replica Cost problem.

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, B being the unique child of A, the tree root (see Fig. 1). Each node can process $W = 1$ request.

If B has two client children, each making 1 request, the solution consists in placing a replica on each node, and both clients car be served (Fig. 1(a)). This works for the *Single* policy (and thus for the *Multiple* one). However, if B 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 (Fig. 1(b)).

This example demonstrates the usefulness of the new policy: the *Multiple* policy allows to find solutions when the classical *Single* policy does not. More generally, a single server policy will never have a solution if one client sends more requests than the largest server capacity.

3.2 Impact of the access policy on the cost of a solution

In this section we build an instance of the Replica Counting problem where both access policies have a solution, but the solution of *Multiple* is arbitrarily better than the solution of *Single*.

Consider the instance of Replica Counting represented in Fig. 2, with $3 + n$ nodes of capacity $W = 4n$. The root A has $n + 2$ children nodes B, C and

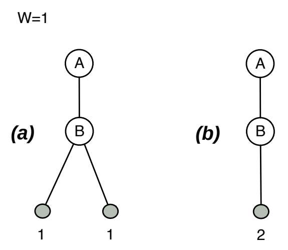

Fig. 1. Solution existence.

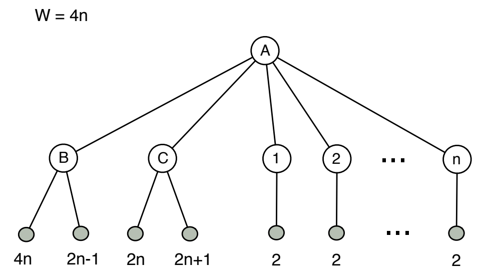

Fig. 2. Solution cost.

$1, ..., n$. Node B has two client children, one with $4n$ requests and the other with $2n-1$ requests. Node C has two client children, one with $2n$ requests and the other with $2n+1$ requests. Each node numbered i has a unique child, a client with 2 requests.

The *Multiple* policy assigns 3 replicas to A, B and C. B handles the $4n$ requests of its first client, while the other client is served by A. C handles $2n$ requests from both of its clients, and the 1 remaining request is processed by A. Server A therefore processes $(2n-1)+1 = 2n$ requests coming up from B and C. Requests coming from the n remaining nodes sum up to $2n$, thus A is able to process all of them.

For the *Single* policy, we need to assign replicas everywhere. Indeed, with this policy, C cannot handle more than $2n+1$ requests since it is unable to process requests from both of its children, and thus A has $(2n-1)+2n$ requests coming from B and C. It cannot handle any of the $2n$ remaining requests, and thus each remaining node must process requests coming from its own client. This leads to a total of $n+3$ replicas.

The performance factor is thus $\frac{n+3}{3}$, which can be arbitrarily big when n becomes large.

3.3 Lower bound

Obviously, the cost of an optimal solution of the Replica Counting problem (for any policy) cannot be lower than the 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 example of Fig. 3 shows that the solution can be arbitrarily higher than this lower bound, even for the *Multiple* policy, since many servers are required to be placed and each of them is only handling a small portion of work.

Consider Fig. 3, with $n+2$ nodes of capacity $W = n$. The root of the tree A has $n+1$ children: B and $1, ..., n$. Node B has two client children, each sending n requests. Each other node has a unique child, a client with only one request. The

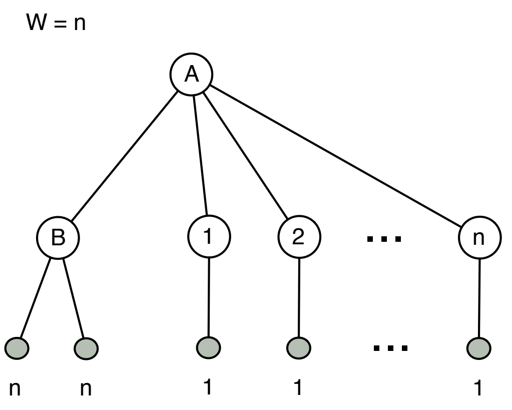

Fig. 3. The lower bound cannot be approximated for Replica Counting.

lower bound is $\left\lceil \frac{\sum_{i \in \mathcal{C}} r_i}{W} \right\rceil = \frac{3n}{n} = 3$. However, both policies will assign replicas to A and B to cover both clients of B, and will then need n extra replicas, one per node $1, ..., n$. The total cost is thus $n+2$ replicas, arbitrarily higher than the lower bound.

Previous examples give an insight of the combinatorial nature of the replica placement optimization problems, even in its simplest variant with identical servers, Replica Counting. The following section corroborates this insight: most problems are shown NP-hard, even though some variants have polynomial complexity.

4 Complexity results

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?

	Single	***Multiple***
Replica Counting	NP-complete	polynomial
Replica Cost	NP-complete	NP-complete

Table 1. Complexity results.

Table 1 captures the complexity results. These complexity results are proved in [2, 1]. The NP-completeness of the *Single*/Replica Counting case comes as a surprise, since all previously known instances with the extra *Closest* constraint were shown to be polynomial, using dynamic programming algorithms. With different servers (Replica Cost), the problem is combinatorial due to resource heterogeneity. The only polynomial case is the *Multiple*/Replica Counting combination, in which the optimal solution can be found using a multi-pass greedy algorithm described in [3].

5 From *Multiple* to *Single*

In this section, we give a procedure to build a *Single* allocation for the Replica Counting problem, with a guarantee on the cost. Indeed, while an optimal solution for the *Multiple* problem can be found in polynomial time, the *Single* problem is NP-complete. Still, some applications may require a *Single* policy since the implementation is much more straightforward for such a policy. Our goal is thus to build a good *Single* allocation, compared to a lower bound given by the *Multiple* solution.

Recall that in the worst case, the optimal *Single* can be arbitrarily worse than the optimal *Multiple* (see Section 3.2). Thus we aim at characterizing problem instances for which a good *Single* solution can be derived.

5.1 Problem formulation

Let $(\mathcal{C}, \mathcal{N})$ be a problem instance in which $r_i \leq W$ for all $i \in \mathcal{C}$ (otherwise, there is no solution to the *Single* problem). We are given an optimal *Multiple* solution for this problem, of cost M (i.e., M is the number of servers in this solution). We aim at finding a *Single* solution with a cost $S \leq 2M$, and at characterizing cases in which this is possible.

5.2 Linear trees

First let us consider a linear tree consisting in n nodes, ordered as a linear chain $1 \to 2 \to \ldots \to n$, and a set $\mathcal{C}_j$ of clients attached to node j of the chain, for $1 \leq j \leq n$ (note that $\mathcal{C} = \cup_{1 \leq j \leq n} \mathcal{C}_j$). Furthermore, we assume that $jW \geq 2 \sum_{i \in \cup_{1 \leq k \leq j} \mathcal{C}_k} r_i$, meaning that, at each level of the chain, there are twice more nodes than the minimum number of servers requested to handle all requests from the root down to this level. On the whole chain, we have a condition on the total number of nodes, $n \geq \frac{2}{W} \sum_{i \in \mathcal{C}} r_i$.

We build a solution to *Single* by assigning requests greedily, starting from the clients at the top of the tree, since they have less choice of nodes where to be processed. At each step, we try to give requests to servers that are already processing some requests, and we try to allocate requests starting from the bottom of the chain. If it fails, a new server is created on the first free node. The **linear-tree** procedure returns the *Single* solution in which $s_{i,j} = r_i$ if node j is processing requests of client i, and s_j is the total number of requests processed by node j, $s_j = \sum_{i \in \mathcal{C}} s_{i,j}$.

```
procedure linear-tree ($\mathcal{C},\mathcal{N}$)
$\forall i \in \mathcal{C}, \forall j \in \mathcal{N}, s_{i,j}=0$; $\forall j \in \mathcal{N}, s_j = 0$;      // Initialisation.
for $j = 1..n$ do
    for $i \in \mathcal{C}_j$ do
        // Try to add requests to an existing server.
        for $j' = j..1$ do
            if $\sum_{k\in\mathcal{N}} s_{i,k} = 0$ and $s_{j'} \neq 0$ then
                if $r_i + s_{j'} \leq W$ then $s_{i,j'} = r_i$; $s_{j'} = s_{j'} + r_i$;
            end
        end
        // If the previous loop did not succeed, create a new server.
        if $\sum_{k\in\mathcal{N}} s_{i,k} = 0$ then
            for $j' = j..1$ do
                if $\sum_{k\in\mathcal{N}} s_{i,k} = 0$ and $s_{j'} = 0$ then  $s_{i,j'} = r_i$; $s_{j'} = r_i$;
            end
        end
    end
end
```

The procedure never fails because of the assumption on the number of nodes. Indeed, let us count the number of servers allocated by the procedure, S_j, at each step of the loop on j. Each couple of servers (k, k') is such that $s_k + s_{k'} > W$, otherwise the greedy allocation would have assigned all requests to one of these servers. Thus, all servers but eventually the last one are handling at least $W/2$ requests. If $S_j \geq 2$, we associate the last server, k (with possibly less than $W/2$ requests) with another one, k', thus we have $s_k + s_{k'} > W$. Then it is easy to see that the total number of requests handled by the solution at this level j is greater than $(S_j - 2)\frac{W}{2} + W = S_j\frac{W}{2}$. Moreover, we know that the total number of requests at level j is $\sum_{i\in\cup_{1\leq k\leq j}\mathcal{C}_k} r_i$, and thus $S_j < \frac{2}{W}\sum_{i\in\cup_{1\leq k\leq j}\mathcal{C}_k} r_i \leq j$ by hypothesis. If $S_j = 1$, then $S_j \leq j$ since $j \geq 1$. Therefore, at each level, there are enough nodes available so that we are able to perform the allocation (j nodes available, and S_j servers needed). From the upper bound on the total number of servers S, we can immediately derive the upper bound on the cost, since M must be at least equal to $\left\lceil \frac{\sum_{i\in\mathcal{C}} r_i}{W} \right\rceil$ in order to handle all requests. We have $S = S_n \leq 2\frac{\sum_{i\in\mathcal{C}} r_i}{W}$ from the previous reasoning, and thus $S \leq 2M$.

5.3 General trees

The problem becomes more complex in the case of general trees, because several branches of the tree are interacting. However, the transformation procedure can still be performed, dealing successively with each branch of the tree and enforcing a condition on the minimum number of nodes on each branch.

We start by applying the **linear-tree** procedure successively on each branch of the tree. The number of servers is not guaranteed anymore, thus the procedure may fail. If at the end of the procedure on a branch, some clients have not

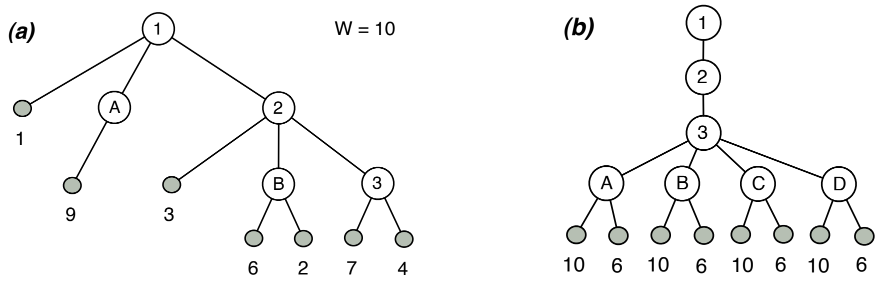

Fig. 4. Dealing with a general tree

been assigned to any server, let j be the server which corresponds to the first branching. We consider all requests coming from other branches already assigned to servers j up to the root server, and if possible we either create a new server in the corresponding branch, or move these requests down into this branch. The example of Fig. 4(a) illustrates this procedure. Assume that the two leftmost branches have already been processed, with server 1 handling clients 1 and 9, server 2 handling clients 3 and 6, and server B handling client 2. It is not possible to process requests of clients 7 and 4, thus we move down other requests, either client 9 onto server A (server creation), or client 6 onto server B.

However, it might fail even if the constraint on the minimum number of servers at each level of each branch is respected, as for instance in the example of Fig. 4(b). In this case, each server A,B,C and D will be in charge of the corresponding client with 10 requests, and other requests cannot be assigned to servers 1, 2 and 3 with a *Single* policy.

Thus we add an extra constraint on the minimum number of nodes in a subtree, in order to guarantee the construction of the *Single* solution. Of course, note that the procedure can be followed even if the constraints are violated, but in such cases it might fail. The idea consists in adding a constraint similar to the linear tree one, but generalized on subtrees to allow correct branching. For each node $j \in \mathcal{N}$ such that $|\mathsf{children}(j) \cap \mathcal{N}| \geq 2$, we request that $\mathsf{subtree}(j)$ follows the linear tree property. Thus all clients of a branch rooted in j will possibly be processed by servers in $\mathsf{subtree}(j)$, and it will always be possible to process requests when dealing with a new branch.

Formally, the constraint can be written as follows (recall that r is the root of the tree, and that the property also is true for all linear chains starting from the root – case $j = r$):

$$\begin{aligned}\forall j \in \{j' \in \mathcal{N} \mid |\mathsf{children}(j') \cap \mathcal{N}| \geq 2\} \cup \{r\},\ \forall k \in \mathsf{subtree}(j) \cap \mathcal{N}, \\ \text{let } X = \{j\} \cup \mathsf{Ancestors}(k) \cap \mathsf{subtree}(j). \\ \text{Then } |X| \geq \frac{2}{W} \sum_{\ell \in X} \sum_{i \in \mathsf{children}(\ell) \cap \mathcal{C}} r_i \qquad (1)\end{aligned}$$

The algorithm is then the following:

1. For each node $j \in \mathcal{N}$, let $br(j) = |\mathsf{children}(j) \cap \mathcal{N}|$ be the number of branches rooted in j and not yet processed.
2. Call the **linear-tree** procedure on the leftmost branch of the tree, $r = 1 \rightarrow 2 \rightarrow \ldots \rightarrow k$. We denote the current branch by $cb = (1, 2, ..., k) \subseteq \mathcal{N}$, and we mark this branch as processed: $\forall \ell \in cb,\ br(\ell) = br(\ell) - 1$.
3. For $j = \max_{j' \in cb}\{j' \mid br(j') \geq 1\}$ (first branching from the bottom of the tree), call the **linear-tree** procedure on the leftmost branch rooted in j and not yet processed, denoted as $j \rightarrow j_1 \rightarrow \ldots \rightarrow j_k$, with $j < j_1 < \ldots < j_k$. We set $cb = (j, j_1, ..., j_k)$ and mark it as processed: $\forall \ell \in cb,\ br(\ell) = br(\ell) - 1$.
4. If required, call the **merge-servers** procedure on the current branch.
5. Go back to step 3, until there is no more branching not yet processed (i.e., $\forall j \in \mathcal{N},\ br(j) \leq 0$).

We prove in the following that the algorithm returns a *Single* solution with a cost $S \leq 2M$, with in some cases the extra constraint that the tree is binary. At the end of step 2, there is at most one server which is handling less than $W/2$ requests in the branch. The allocation always is possible because of constraint (1) applied on r. For step 3, let us check that there exists a *Single* solution. The number of requests of clients attached to node j_1 is at most W because of constraint (1), thus it is always possible to create a server at this node and handle these requests. Similarly, there is no problem when assigning requests from clients attached to nodes j_2 to j_k. However, at the end of this call, we might have two servers handling less than $W/2$ requests, one in the subtree already processed, x, and one in the new branch, y. In this case, we call the **merge-servers** procedure which aims at suppressing one of these servers. If $x \in \mathsf{Ancestors}(j)$ (recall that j is the root of the current branch), then we can simply move requests processed by y to server x, and there remains at most one server with less than $W/2$ requests. Similarly, if one node in $\mathsf{Ancestors}(j)$ is not yet a server, we can move requests processed by x and y onto this node, thus merging both servers into a single one. Otherwise, because of constraint (1) at node j, it is always possible to process requests of the current branch without using server j. However, we need at this point an additional constraint on the subtree rooted in j in order to have the guarantee: it should be a binary tree. With this extra constraint, we can process all requests of both subtrees without using server j, and we can move requests processed by x to server j, since the requests coming from clients attached to j are less than $W/2$ (constraint (1)). Then, if there are still two servers with less than $W/2$ requests, these servers are y and j, and it is possible to move requests from y to j.

Without the binary tree constraint, it might not be possible to merge servers. For instance, consider a tree whose root r has 4 children nodes, 1..4. Nodes 1..4 each has one single client with 4 requests, and $W = 10$. Then two clients will be processed by r, but there will remain two servers each processing only $4 < W/2$ requests, with no possibility of merging. Note however that the solution still has the performance guarantee $S \leq 2M$, and even in this case $S = M$ since 3 servers are required for both policies.

Therefore, for a binary tree respecting constraint (1), we can build a *Single* solution with a cost $S \leq 2M$. These constraints can however be relaxed and we expect the procedure to return efficient *Single* solutions in most cases anyway.

6 Related work

Early work on replica placement by Wolfson and Milo [10] 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, and thus they address the case of special topologies, and in particular tree networks. They give a polynomial solution in a fully homogeneous case, using the *Single* closest server access policy. More recent works [4, 8] use the same access policy, and 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. All papers listed above consider the *Single* closest access policy. As already stated, most problems are NP-complete, except for some very simplified instances. Karlsson et al [7, 6] compare different objective functions and several heuristics to solve these complex problems.

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 [9] 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.

7 Conclusion

In this work, we have carefully analyzed different strategies for replica placement, and proved that a *Multiple* solution may be arbitrarily better than a *Single* one, even on homogeneous platforms. Moreover, we have provided an algorithm to build a *Single* solution, which is guaranteed to use no more than two times more servers than the optimal *Multiple* solution, given some constraints on the problem instance. This is a very interesting result, given that the *Single* problem on homogeneous platforms is NP-difficult, and that some applications may not support multiple allocations.

Even though the constraints on the trees are quite restrictive, the procedure can be applied on any tree and still return good *Single* solutions, even if the application tree does not allow for a guarantee on the solution. We expect that the ratio of 2 should be achievable in most practical situations. It would be very interesting to simulate the procedure on random application trees and practical ones, in order to figure out the percentage of success and the average performance ratio lost by moving from a *Multiple* solution to a *Single* one. We plan to explore such directions in future work.

References

1. A. Benoit, V. Rehn, and Y. Robert. Impact of QoS on Replica Placement in Tree Networks. Research Report 2006-48, LIP, ENS Lyon, France, Dec. 2006. Available at `graal.ens-lyon.fr/~abenoit/`. Short version appears in ICCS'2007, the 7th International Conference on Computational Science.
2. 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/~abenoit/`. Short version appears in HCW'2007, the 16th IEEE Heterogeneity in Computing Workshop.
3. A. Benoit, V. Rehn-Sonigo, and Y. Robert. Replica Placement and Access Policies in Tree Networks. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1614–1627, 2008.
4. I. Cidon, S. Kutten, and R. Soffer. Optimal allocation of electronic content. *Computer Networks*, 40:205–218, 2002.
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. 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.
7. M. Karlsson, C. Karamanolis, and M. Mahalingam. A framework for evaluating replica placement algorithms. Research Report HPL-2002-219, HP Laboratories, Palo Alto, CA, 2002.
8. P. Liu, Y.-F. Lin, and J.-J. Wu. Optimal placement of replicas in data grid environments with locality assurance. In *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society Press, 2006.
9. G. Rodolakis, S. Siachalou, and L. Georgiadis. Replicated server placement with QoS constraints. *IEEE Trans. Par. Distr. Systems*, 17(10):1151–1162, 2006.
10. O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.*, 16(1):181–205, 1991.