Power-aware replica placement and update strategies in tree networks

Anne Benoit, Paul Renaud-Goud, yves Robert

Power-aware replica placement and update strategies in tree networks

Anne Benoit, Paul Renaud-Goud, Yves Robert
LIP, Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Email: {Anne.Benoit|Paul.Renaud-Goud|Yves.Robert}@ens-lyon.fr

October 1, 2010

Abstract

This paper deals with optimal strategies to place replicas in tree networks, with the double objective to minimize the total cost of the servers, and/or to optimize power consumption. The client requests are known beforehand, and some servers are assumed to pre-exist in the tree. Without power consumption constraints, the total cost is an arbitrary function of the number of existing servers that are reused, and of the number of new servers. Whenever creating and operating a new server has higher cost than reusing an existing one (which is a very natural assumption), cost optimal strategies have to trade-off between reusing resources and load-balancing requests on new servers. We provide an optimal dynamic programming algorithm that returns the optimal cost, thereby extending known results without pre-existing servers. With power consumption constraints, we assume that servers operate under a set of M different modes depending upon the number of requests that they have to process. In practice M is a small number, typically 2 or 3, depending upon the number of allowed voltages. Power consumption includes a static part, proportional to the total number of servers, and a dynamic part, proportional to a constant exponent of the server mode, which depends upon the model for power. The cost function becomes a more complicated function that takes into account reuse and creation as before, but also upgrading or downgrading an existing server from one mode to another. We show that with an arbitrary number of modes, the power minimization problem is NP-complete, even without cost constraint, and without static power. Still, we provide an optimal dynamic programming algorithm that returns the minimal power, given a threshold value on the total cost; it has exponential complexity in the number of modes M, and its practical usefulness is limited to small values of M. Still, experiments conducted with this algorithm show that it can process large trees in reasonable time, despite its worst-case complexity.

Key words: Replica placement, tree networks, power consumption, update strategies, complexity results, dynamic programming algorithms.

1 Introduction

We revisit the well-known replica placement problem in tree networks [6, 19, 2], with two new objectives: reusing pre-existing replicas, and enforcing an efficient power management. In a nutshell, the replica placement problem is the following: we are given a tree-shaped network where clients are periodically 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 internal node. Note that the distribution tree (clients and nodes) is fixed in the approach. This key assumption is quite natural for a broad spectrum of applications, such as electronic, ISP, or VOD service delivery (see [10, 6, 12] and additional references in [19]). 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.

In the original problem, there is no replica before execution; 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, serve all the clients located in their subtree (so that the root, if equipped with a replica, can serve any client). The rule of the game is to assign replicas to nodes so that the total number of replicas is minimized. This problem is well-understood: it can be solved in time $O(N^2)$ (dynamic programming algorithm of [6]), or even in time $O(N \log N)$ (optimized greedy algorithm of [19]).

The first contribution of this paper is to tackle the replica placement problem when the tree is equipped with pre-existing replicas before execution. This extension is a first step towards dealing with dynamic replica management: if the number and location of client requests evolve over time, the number and location of replicas must evolve accordingly, and one must decide how to perform a configuration change (at what cost?) and when (how frequently reconfigurations should occur?) A first approach to this complicated dynamic problem is provided in [18], where replicas are either moved or created at "regular intervals", whose duration is determined by the arrival rate of client requests. The algorithms in [18] provide a heuristic solution to the problem, but no complexity result is presented. Similarly, [5, 15, 16] tackle the problem of placing replicas with server capacity constraint, where servers are re-allocated to new sites when a performance metric degrades significantly. However, in these papers, the distribution tree is not fixed, which renders all problems highly combinatorial, and which departs from our fixed network assumption. In the present work, the aim is to assess the difficulty of a single reconfiguration, and we provide an optimal polynomial algorithm to minimize the cost of such a reconfiguration. The main difficulty here is to trade-off between two conflicting goals, namely (i) reusing existing servers rather than creating new ones, and (ii) load-balancing the requests equally among the servers.

Another contribution of this paper is to extend replica placement algorithms to cope with power consumption constraints. Minimizing the total power consumed by the servers has recently become a very important objective, both for economic and environmental reasons [13]. To help reduce power dissipation, multi-modal processors are used: each processor has a discrete number of predefined speeds (or modes), which correspond to different voltages that the processor can be subjected to. The power consumption is the sum of a static part (the cost for a processor to be turned on) and a dynamic part. This dynamic part is a strictly convex function of the processor speed, so that the execution of a given amount of work costs more power if a processor runs in a higher mode [8]. More precisely, a processor operated at mode W_i dissipates W_i^α watts, where $\alpha \in [2..3]$ is some constant specified by the model [9, 14, 3, 1, 4]. Faster modes allow servers to handle more requests per time-units, while they also lead to a much

higher (supra-linear) power consumption. An important result of this paper is that minimizing power consumption is a NP-complete problem, independently of the incurred cost (in terms of new and pre-existing servers) of the solution. In fact this result holds true even without pre-existing replicas, and without static power: balancing server modes across the tree already is a hard combinatorial problem.

The cost of the best power-efficient solution may indeed be prohibitive, which calls for a bi-criteria approach: minimizing power consumption while enforcing a threshold cost that cannot be exceeded. We investigate the case where there is only a fixed number of modes and show that there are polynomial-time algorithms capable of optimizing power for a bounded cost, even with pre-existing replicas, with static power and with a complex cost function. This result has a great practical significance, because state-of-the-art processors can only be operated with a restricted number of voltage levels, hence with a few modes [11, 8].

Next we run simulations to show the practical utility of our algorithms, despite their high worst-case complexity. We illustrate the impact of taking pre-existing servers into account, and how power can be saved thanks to the optimal bi-criteria algorithm.

The rest of the paper is organized as follows. Section 2 is devoted to a detailed presentation of the target optimization problems, and provides a summary of new complexity results. The next two sections are devoted to the proofs of these results: Section 3 deals with computing the optimal cost of a solution, with pre-existing replicas in the tree, while Section 4 addresses all power-oriented problems. Then we report the simulation results in Section 5. Finally, we state some concluding remarks and future working directions in Section 6.

2 Framework

This section is devoted to a precise statement of the problem. We start with the general problem without power consumption constraints, and next we introduce the power consumption model. Then we state the objective functions (with or without power), and the associated optimization problems. Finally we give a summary of all complexity results that we provide in the paper.

2.1 Replica servers

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. Each client $i \in \mathcal{C}$ (leaf of the tree) is sending r_i requests per time unit to a database object. Internal nodes equipped with a replica (also called *servers*) will process all requests from clients in their subtree. An internal node $j \in \mathcal{N}$ may have already been provided with a replica, and we let $\mathcal{E} \subseteq \mathcal{N}$ be the set of pre-existing servers. Servers in $\mathcal{E}$ will be either reused or deleted in the solution. Note that 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.

Without power consumption constraints, the problem is to find a *solution*, i.e., a set of servers capable of handling all requests, that minimizes some cost function. We formally define a valid solution before detailing its cost. We start with some notations. 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, and $\mathsf{subtree}_j \subseteq \mathcal{N} \cup \mathcal{C}$ is the subtree rooted in j, excluding j. A solution is a set $\mathcal{R} \subseteq \mathcal{N}$ of servers. Each client i is assigned a single server $\mathsf{server}_i \in \mathcal{R}$ that is responsible for processing all its r_i requests, and this server is restricted to be the first ancestor of i (i.e., the first node in the unique path that leads from i up to the root r) equipped with a server (hence the name *closest* for the request service policy). Such a server must exist in $\mathcal{R}$ for each client. In addition, all servers are identical and

have a limited capacity, i.e., they can process a maximum number W of requests. Let req_j be the number of requests processed by $j \in \mathcal{R}$. The capacity constraint writes

$$\forall j \in \mathcal{R}, \ \ \mathsf{req}_j = \sum_{i \in \mathcal{C} \mid j = \mathsf{server}_i} r_i \leq W. \tag{1}$$

Now for the cost function, because all servers are identical, the cost of operating a server can be normalized to 1. When introducing a new server, there is an additional cost create, so that running a new server costs $1 + \mathsf{create}$ while reusing a server in $\mathcal{E}$ only costs 1. There is also a deletion cost delete associated to deleting each server in $\mathcal{E}$ that is not reused in the solution. Let $E = |\mathcal{E}|$ be the number of pre-existing servers. Let $R = |\mathcal{R}|$ be the total number of servers in the solution, and $e = |\mathcal{R} \cap \mathcal{E}|$ be the number of reused servers. Altogether, the cost is

$$\mathsf{cost}(\mathcal{R}) = R + (R - e) \times \mathsf{create} + (E - e) \times \mathsf{delete}. \tag{2}$$

This cost function is quite general. Because of the create and delete costs, priority is always given to reusing pre-existing servers. If $\mathsf{create} + 2 \times \mathsf{delete} < 1$, priority is given to minimizing the total number of servers R: indeed, if this condition holds, it is always advantageous to replace two pre-existing servers by a new one (if capacities permit).

2.2 With power consumption modes

With power consumption constraints, we assume that servers may operate under a set $\mathcal{M} = \{W_1, \ldots, W_M\}$ of different speeds, or *modes*, depending upon the number of requests that they have to process per time unit. Here modes are indexed according to increasing values, and $W_M = W$, the maximal capacity. If a server $j \in \mathcal{R}$ processes req_j requests, with $W_{i-1} < \mathsf{req}_j \leq W_i$, then it is operated at mode W_i, and we let $\mathsf{mode}(j) = i$. The power consumption of a server $j \in \mathcal{R}$ obeys the classical model

$$\mathcal{P}(j) = \mathcal{P}^{(\mathsf{static})} + W^{\alpha}_{\mathsf{mode}(j)}.$$

Here, $\mathcal{P}^{(\mathsf{static})}$ is the static power consumption (constant part), while $W^{\alpha}_{\mathsf{mode}(j)}$ is the dynamic part that depends upon the operated mode. Finally, $\alpha \in [2..3]$ is a rational constant that depends upon the model for power [9, 14, 3, 1, 4]. The total power consumption $\mathcal{P}(\mathcal{R})$ of the solution is the sum of the power consumption of all server nodes:

$$\mathcal{P}(\mathcal{R}) = \sum_{j \in \mathcal{R}} \mathcal{P}(j) = R \times \mathcal{P}^{(\mathsf{static})} + \sum_{j \in \mathcal{R}} W^{\alpha}_{\mathsf{mode}(j)}. \tag{3}$$

Intuitively, this equation calls for balancing two conflicting terms: static power is minimized with few servers, while dynamic power is minimized with many servers operated in the slowest mode.

With different power modes, it is natural to refine the cost function, and to include a cost for changing the mode of a pre-existing server (upgrading it to a higher mode, or downgrading it to a lower mode). In the most detailed model, we would introduce:

- create_i, the cost for creating a new server operated at mode W_i;
- $\mathsf{changed}_{i,i'}$, the cost for changing the mode of a pre-existing server from W_i to $W_{i'}$; and
- delete_i, the cost for deleting a pre-existing server operated at mode W_i.

Note that it is reasonable to let $\mathsf{changed}_{i,i} = 0$ (no change); values of $\mathsf{changed}_{i,i'}$ with $i < i'$ correspond to upgrade costs, while values with $i' < i$ correspond to downgrade costs. In accordance with these new cost parameters, given a solution $\mathcal{R}$, we count the number of servers as follows:

- n_i, the number of new servers operated at mode W_i;
- $e_{i,i'}$, the number of reused pre-existing servers whose operation modes have changed from W_i to $W_{i'}$; and
- k_i, the number of pre-existing server operated at mode W_i that have not been reused.

The cost of the solution $\mathcal{R}$ with a total of $R = \sum_{i=1}^{M} n_i + \sum_{i=1}^{M} \sum_{i'=1}^{M} e_{i,i'}$ servers becomes:

$$\mathsf{cost}(\mathcal{R}) = R + \sum_{i=1}^{M} \mathsf{create}_i \times n_i + \sum_{i=1}^{M} \mathsf{delete}_i \times k_i + \sum_{i=1}^{M} \sum_{i'=1}^{M} \mathsf{changed}_{i,i'} \times e_{i,i'}. \quad (4)$$

Of course this complicated cost function can be simplified to make the model more tractable; for instance all creation costs create_i can be set identical, all deletion costs delete_i can be set identical, all upgrade and downgrade values $\mathsf{changed}_{i,i'}$ can be set identical, and the latter can even be neglected.

2.3 Objective functions

Without power consumption constraints, the objective is to minimize the cost, as defined by Equation (2). We distinguish two optimization problems, either with pre-existing replicas in the tree or without:

- MinCost-NoPre, the classical cost optimization problem [6] without pre-existing replicas. Indeed, in that case, Equation (2) reduces to finding a solution with the minimal number of servers.
- MinCost-WithPre, the cost optimization problem with pre-existing replicas.

With power consumption constraints, the first optimization problem is MinPower, which stands for minimizing power consumption, independently of the incurred cost. But the cost of the best power-efficient solution may indeed be prohibitive, which calls for a bi-criteria approach: MinPower-BoundedCost is the problem to minimize power consumption while enforcing a threshold cost that cannot be exceeded. This bi-criteria problem can be declined in two versions, without pre-existing replicas (MinPower-BoundedCost-NoPre) and with pre-existing replicas (MinPower-BoundedCost-WithPre).

2.4 Summary of results

In this paper, we prove the following complexity results for a tree with N nodes:

Theorem 1 MinCost-WithPre can be solved in polynomial time with a dynamic programming algorithm whose worst case complexity is $O(N^5)$.

Theorem 2 MinPower is NP-complete.

Theorem 3 With a constant number M of modes, both versions of MinPower-BoundedCost can be solved in polynomial time with a dynamic programming algorithm. The complexity of this algorithm is $O(N^{2M+1})$ for MinPower-BoundedCost-NoPre and $O(N^{2M^2+2M+1})$ for MinPower-BoundedCost-WithPre.

Note that MinPower remains NP-complete without pre-existing replicas, and without static power: the proof of Theorem 2 (see Section 4.2) shows that balancing server modes across the tree already is a hard combinatorial problem. On the contrary, with a fixed number

of modes, there are polynomial-time algorithms capable of optimizing power for a bounded cost, even with pre-existing replicas, with static power and with a complex cost function. These algorithms can be viewed as pseudo-polynomial solutions to the MinPower-BoundedCost problems.

3 Complexity results: update strategies

In this section, we focus on the MinCost-WithPre problem: we need to update the set of replicas in a tree, given a set of pre-existing servers, so as to minimize the cost function.

In Section 3.1, we show on an illustrative example that the strategies need to trade-off between reusing resources and load-balancing requests on new servers: the greedy algorithm proposed in [19] for the MinCost-NoPre problem is no longer optimal. We provide in Section 3.2 a dynamic programming algorithm which returns the optimal solution in polynomial time, and we prove its correctness. The analysis of the execution time is given in Section 3.3.

3.1 Running example

We consider the example of Figure 1. There is one pre-existing replica in the tree at node B, and we need to decide whether to reuse it or not. For taking decisions locally at node A, the trade-off is the following:

- either we keep server B, and there are 7 requests going up in the tree from node A;
- either we remove server B and place a new server at node C, hence having only 4 requests going up in the tree from node A;
- either we keep the replica at node B and add one at node A or C, thereby having no traversing request any more.

The choice cannot be made locally, since it depends upon the remainder of the tree: if the root r has two client requests, then it was better to keep the pre-existing server B. However, if it has four requests, two new servers are needed to satisfy all requests, and one can then remove server B which becomes useless (i.e., keep one server at node C and one server at node r).

From this example, it seems very difficult to design a greedy strategy to minimize the solution cost, while accounting for pre-existing replicas. We propose in the next section a dynamic programming algorithm which solves the MinCost-WithPre problem.

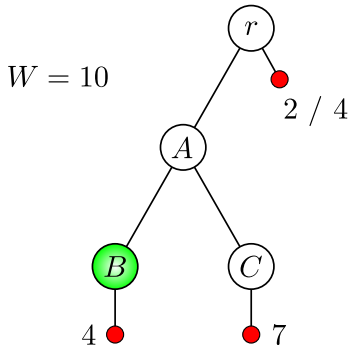

Figure 1: Example: reusing pre-existing replicas.

3.2 Dynamic programming algorithm

Let W be the total number of requests that a server can handle, and r_i the number of requests issued by client $i \in \mathcal{C}$.

At each node $j \in \mathcal{N}$, we fill a table of maximum size $(E+1) \times (N-E+1)$ which indicates, for exactly $0 \leq e \leq E$ existing servers and $0 \leq n \leq N-E$ new servers in the subtree rooted in j (excluding j), the solution which leads to the minimum number of requests that have not been processed in the subtree. This solution for (e,n) values at node j is characterized by the minimum number of requests that is obtained, $minr^j_{(e,n)}$, and by the number of requests processed at each node $j' \in \mathsf{subtree}_j$, $req^j_{(e,n)}(j')$. Note that each entry of the table has a maximum size $O(N)$ (in particular, this size is reached at the root of the tree). The req variables ensure that it is possible to reconstruct the solution once the traversal of the tree is complete.

The call **init**(r) (see Algorithm 1, p. 21), where r is the root of the tree, performs the initialization: tables are initialized to default values (no solution). We set $minr^j_{(e,n)} = W + 1$ to indicate that there is no solution, since in any valid solution, $minr^j_{(e,n)} \leq W$.

The main algorithm (see Algorithm 2, p. 21) fills the tables while performing a bottom-up traversal of the tree, and the solution can be found within the table of the root node (see Algorithm 4, p. 23). Initially, we fill the table for nodes j which have only client nodes: $minr^j_{(0,0)} = \sum_{i \in \mathsf{children}_j \cap \mathcal{C}} r_i$, and $minr^j_{(k,l)} = W+1$ for $k > 0$ or $l > 0$. There are no nodes in the subtree of j, thus no req variables to set. The variable $client(j)$ keeps track of the number of requests directly issued by a client at node j. Also, recall that the decision whether to place a replica at node j or not is not accounted for in the table of j, but when processing the parent of node j.

Then, for a node $j \in \mathcal{N}$, we perform the same initialization, before processing children nodes one by one. The processing of child i of node j is done through the call to the **merge**(j,i) procedure (see Algorithm 3, p. 22), and it is informally described below.

First, we copy the current table of node j into a temporary one, with values $tminr$ and $treq$. Note that the table is initially almost empty, but this copy is required since we process children one after the other, and when we call **merge**(j,i) for the k^{th} children node, the table of j already contains information from the merge with the previous $k-1$ children nodes.

Then, for $0 \leq e \leq E$ and $0 \leq n \leq N-E$, we need to compute the new $minr^j_{(e,n)}$, and to update the $req^j_{(e,n)}$ values. We try all combinations with e' existing replicas and n' new replicas in the temporary table (i.e., information about children already processed), $e-e'$ existing replicas and $n-n'$ new replicas in the subtree of child i. We furthermore try solutions with a replica placed at node i, and we account for it in the value of e if $i \in \mathcal{E}$ (i.e., for a given value e', we place only $e-e'-1$ replica in the subtree of i, plus one on i); otherwise we account for it in the value of n. Each time we find a solution which is better than the one previously in the table (in terms of $minr$), we copy the values of req from the temporary table and the table of i, in order to retain all the information about the current best solution.

The key of the algorithm resides in the fact that during this *merging* process, the optimal solution will always be one which lets the minimum of requests pass through the subtree (see Lemma 1).

The solution to the replica placement problem with pre-existing servers MinCost-WithPre is computed through a call to **replica-update** (see Algorithm 4), which returns a set of replica $\mathcal{R}$ minimizing the cost: we scan all solutions in order to return a valid one of minimum cost.

To prove that the algorithm returns an optimal solution, we show in Lemma 1 that the solutions that are discarded while filling the tables, never lead to a better solution than the one

that is finally returned.

Lemma 1 Consider a subtree rooted at node $j \in \mathcal{N}$. If an optimal solution uses e pre-existing servers and places n new servers in this subtree, then there exists an optimal solution of same cost, for which the placement of these servers minimizes the number of requests traversing j.

Proof 1 *Let $\mathcal{R}_{opt}$ be the set of replicas in the optimal solution with (e, n) servers (i.e., e pre-existing and n new in $\mathsf{subtree}_j$). We denote by rmin the minimum number of requests that must traverse j in a solution using (e, n) servers, and by $\mathcal{R}_{loc}$ the corresponding (local) placement of replicas in $\mathsf{subtree}_j$.*

If $\mathcal{R}_{opt}$ is such that more than rmin requests are traversing node j, we can build a new global solution which is similar to $\mathcal{R}_{opt}$, except for the subtree rooted in j for which we use the placement of $\mathcal{R}_{loc}$. The cost of the new solution is identical to the cost of $\mathcal{R}_{opt}$, therefore it is an optimal solution. It is still a valid solution, since $\mathcal{R}_{loc}$ is a valid solution and there are less requests than before to handle in the remaining of the tree (only rmin requests traversing node j).

This proves that there exists an optimal solution which minimizes the number of requests traversing each node, given a number of pre-existing and new servers.

The algorithm computes all local optimal solutions for all values (e, n). During the merge procedure, we try all possible numbers of pre-existing and new servers in each subtree, and we minimize the number of traversing requests, thus finding an optimal local solution. Thanks to Lemma 1, we know that there is a global optimal solution which builds upon these local optimal solutions.

3.3 Execution time of the algorithm

Recall that N is the total number of nodes, and E is the number of pre-existing nodes.

The call to **init**(r) makes a traversal of the tree, and at each node, the table of size $O((E+1)\times(N-E+1))$ is initialized. The total cost for this call is therefore in $O(N\times(N-E+1)\times(E+1))$.

For the main procedure, the processing of a node with only client children is done in constant time $O(1)$. The processing of each non-client child consists in a call to the **merge** procedure, and there is only one such per node of the tree, and therefore N calls to this procedure during the whole execution.

The initialization of the merging procedure takes a time $O((N-E+1)\times(E+1))$. Then, we try all solutions with $e \leq E$ existing replicas, and $n \leq N-E$ new replicas. Given e and n, there are no more than $O((N-E+1)\times(E+1))$ possible solutions ($0 \leq e' \leq e \leq E$ existing replicas and $0 \leq n' \leq n \leq N-E$ new replicas on the children already processed, with or without a replica on the child currently being processed). Finally, the total number of iterations in the loop is bounded by $O((N-E+1)^2\times(E+1)^2)$. The most consuming operation in the loop is to copy the *req* variables, which is done in $O(N)$. However, this copy can be done outside the loop: we keep track of the best solution for each couple (e, n), and update the *req* variables in another loop over (e, n). It is done by decreasing values of e and n, since the update for (e, n) requires the non-updated values with (e', n') such that $e' \leq e$ and $n' \leq n$. The total cost with this optimization (see [17] for the implementation) is therefore the number of iterations, i.e., $O((N-E+1)^2\times(E+1)^2)$.

Then, scanning the table at the root is done in $O((N-E+1)\times(E+1))$, and reconstructing the solution takes a single tree traversal, i.e., it is in $O(N)$. Finally, the complexity of the dynamic closest replica placement algorithm is in $O(N\times(N-E+1)^2\times(E+1)^2)$, which corresponds to the N calls to the merging procedure. The algorithm is therefore of polynomial complexity, at most $O(N^5)$ for a tree with N nodes. This concludes the proof of Theorem 1.

4 Complexity results with power

In this section, we tackle the MinPower and MinPower-BoundedCost problems. First in Section 4.1, we use an example to show why minimizing the number of requests traversing the root of a subtree is no longer optimal, and we illustrate the difficulty to take local decisions even when restricting to the simpler mono-criterion MinPower problem. Then in Section 4.2, we prove the NP-completeness of the latter problem with an arbitrary number of modes (Theorem 2). However, we propose a pseudo-polynomial algorithm to solve the problem in Section 4.3. This algorithm turns out to be polynomial when the number of modes is constant, hence usable in a realistic setting with two or three modes (Theorem 3).

4.1 Running example

Consider the example of Figure 2. There are two modes, $W_1 = 7$ and $W_2 = 10$, and we focus on the power minimization problem. For simplicity, we assume that the power consumption of a node running at mode W_i is $10 + W_i^2$, for $i = 1, 2$ (10 is the static power, and we set $\alpha = 2$). We consider the subtree rooted in A. Several decisions can be taken locally:

- place a server at node A, running at mode W_2, hence minimizing the number of traversing requests. Another solution without traversing requests is to have two servers, one at node B and one at node C, both running at mode W_1, but this would lead to a higher power consumption, since $20 + 2 \times 7^2 > 10 + 10^2$;
- place a server running at mode W_1 at node C, thus having 3 requests going through node A.

The choice cannot be made greedily, since it depends upon the rest of the tree: if the root r has four client requests, then it is better to let some requests through (one server at node C), since it optimizes power consumption. However, if it has ten requests, it is necessary to have no request going through A, otherwise node r is not able to process all its requests.

From this example, it seems very hard to design a greedy strategy to minimize the power consumption. Similarly, if we would like to reuse the algorithm of Section 3 to solve the MinPower-BoundedCost-WithPre bi-criteria problem, we would need to account for modes. Indeed, the best solution of subtree A with one server is no longer always the one which minimizes the number of requests (in this case, placing one server on node A), since it can be better for power consumption to let three requests traverse node A and balance the load upper in the tree.

We prove in the next section the NP-completeness of the problem, when the number of modes is arbitrary. However, we can adapt the dynamic programming algorithm, which becomes

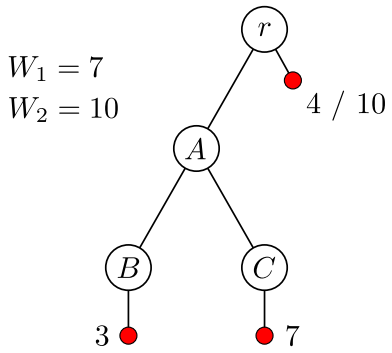

Figure 2: Example: minimizing power consumption.

exponential in the number of modes, but hence remains polynomial for a constant number of modes (see Section 4.3).

4.2 NP-completeness of MinPower

In this section, we prove Theorem 2, i.e., the NP-completeness of the MinPower problem, even with no static power, when there is an arbitrary number of modes.

Proof 2 (Proof of Theorem 2) *We consider the associated decision problem: given a total power consumption $\mathcal{P}$, is there a solution which does not consume more than $\mathcal{P}$?*

First, the problem is clearly in NP: given a solution, i.e., a set of servers, and the mode of each server, it is easy to check in polynomial time that no capacity constraint is exceeded, and that the power consumption meets the bound.

To establish the completeness, we use a reduction from 2-Partition [7]. We consider an instance $\mathcal{I}_1$ of 2-Partition: given n strictly positive integers $a_1, a_2, \ldots, a_n$, does there exist a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? Let $S = \sum_{i=1}^{n} a_i$; we assume that S is even (otherwise there is no solution).

We build an instance $\mathcal{I}_2$ of our problem where each server has $n+2$ modes. We assume that the a_i are sorted in increasing order, i.e., $a_1 \leq \cdots \leq a_n$. The modes are then, in increasing order:

- $W_1 = K$;
- $\forall 1 \leq i \leq n$, $W_{i+1} = K + a_i \times X$;
- $W_{n+2} = K + S \times X$;

where the values of K and X will be determined later.

We furthermore set that there is no static power, and the power consumption for a server running at capacity W_i is therefore $\mathcal{P}_i = W_i^{\alpha}$, where α is the rational exponent used in the computation of the power (see Section 2), and $2 \leq \alpha \leq 3$. The idea is to have K large and X small, so that we have an upper bound on the power consumed by a server running at capacity W_{i+1}, for $1 \leq i \leq n$:

$$W_{i+1}^{\alpha} = (K + a_i \times X)^{\alpha} \leq K^{\alpha} + a_i + \frac{1}{n}. \tag{5}$$

To ensure that Equation (5) is satisfied, we set

$$X = \frac{1}{\alpha \times K^{\alpha - 1}},$$

and then we have $(K + a_i \times X)^{\alpha} = K^{\alpha}(1 + \frac{a_i}{\alpha K^{\alpha}})^{\alpha}$, with $K > S$ and therefore $\frac{a_i}{\alpha K^{\alpha}} < 1$. We set $x_i = \frac{a_i}{\alpha K^{\alpha}}$, and we want to ensure that:

$$(1 + x_i)^{\alpha} \leq 1 + \alpha \times x_i + \frac{1}{n \times K^{\alpha}}. \tag{6}$$

To do so, we study the function

$$f(x) = (1 + x)^{\alpha} - (1 + \alpha \times x) - 5x^2.$$

We have $f(0) = 0$, and $f'(x) = \alpha(1 + x)^{\alpha - 1} - \alpha - 10x$. We have $f'(0) = 0$, and $f''(x) = \alpha(\alpha - 1)(1 + x)^{\alpha - 2} - 10$. Since $\alpha \leq 3$, $\alpha(\alpha - 1)(1 + x)^{\alpha - 2} \leq 6(1 + x)$, and for $x \leq \frac{1}{2}$, $f''(x) < 0$. We deduce that $f'(x)$ is non increasing for $x \leq \frac{1}{2}$, and since $f'(0) = 0$, $f'(x)$ is

negative for $x \leq \frac{1}{2}$. *Finally,* $f(x)$ *is non increasing for* $x \leq \frac{1}{2}$, *and since* $f(0) = 0$, *we have* $(1+x)^\alpha < (1 + \alpha \times x) + 5x^2$ *for* $x \leq \frac{1}{2}$.

Equation (6) is therefore satisfied if $5x_i^2 \leq \frac{1}{n \times K^\alpha}$, *i.e.,* $K^\alpha \geq \frac{5a_i^2 \times n}{\alpha^2}$. *This condition is satisfied for*

$$K = n \times S^2,$$

and we then have $x_i < \frac{1}{2}$, *which ensures that the previous reasoning was correct. Finally, with these values of* K *and* X, *Equation (5) is satisfied.*

Then, the distribution tree is the following: the root node r *has one client with* $K + \frac{S}{2} \times X$ *requests, and* n *children* $A_1, \ldots, A_n$. *Each node* A_i *has a client with* $a_i \times X$ *requests, and a children node* B_i *which has* K *requests. Figure 3 illustrates the instance of the reduction.*

Finally, we ask if we can find a placement of replicas with a maximum power consumption of:

$$\mathcal{P}_{max} = (K + S \times X)^\alpha + n \times K^\alpha + \frac{S}{2} + \frac{n-1}{n}.$$

Clearly, the size of $\mathcal{I}_2$ *is polynomial in the size of* $\mathcal{I}_1$, *since* K *and* X *are of polynomial size. We now show that* $\mathcal{I}_1$ *has a solution if and only if* $\mathcal{I}_2$ *does.*

Let us assume first that $\mathcal{I}_1$ *has a solution,* I. *The solution for* $\mathcal{I}_2$ *is then as follows: there is one server at the root, running at capacity* W_{n+2}. *Then, for* $i \in I$, *we place a server at node* A_i *running at capacity* W_{1+i}, *while for* $i \notin I$, *we place a server at node* B_i *running at capacity* W_1. *It is easy to check that all capacity constraints are satisfied for nodes* A_i *and* B_i. *At the root of the tree, there are* $K + \frac{S}{2} \times X + \sum_{i \notin I} a_i \times X$, *which sums up to* $K + S \times X$. *The total power consumption is then* $\mathcal{P} = (K + S \times X)^\alpha + \sum_{i \in I}(K + a_i \times X)^\alpha + \sum_{i \notin I} K^\alpha$. *Thanks to Equation (5),* $\mathcal{P} \leq (K + S \times X)^\alpha + \sum_{i \in I}\left(K^\alpha + a_i + \frac{1}{n}\right) + \sum_{i \notin I} K^\alpha$, *and finally,* $\mathcal{P} \leq (K + S \times X)^\alpha + n \times K^\alpha + \sum_{i \in I} a_i + \frac{n-1}{n}$. *Since* I *is a solution to 2-Partition, we have* $\mathcal{P} \leq \mathcal{P}_{max}$. *Finally,* $\mathcal{I}_2$ *has a solution.*

Suppose now that $\mathcal{I}_2$ *has a solution. There is a server at the root node* r, *which runs at mode* W_{n+2}, *since this is the only way to handle its* $K + \frac{S}{2} \times X$ *requests. This server has a power consumption of* $(K + S \times X)^\alpha$. *Then, there cannot be more than* n *other servers. Indeed, if there were* $n+1$ *servers, running at the smallest mode* W_1, *their power consumption would be* $(n+1)K^\alpha$, *which is strictly greater than* $n \times K^\alpha + \frac{S}{2} + 1$. *Therefore, the power consumption would exceed* $\mathcal{P}_{max}$. *So, there are at most* n *extra servers.*

Consider that there exists $i \in \{1, \ldots, n\}$ *such that there is no server, neither on* A_i *nor on* B_i. *Then, the number of requests at node* r *is at least* $2K$; *however,* $2K > W_{n+2}$, *so the server*

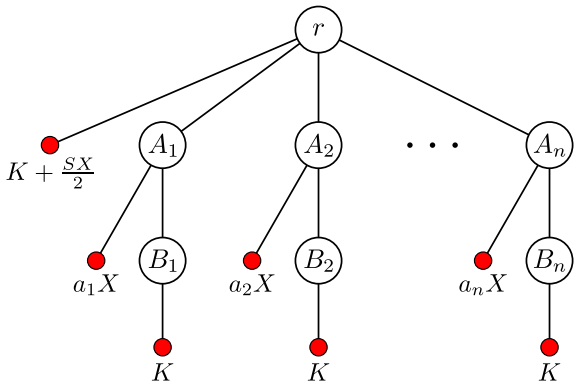

Figure 3: Illustration of the NP-completeness proof.

cannot handle all these requests. Therefore, for each $i \in \{1, \ldots, n\}$, there is exactly one server either on A_i or on B_i. We define the set I as the indices for which there is a server at node A_i in the solution. Now we show that I is a solution to $\mathcal{I}_1$, the original instance of 2-Partition.

First, if we sum up the requests at the root node, we have:

$$K + \frac{S}{2} \times X + \sum_{i \notin I} a_i \times X \leq K + S \times X.$$

Therefore, $\sum_{i \notin I} a_i \leq \frac{S}{2}$.

Now, if we consider the power consumption of the solution, we have:

$$(K + S \times X)^\alpha + \sum_{i \in I} (K + a_i \times X)^\alpha + \sum_{i \notin I} K^\alpha \leq \mathcal{P}_{max}.$$

Let us assume that $\sum_{i \in I} a_i > \frac{S}{2}$. Since the a_i are integers, we have $\sum_{i \in I} a_i \geq \frac{S}{2} + 1$. It is easy to see that $(K + a_i \times X)^\alpha > K^\alpha + a_i$. Finally, $\sum_{i \in I} (K + a_i \times X)^\alpha + \sum_{i \notin I} K^\alpha \geq n \times K^\alpha + \sum_{i \in I} a_i \geq n \times K^\alpha + \frac{S}{2} + 1$. This implies that the total power consumption is greater than $\mathcal{P}_{max}$, which leads to a contradiction, and therefore $\sum_{i \in I} a_i \leq \frac{S}{2}$.

We derive that $\sum_{i \notin I} a_i = \sum_{i \in I} a_i = \frac{S}{2}$, the solution is a 2-Partition for instance $\mathcal{I}_1$. This concludes the proof.

4.3 A pseudo-polynomial algorithm for MinPower-BoundedCost

In this section, we sketch how to adapt the algorithm of Section 3 to account for power consumption. As illustrated in the example of Section 4.1, the current algorithm may lead to a non-optimal solution for the power consumption if used only with the higher mode for servers. Therefore, we refine it and compute, in each subtree, the optimal solution with, for $1 \leq j, j' \leq M$,

- exactly n_j new servers running at mode W_j;
- exactly $e_{j,j'}$ pre-existing servers whose operation modes have changed from W_j to $W_{j'}$.

Recall that we previously had only two parameters, n the number of new servers, and e the number of pre-existing servers, thus leading to a total of $(N - E + 1)^2 \times (E + 1)^2$ iterations for the **merge** procedure (Lines 8-9 of Algorithm 3). Now, the number of iterations is $(N - E + 1)^{2M} \times (E + 1)^{2M^2}$, since we have $2 \times M$ loops of maximum size $N - E + 1$ over the n_j and n'_j, and $2 \times M^2$ loops of maximum size $E + 1$ over the $e_{j,j'}$ and $e'_{j,j'}$.

The new algorithm is similar, except that during the merge procedure, we must consider the type of the current node that we are processing (existing or not), and furthermore set it to all possible modes. This is done at Lines 16 and 23 of Algorithm 3, when we try to add a server at node i. We therefore add a loop of size M.

We do not formalize the new **merge** procedure, since its principle is similar to Algorithm 3, except that we need to have larger tables at each node, and to iterate over all parameters. The complexity of the N calls to this procedure is now in $O(N \times M \times (N - E + 1)^{2M} \times (E + 1)^{2M^2})$.

Of course, we need also to update the **init** and **main** procedures to account for the increasing number of parameters. Finally, we rewrite the equivalent of Algorithm 4 but according to the bi-criteria objective function: first we compute all costs, accounting for the cost of changing modes, and then we scan all solutions, and return one whose cost is not greater than the threshold, and which minimizes the power consumption. The most time-consuming part of the algorithm is still the call to the **merge** procedures, hence a complexity in $O(N \times M \times (N - E + 1)^{2M} \times (E + 1)^{2M^2})$.

With a constant number of capacities, this algorithm is polynomial, which proves Theorem 3. For instance, with $M = 2$, the worst case complexity is $O(N^{13})$. Without pre-existing servers, this complexity is reduced to $O(N^5)$.

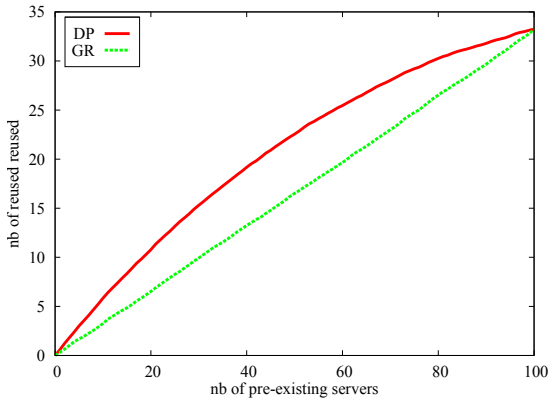

Figure 4: Experiment 1: increasing number of pre-existing servers.

5 Simulations

In this section, we compare our algorithms with the algorithms of [19], which do not account for pre-existing servers and for power consumption. First in Section 5.1, we focus on the impact of pre-existing servers. Then we consider the power consumption minimization criterion in Section 5.2.

Experiments were conducted using a platform based on an Intel Xeon 5250 processor. Source code of all algorithms and simulations is publicly available on the Web [17].

5.1 Impact of pre-existing servers

In this set of experiments, we randomly build a set of distribution trees with $N = 100$ internal nodes of maximum capacity $W = 10$. Each internal node has between 6 and 9 children, and clients are distributed randomly throughout the tree: each internal node has a client with a probability 0.5, and this client has between 1 and 6 requests.

In the first experiment, we draw 200 random trees without any existing replica in them. Then we randomly add $0 \leq E \leq 100$ pre-existing servers in each tree. Finally, we execute both the greedy algorithm (GR) of [19], and the algorithm of Section 3 (DP) on each tree, and since both algorithms return a solution with the minimum number of replicas, the cost of the solution

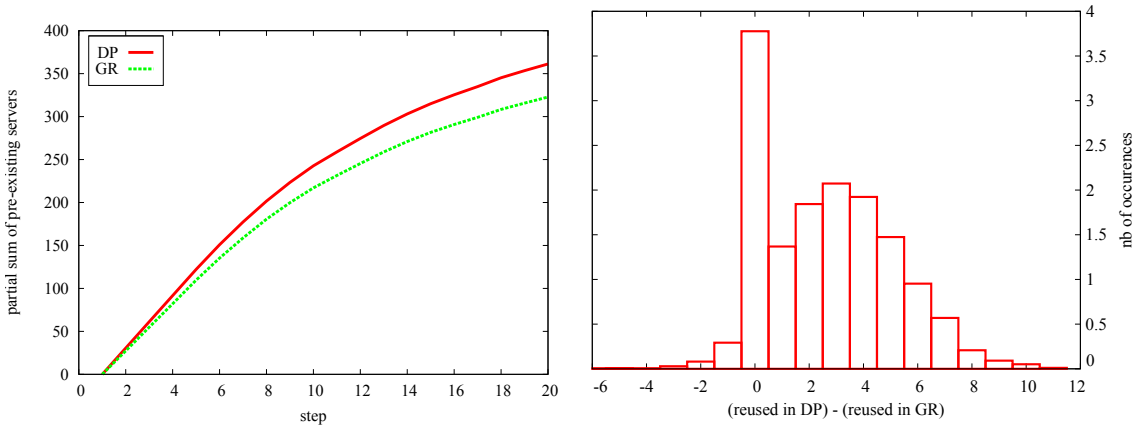

Figure 5: Experiment 2: consecutive executions of the algorithms.

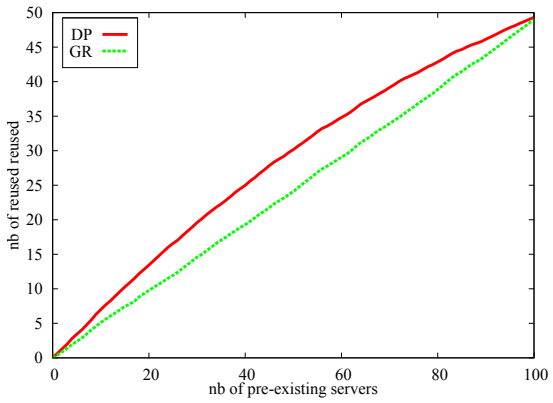

Figure 6: Experiment 1 with high trees.

is directly related to the number of pre-existing replicas that are reused. Figure 4 shows the average number of pre-existing servers that are reused in each solution over the 200 trees, for each value of the number E of pre-existing servers. When the tree has a very small ($E \approx 0$) or very large ($E \approx N$) number of pre-existing replicas, both algorithms will return the same solution. Still, DP achieves an average reuse of 4.13 more servers than GR, and it can reuse up to 15 more servers.

In a second experiment, we study the behavior of the algorithm in a *dynamic* setting, with 20 update steps. At each step, starting from the current solution, we update the number of requests per client and recompute an optimal solution with both algorithms, starting from the servers that were placed at the previous step. Initially, there are no pre-existing servers, and at each step, both algorithms obtain a different solution. However, they always reach the same total number of servers since they have the same requests; but after the first step, they may have a different set of pre-existing servers. Similarly to Experiment 1, the simulation is conducted on 200 distinct trees, and results are averaged over all trees. In Figure 5 (left), at each step, we compare the number of existing replicas in the solutions found by the two algorithms, and hence the cost of the solutions. We plot the cumulative number of servers that have been reused so far (hence accounting for all previous steps). As expected, the DP algorithm makes a better reuse of pre-existing replicas. Figure 5 (right) compares, at each step, the number of

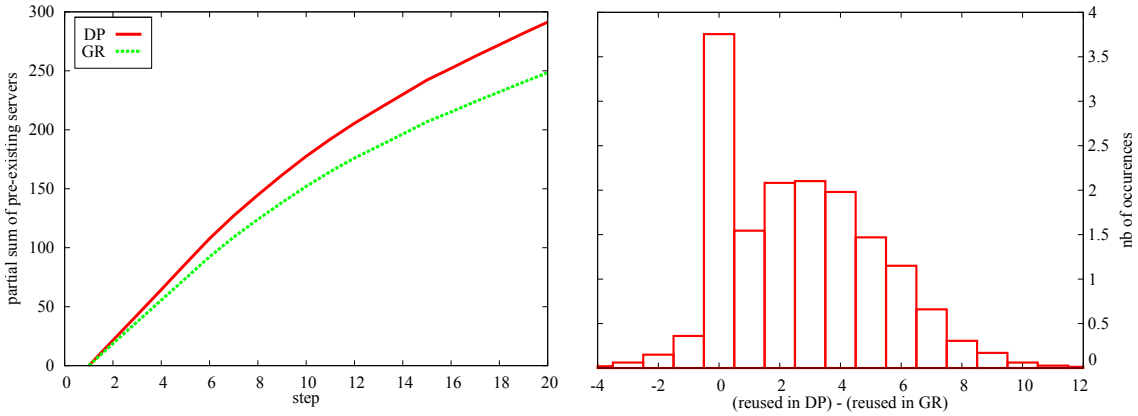

Figure 7: Experiment 2 with high trees.

pre-existing servers reused by DP and by GR. We count the average number of steps (over 20) at which each value is reached. It occasionally happens that the greedy algorithm performs a better reuse, because it is not starting from the same set of pre-existing servers, but overall this experiment confirms the better reuse of the dynamic programming algorithm, even when the algorithms are applied on successive steps.

Note however that taking pre-existing replicas into account has an impact on the execution time of the algorithm: in these experiments, GR runs in less than one second per tree, while DP takes around forty seconds per tree.

Also, we point out that the shape of the trees does not seem to modify the general behaviour: the results with trees where each node has between 2 and 4 children are depicted in Figure 6 and Figure 7.

5.2 With power consumption

To study the practical applicability of the bi-criteria algorithm (DP) for the MinPower-BoundedCost problem (see Section 4.3), we have implemented it with two modes $W_1 = 5$ and $W_2 = 10$, and compared it with the algorithm in [19]; this algorithm does not account for power minimization, but minimizes the value of the maximal capacity W when given a cost bound. More precisely, in the experiment we try all values $5 \leq W \leq 10$, and compute the corresponding cost and power consumption. To be fair, when a server has 5 requests or less, we operate it

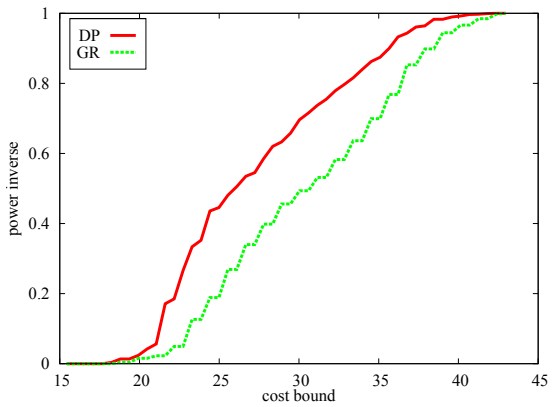

Figure 8: Experiment 3: Power minimization.

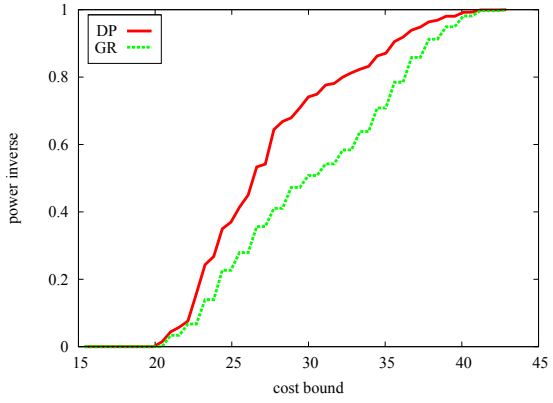

Figure 9: Experiment 3 without pre-existing replica.

under the first mode W_1. Given a bound on the cost, we keep the solution that minimizes the power consumption. We call GR this version of the algorithm in [19] modified for power as explained above.

We randomly build 100 trees with 50 nodes each, and we select 5 nodes as pre-existing servers. Clients have between 1 and 5 requests, so that a solution with replicas in the first mode can always be found. The cost function is such that, for any $i, i' \in \{1, 2\}$, $\mathsf{create}_i = 0.1$, $\mathsf{delete}_i = 0.01$ and $\mathsf{changed}_{i,i'} = 0.001$. The power consumed by a server in mode i is $\mathcal{P}_i = \frac{1}{10}W_1^3 + W_i^3$. In Figure 8, we plot the inverse of the power of a solution, given a bound on the cost (the higher the better). If the algorithm fails to find a solution for a tree, the value is 0, and we average the inverse of the power over the 100 trees, for both algorithms. For intermediate cost values, our algorithm is much better than the version of [19] in terms of power consumption: GR consumes in average more than 30% more power than DP, when the cost bound is between 29 and 34.

Here again, it takes more time to obtain the optimal solution with DP than to run the greedy algorithm several times: GR runs in around one second per tree, while DP takes around five minutes per tree. Also, we have performed some more experiments with slightly different parameters, and got some little differences.

Firstly we look at the power part of the DP algorithm, running on trees without pre-existing replicas (see Figure 9). For low bound costs the two curves are close together because DP finds a solution if and only if GR finds a solution, and the dissipated power is high; and there is no significant difference for other costs.

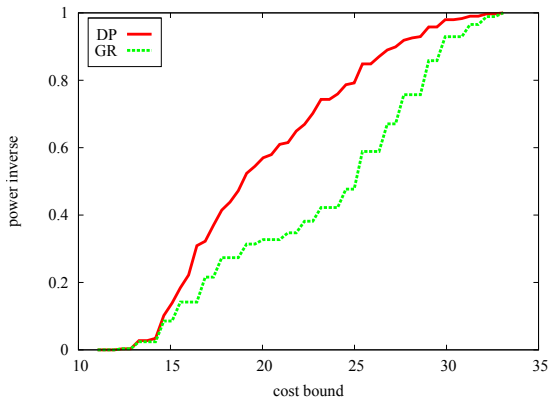

Figure 10: Experiment 3 with high trees.

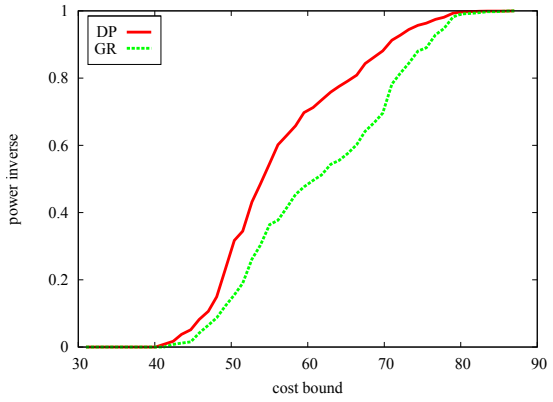

Figure 11: Experiment 3 with different cost.

Then, we run the experiment on high trees (each internal node has from 2 to 4 children). Results are shown in Figure 10. The ratio between the dynamic programming algorithm and the greedy one is better than the ratio on fat trees for intermediate costs: when the bound cost is between 22 and 27, GR consumes up in average more than 40% more power than DP, and 60% between 23 and 25.

In Figure 11, we show the results for a cost function such that deleting and creating costs are high (more precisely, for any $i, i' \in \{1, 2\}$, $\mathsf{create}_i = \mathsf{delete}_i = 1$ and $\mathsf{changed}_{i,i'} = 0.1$). Compared to the first cost, the ratio between DP and GR is better for lowest cost, because GR find less solution than DP. DP indeed can find solution with lower cost, taking pre-existing replicas into account.

Finally, we would like to point out that the DP algorithm scales reasonably well: without power, we are able to process trees with 500 nodes and 125 pre-existing servers in 30 minutes; with power and no pre-existing server, we can process trees with 300 nodes in one hour. The algorithm with power and pre-existing servers is the most time-consuming: it takes around one hour to process a tree with 70 nodes and 10 pre-existing servers.

6 Conclusion

In this paper, we have addressed the problem of updating the placement of replicas in a tree network. We have provided an optimal dynamic programming algorithm whose cost is at most $O(N^5)$, where N is the number of nodes in the tree. This complexity may seem high for very large problem sizes, but our implementation of the algorithm is capable of managing trees with up to 500 nodes in half an hour, which is reasonable for a large spectrum of applications (e.g., such as database updates during the night).

The optimal placement update algorithm is a first step towards dealing with dynamic replica management. When client requests evolve over time, the placement of the replicas must be updated at regular intervals, and the overall cost is a trade-off between two extreme strategies: (i) "lazy" updates, where there is an update only when the current placement is no longer valid; the update cost is minimized, but changes in request volume and location since the last placement may well lead to poor resource usage; and (ii) systematic updates, where there is an update every time-step; this leads to an optimized resource usage but encompasses a high update cost. Clearly, the rates and amplitudes of the variations of the number of requests issued by each client in the tree are very important to decide for a good update interval. Still, establishing the cost of an update is a key result to guide such a decision. When un-frequent updates are called for, or when resources have a high cost, the best solution is likely to use our optimal but expensive algorithm. On the contrary, with frequent updates or low-cost servers, we may prefer to resort to faster (but sub-optimal) update heuristics.

Our main contribution is to have provided the theoretical foundations for a single step reconfiguration, whose complexity is important to guide the design of lower-cost heuristics. Also, we have done a first attempt to take power consumption into account, in addition to usual performance-related objectives. Power consumption has become a very important concern, both for economic and environmental reasons, and it is important to account for it when designing replica placement strategies.

Even though our optimal algorithms have a high worst-case complexity, we have successfully implemented all of them, including the most time-consuming scheme capable of optimizing power while enforcing a bounded cost that includes pre-existing servers. We were able to process trees with a reasonable number of nodes. As future work, we plan to design polynomial time heuristics with a lower complexity than the optimal solution. The idea would be to perform some

local optimizations to better load-balance the number of requests per replica, with the goal of minimizing the power consumption. These heuristics should be tuned for dedicated applications, and should (hopefully!) build upon the fundamental results (complexity and algorithms) that we have provided in this paper.

Acknowledgment

The authors are with Université de Lyon, France. A. Benoit and Y. Robert are with the Institut Universitaire de France. This work was supported in part by the ANR *StochaGrid* project.

References

[1] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 113–121, 2003.

[2] A. Benoit, V. Rehn-Sonigo, and Y. Robert. Replica placement and access policies in tree networks. *IEEE Trans. Parallel and Distributed Systems*, 19(12):1614–1627, 2008.

[3] A. P. Chandrakasan and A. Sinha. Jouletrack: A web based tool for software energy profiling. In *Design Automation Conference*, pages 220–225. IEEE Computer Society Press, 2001.

[4] J.-J. Chen and T.-W. Kuo. Multiprocessor energy-efficient scheduling for real-time tasks. In *Proceedings of International Conference on Parallel Processing (ICPP)*, pages 13–20. IEEE CS Press, 2005.

[5] Y. Chen, R. H. Katz, and J. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *First Int. Workshop on Peer-to-Peer Systems (IPTPS'01)*, pages 306–318, London, UK, 2002. Springer-Verlag.

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

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

[8] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[9] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202. ACM Press, 1998.

[10] 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.

[11] M. Larabel. Intel EIST SpeedStep.

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

[13] M. P. Mills. The internet begins with coal. *Environment and Climate News*, 1999.

[14] K. Pruhs, R. van Stee, and P. Uthaisombut. Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, 43:67–80, 2008.

[15] R. M. Rahman, K. Barker, and R. Alhajj. Effective dynamic replica maintenance algorithm for the grid environment. In *Advances in Grid and Pervasive Computing*, volume 3947, pages 336–345. Springer LNCS 3947, 2006.

[16] R. M. Rahman, K. Barker, and R. Alhajj. Replica placement design with static optimality and dynamic maintainability. In *IEEE Int. Symp. on Cluster Computing and the Grid (CCGRID'06)*, pages 434–437, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[17] P. Renaud-Goud. Source code for the simulations.

[18] M. Shorfuzzaman, P. Graham, and R. Eskicioglu. Adaptive popularity-driven replica placement in hierarchical data grids. *J. Supercomputing*, 51(3):374–392, 2010.

[19] J.-J. Wu, Y.-F. Lin, and P. Liu. Optimal replica placement in hierarchical Data Grids with locality assurance. *J. Parallel and Distributed Computing*, 68(12):1517–1538, 2008.

Appendices

Algorithm 1: Initialization procedure.

```
1  procedure init (node j ∈ 𝒩)
2  begin
       /* Initializing the tables.  */
3      for 0 ≤ e ≤ E do
4          for 0 ≤ n ≤ N − E do
               /* No solution.  */
5              minr^j_(e,n) = W + 1;
       /* Recursive call.  */
6      for i ∈ children_j ∩ 𝒩 do
7          init(i);
8  end
9  end
```

Algorithm 2: Main procedure.

```
1  procedure main (node j ∈ 𝒩)
2  begin
       /* Init.  client children.  */
3      client(j) = 0;
4      for i ∈ children_j ∩ 𝒞 do
5          client(j) = client(j) + r_i;
6      minr^j_(0,0) = client(j);
7      if minr^j_(0,0) > W then exit(no solution);
       /* Processing child nodes.  */
8      for i ∈ children_j ∩ 𝒩 do
9          main(i); /* Recursive call.  */
10         merge(j, i);
11 end
12 end
```

Algorithm 3: Processing a child node.

```
1  procedure merge(j, i)
2  begin
       /* Duplicate table at node j, and clean up.  */
3      for 0 ≤ e ≤ E do
4          for 0 ≤ n ≤ N − E do
5              tminr_(e,n) = minr^j_(e,n);
6              minr^j_(e,n) = W + 1; /* No solution in the merged table.  */
7              for j' ∈ subtree_j ∩ N do treq_(e,n)(j') = req^j_(e,n)(j');
       /* Try all solutions with e existing replicas and n new replicas.  */
8      for 0 ≤ e ≤ E do  for 0 ≤ n ≤ N − E do
9          for 0 ≤ e' ≤ e do  for 0 ≤ n' ≤ n do
10             if tminr_(e',n') ≤ W then
                   /* e' existing and n' new on children already processed, e − e'
                      existing and n − n' new in the subtree of i, no replica on i.
                      */
11                 if minr^i_(e−e',n−n') + tminr_(e',n') ≤ min(W, minr^j_(e,n)) then
                       /* Better solution than existing one for (e, n).  */
12                     minr^j_(e,n) = minr^i_(e−e',n−n') + tminr_(e',n');
13                     for j' ∈ subtree_j ∩ N do
14                         if j' ∈ subtree_i then req^j_(e,n)(j') = req^i_(e−e',n−n')(j');
15                         else req^j_(e,n)(j') = treq_(e',n')(j');
                   /* e' existing and n' new on children already processed,
                      replica on i.  */
16                 if (i ∈ E) and (e' < e) then
                       /* e − e' − 1 existing and n − n' new in the subtree of i.  */
17                     if tminr_(e',n') ≤ minr^j_(e,n) then
                           /* Better solution than existing one for (e, n).  */
18                         minr^j_(e,n) = tminr_(e',n');
19                         for j' ∈ subtree_j ∩ N do
20                             if j' ∈ subtree_i then req^j_(e,n)(j') = req^i_(e−e'−1,n−n')(j');
21                             else req^j_(e,n)(j') = treq_(e',n')(j');
22                         req^j_(e,n)(i) = minr^i_(e−e'−1,n−n');
23             else if (i ∉ E) and (n' < n) then
                       /* e − e' existing and n − n' − 1 new in the subtree of i.  */
24                     if tminr_(e',n') ≤ minr^j_(e,n) then
                           /* Better solution than existing one for (e, n).  */
25                         minr^j_(e,n) = tminr_(e',n');
26                         for j' ∈ subtree_j ∩ N do
27                             if j' ∈ subtree_i then req^j_(e,n)(j') = req^i_(e−e',n−n'−1)(j');
28                             else req^j_(e,n)(j') = treq_(e',n')(j');
29                         req^j_(e,n)(i) = minr^i_(e−e',n−n'−1);
30 end
31 end
```

Algorithm 4: Replica placement algorithm with pre-existing servers (MinCost-WithPre problem).

```
1  algorithm replica-update
2  begin
3      init(r);
4      main(r);
       /* Initially, no best solution.  */
5      cmin = N × (1 + create + delete);
6      minEN = (−1, −1);
       /* Scanning root table:  compute all costs for (e, n).  */
7      for 0 ≤ e ≤ E do
8          for 0 ≤ n ≤ N − E do
9              req^r_(e,n)(r) = minr^r_(e,n);
10             cost = N × (1 + create + delete);
11             if minr^r_(e,n) = 0 then
12                 cost = (e + n) + n × create + (E − e) × delete ;
13             else if minr^r_(e,n) ≤ W and r ∈ E then
14                 cost = (e + n + 1) + n × create + (E − e − 1) × delete ;
15             else if minr^r_(e,n) ≤ W and r ∈ N \ E then
16                 cost = (e + n + 1) + (n + 1) × create + (E − e) × delete ;
               /* Check if this solution is better than previous best.  */
17             if cost < cmin then
18                 cmin = cost; minEN = (e, n);
       /* Reconstruct solution:  R is the set of replicas.  */
19     if minEN = (−1, −1) then exit(no solution);
20     else
21         R = ∅;
22         for j ∈ N do
23             if req^r_minEN(j) > 0 then R = R ∪ {j};
24         return(R);
25 end
26 end
```