Optimal algorithms and approximation algorithms for replica placement with distance constraints in tree networks

A. Benoit[1], H. Larchevêque[2], P. Renaud-Goud[1]
1. LIP, Ecole Normale Supérieure de Lyon, France, {Anne.Benoit|Paul.Renaud-Goud}@ens-lyon.fr
2. LABRI, University of Bordeaux I, France, hubert.larcheveque@labri.fr

***Abstract*—In this paper, we study the problem of replica placement in tree networks subject to server capacity and distance constraints. The client requests are known beforehand, while the number and location of the servers are to be determined. The *Single* policy enforces that all requests of a client are served by a single server in the tree, while in the *Multiple* policy, the requests of a given client can be processed by multiple servers, thus distributing the processing of requests over the platform. For the *Single* policy, we prove that all instances of the problem are NP-hard, and we propose approximation algorithms. The problem with the *Multiple* policy was known to be NP-hard with distance constraints, but we provide a polynomial time optimal algorithm to solve the problem in the particular case of binary trees when no request exceeds the server capacity.**

***Keywords*-Replica placement, distance constraints, optimal algorithms, approximation algorithms, tree networks, binary tree, single vs multiple policy.**

I. Introduction

We revisit the well-known replica placement problem in tree networks [1], [2], [3], and derive new complexity results and approximation algorithms. 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. A client is a leaf node of the tree, and it may either process its requests locally, or forward them to a server further up in the tree. Note that the distribution tree (clients, nodes, number of requests) is fixed in the approach. This key assumption is quite natural for a broad spectrum of applications, such as electronic, Internet Service Provider, or Video on Demand service delivery (see [4], [1], [5] and additional references in [2]). 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. The objective is to decide where to place replicas, and which requests each server will be processing, so as to minimize the number of servers. When equipped with a replica, a node can process a number of requests, up to its capacity limit, from clients located in its subtree. In addition to these *server capacity* constraints, we consider that the *distance* between a client and the server processing some of its requests cannot exceed $d_{\max}$. A weight assigned to each edge of the tree represents the inter-node distance, it may for instance correspond to a communication cost, or quality of service (QoS) requirements [6], [7]. Hence, the requests must be served in limited time, thereby prohibiting too remote or hard-to-reach replica locations. Moreover, we consider two policies: the *Single* policy enforces that all requests of a client are served by a single server in the tree, while in the *Multiple* policy, the requests of a given client can be processed by multiple servers, thus distributing the processing of requests over the platform.

Many authors deal with variants of these replica placement problems in networks. Some variants of server location problems can be found in [8], [9], [10], [11]. In most of these problems, a set of users in a network want to have access to a given service. The aim is then to identify a set of service providers able to offer a sufficient amount of resources in order to satisfy the requests of the clients, where servers are subject to capacity constraints. In some variants, a quality of service must be guaranteed, in terms of latencies to process the requests. Hence, a smart repartition of the servers in the network may enable to minimize the latencies between any client and its associated server, and also to ensure good fault tolerance properties.

In general graphs, and when no distance constraints are imposed, those problems are very similar to Bin-Packing, a classical optimization problem [12] for which an APTAS (Asymptotic Polynomial Time Approximation Scheme) is known [13]. The server capacities correspond to the size of the bins. The three main features of our problem that differ with Bin-Packing are (i) the fact that bins are associated to servers, and must be placed in the network; (ii) the tree network, which imposes that a server can process only clients in its subtree; and (iii) the distance constraints that we enforce.

The Bin-Packing problem with distance constraints has been studied in [14], where groups of clients are built. The sum of the requests inside a group should not exceed a fixed capacity, and a maximal distance is fixed between two clients of a same group. The aim is therefore to build a minimum of groups (i.e., to use a minimum number of servers) so that each client belongs to one group. The difference with our problem is that the distance constraint is on the diameter of each group, whereas in our case, a server has to be chosen, and the distance constraint is between the server and its clients.

More generally, when tackling the replica placement problem in general graphs, the aim is usually first to extract a "good" spanning tree, i.e., a spanning tree that will optimize some global objective function, and then to place replicas along the spanning tree, typically in order to optimize a more refined function. However, the process of extracting a spanning tree is of combinatorial nature, as it generalizes the well-known NP-hard K-center problem [15]. Therefore, several authors propose sophisticated heuristics whose goal is to solve both steps simultaneously (see [16] for a survey). In this context, some approximation algorithms are provided in [17], [18] for the uniform weights capacitated K-center problem.

In [19], the authors investigate hierarchical bin packing, and prove some approximation results. Even though this is the closest work to ours that we could find, the problem is different since servers do not need to be physically placed in the tree, but rather the objective is to minimize the dispersal of the bins in the tree. Moreover, they do not consider any distance constraints.

The first contributions of this paper are targeting problems with the *Single* policy. The simplest problem instance, with no distance constraints on a binary tree, turns out to be NP-hard in the strong sense, and therefore we establish approximation results. To the best of our knowledge, this is the first attempt to derive approximation algorithms for this replica placement problem on tree networks. We propose a $(\Delta + 1)$-approximation algorithm for this problem with the *Single* policy, where Δ is the arity of the tree. For the problem without distance constraints, we design a 2-approximation algorithm that works for general trees, and whose approximation ratio is independent of Δ.

Concerning problems with the *Multiple* policy, we establish several new complexity results. While it is already known that the general problem is NP-hard, and that it can be solved in polynomial time without distance constraints (see [3]), we prove that, surprisingly, this problem with distance constraints in the specific case of a binary tree can also be solved in polynomial time, by exhibiting an involved polynomial time algorithm. Note that this result holds only when all requests of a client can always entirely be served locally. Otherwise, we prove that the problem remains NP-hard.

II. FRAMEWORK

This section is devoted to a precise statement of the optimization problems that we study in this paper.

We consider a distribution tree $\mathcal{T} = \mathcal{C} \cup \mathcal{N}$. The set of internal nodes is $\mathcal{N}$ and the set of leaf nodes is $\mathcal{C}$. The root of the tree is denoted by r. For $j \in \mathcal{C} \cup \mathcal{N} \setminus \{r\}$, $\mathit{parent}(j) \in \mathcal{N}$ is the parent of node j in the tree. For $j \in \mathcal{N}$, $\mathit{children}(j) \subset \mathcal{C} \cup \mathcal{N}$ is the set of children of node j in the tree, and $\mathit{subtree}(j) \subseteq \mathcal{C} \cup \mathcal{N}$ is the subtree rooted in j, including j. Δ is the arity of the tree. Moreover, for each node $j \in \mathcal{C} \cup \mathcal{N} \setminus \{r\}$, δ_j is the *distance* from node j to $\mathit{parent}(j)$: it can be seen as a weight assigned to each edge of the tree, and it can correspond for instance to a communication cost. At the root of the tree, we set $\delta_r = +\infty$.

Each node $i \in \mathcal{C}$ is sending r_i requests per time unit to a database object (note that the number of requests is usually assumed to be integer), and the distance between node i and any node processing some of these r_i requests cannot exceed $d_{\max}$. A node $j \in \mathcal{C} \cup \mathcal{N}$ may or may not have been provided with a replica of the database. If node j has been equipped with a replica (i.e., it is a server), then it can process requests from any node i in its subtree, given that the distance between node i and node j is not greater than $d_{\max}$. Note that it cannot process requests from a node that is not in its subtree, as for instance its parent node. In other words, there is a unique path from a client node i to the root r of the tree: $i = i_1 \to i_2 \to \cdots \to i_k = r$, and each node i_ℓ in this path (with $1 \leq \ell \leq k$) is eligible to process some or all the requests issued by i when provided with a replica, given that $\sum_{1 \leq \ell' < \ell} \delta_{i_{\ell'}} \leq d_{\max}$.

For each client $i \in \mathcal{C}$, let $\mathit{servers}(i)$ be the set of servers responsible for processing at least one of its requests. There are two scenarios for the number of servers assigned to each client. With the *Single* policy, each client i is assigned a single server that is responsible for processing all its requests, and $|\mathit{servers}(i)| = 1$. With the *Multiple* policy, a client i may be assigned several servers, and we let $r_{i,s}$ be the number of requests from client i processed by server s. All requests must be processed, thus $\sum_{s \in \mathit{servers}(i)} r_{i,s} = r_i$. In the *Single* case, a unique server s_i is handling all r_i requests, and $r_{i,s_i} = r_i$.

Let $\mathcal{R}$ be the set of replicas: $\mathcal{R} = \{s \in \mathcal{C} \cup \mathcal{N} \mid \exists i \in \mathcal{C}\ ,\ s \in \mathit{servers}(i)\}$. The processing capacity of each node is fixed to W, which is the total number of requests that it can process per time unit when it has been assigned a replica. In addition to the distance constraints, the constraints on server capacities must be fulfilled: $\forall s \in \mathcal{R},\ \sum_{i \in \mathcal{C} | s \in \mathit{servers}(i)} r_{i,s} \leq W$.

Finally, the objective function is to minimize the number of replicas that are placed, i.e., minimize $|\mathcal{R}|$.

We are now ready to formally define the various instances of the problem. The objective is to minimize the number of replicas, while ensuring that both server capacities and distance constraints are enforced, either with the *Single* or with the *Multiple* policy. The problem names are respectively SINGLE or MULTIPLE. For the distance constraints, we consider the particular case with no constraint (NOD). Hence, SINGLE-NOD (resp., MULTIPLE-NOD) is the problem with the single server (resp. multiple servers) policy and no distance constraints. Finally, we give a particular attention to instances in which the distribution tree is binary, i.e., $\Delta = 2$. For instance, the MULTIPLE problem on binary trees is denoted MULTIPLE-BIN, and the SINGLE problem with no distance constraints on binary trees is denoted SINGLE-NOD-BIN.

III. SINGLE POLICY

In this section, we first prove that SINGLE-NOD-BIN is NP-hard in the strong sense, and therefore all problem instances with the *Single* policy are NP-hard, since the reduction is done for the simplest problem instance (see Theorem 1). Then, we prove that for all $\varepsilon > 0$, there is no $(\frac{3}{2} - \varepsilon)$-approximation algorithm for this problem, unless P=NP (see Theorem 2). We provide a $(\Delta + 1)$-approximation algorithm for the SINGLE problem in general trees, where Δ is the arity of the tree (see Section III-C). This algorithm is a Δ-approximation algorithm when there are no distance constraints (SINGLE-NOD). Then we refine the previous algorithm in the case where there are no distance constraints to obtain a 2-approximation algorithm, hence getting closer to the bound of Theorem 2 (see Section III-D).

Note that we assume that for all $i \in \mathcal{C}$, $r_i \leq W$, otherwise there is no solution. Also, the solution with $servers(i) = \{i\}$ for all $i \in \mathcal{C}$, and hence $\mathcal{R} = \mathcal{C}$, is always a valid solution, in which no distance nor capacity constraints are exceeded. However, our goal is to exploit the server nodes of $\mathcal{N}$ to reduce the number of replicas.

A. NP-completeness result of SINGLE-NOD-BIN

Theorem 1: SINGLE-NOD-BIN is NP-hard in the strong sense.

Proof: We consider the associated decision problem: given an integer K, is there a solution with no more than K servers? The problem is clearly in NP: given a set of servers and for each server, the set of all requests handled by the server, it is easy to check in polynomial time if the server capacities are not exceeded.

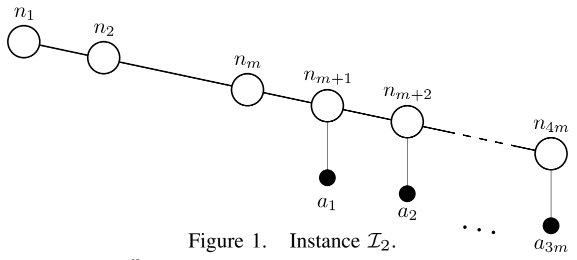

Figure 1. Instance $\mathcal{I}_2$.

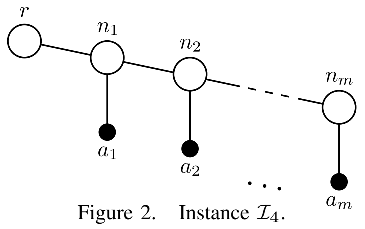

Figure 2. Instance $\mathcal{I}_4$.

To establish the completeness, we use a reduction from 3-PARTITION. We consider an instance $\mathcal{I}_1$ of 3-PARTITION [15]: given $3m + 1$ positive integers $a_1, a_2, \ldots, a_{3m}$ and B such that $B/4 < a_i < B/2$ for $i \in \{1, \ldots, 3m\}$ and $\sum_{i=1}^{3m} a_i = mB$, can we partition these integers into m triples, each of sum B? We build the instance $\mathcal{I}_2$ of SINGLE-NOD-BIN depicted in Fig. 1, and we let c_i be the client with a_i requests. Finally we ask whether there exists a solution with $K = m$ replicas of capacity $W = B$. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$.

We now show that $\mathcal{I}_2$ has a solution if and only if $\mathcal{I}_1$ does. Suppose first that $\mathcal{I}_1$ has a solution. Let then $(a_{k_1}, a_{k_2}, a_{k_3})$ the k^{th} triple in $\mathcal{I}_1$, for $1 \leq k \leq m$. We place a server at node n_k, which is processing requests from c_{k_1}, c_{k_2} and c_{k_3}. Clearly, we have m servers, no server capacity is exceeded, and all requests are handled, thus $\mathcal{I}_2$ has a solution.

Suppose now that $\mathcal{I}_2$ has a solution. There are at most m servers of capacity B, and the sum of all requests is equal to mB, therefore exactly m replicas are set, and each of them handles a sum B of requests. Since $B/4 < a_i < B/2$ for $i \in \{1, \ldots, 3m\}$, a server cannot handle neither more than three requests, nor less than three requests. We conclude that $\mathcal{I}_1$ has a solution. ■

B. Inapproximability result with the Single policy

Theorem 2: Unless P=NP, for all $\varepsilon > 0$, there is no $(\frac{3}{2} - \varepsilon)$-approximation algorithm for SINGLE-NOD-BIN.

Proof: Let us assume that there exists $\varepsilon > 0$ such that there is a $(3/2 - \varepsilon)$-approximation to SINGLE-NOD-BIN. We denote by $algo$ this polynomial time algorithm. We prove that this algorithm allows us to solve 2-PARTITION in polynomial time, and since 2-PARTITION is NP-complete [15], this proves that P=NP.

We consider an instance $\mathcal{I}_3$ of 2-PARTITION: given m positive integers $a_1, a_2, \ldots, a_m$, does there exist a subset I of $\{1,\dots,m\}$ such that $\sum_{i\in I} a_i = \sum_{i\notin I} a_i$? Let $S = \sum_{i=1}^{m} a_i$. We build the instance $\mathcal{I}_4$ of SINGLE-NOD-BIN, see Fig. 2, where the server capacity is $W = S/2$. Note that if $\mathcal{I}_3$ has a solution, then there is a solution to $\mathcal{I}_4$ with two replicas, that can be placed at nodes r and n_1.

Then, we use $algo$ to solve $\mathcal{I}_4$ in polynomial time. If the solution returns two servers ($algo(\mathcal{I}_4) = 2$), then we have a solution to $\mathcal{I}_3$, since the solution is necessarily a 2-partition of the a_i. Otherwise, the solution returns at least three servers ($algo(\mathcal{I}_4) \geq 3$), and since it is a $(3/2-\varepsilon)$-approximation algorithm, if $opt(\mathcal{I}_4)$ is the optimal solution for this instance, it means that $3 \leq algo(\mathcal{I}_4) < \frac{3}{2} opt(\mathcal{I}_4)$, and therefore $opt(\mathcal{I}_4) > 2$, which means that there is no solution to $\mathcal{I}_3$. Therefore, this $(3/2-\varepsilon)$-approximation algorithm allows us to solve $\mathcal{I}_3$ in polynomial time, which concludes the proof. ■

C. Approximation algorithm with distance constraints

First we propose a polynomial time algorithm, named **single-gen**, to solve the SINGLE problem (see Algorithm 1). The solution is obtained by a call to **single-gen**(r), where r is the root of the tree. Initially, the set of servers is empty ($\mathcal{R} = \emptyset$), and the procedure greedily adds servers to this set. It works recursively: **single-gen**(j) performs one call to **single-gen**(j') for each $j' \in children(j)$, and then decides where to place servers. The procedure returns a couple $(req, dist)$, where $req \leq W$ is the number of requests that still need to be processed at or above node j in the tree, and these requests have to be served at a maximum distance $dist$ from node j. The algorithm always returns **single-gen**$(r) = (0, d_{\max})$, i.e., all requests are processed by the servers that have been placed.

The call to **single-gen**(i), where $i \in \mathcal{C}$ is a leaf node of the tree, returns the result $(r_i, d_{\max})$. For any other node $j \in \mathcal{N}$, we recursively call the procedure on each child node $j' \in children(j)$, and collect the corresponding couples $(req_{j'}, dist_{j'})$. Then several cases occur.

1) First, we check whether the requests of a node $j' \in children(j)$ can be processed at node j or above. If they cannot, i.e., $\delta_{j'} > dist_{j'}$, we add node j' to the set of replicas: $\mathcal{R} = \mathcal{R} \cup \{j'\}$, and we set $req_{j'} = 0$ and $dist_{j'} = d_{\max}$ (i.e., no more requests are arriving to node j from node j'). Otherwise, we update the distance $dist_{j'} = dist_{j'} - \delta_{j'}$.

2) If $\sum_{j'\in children(j)} req_{j'} > W$, then we place a server on each child node of j that has at least one request: $\mathcal{R} = \mathcal{R} \cup \{j' \in children(j) \mid req_{j'} > 0\}$. Therefore, no requests are going up in the tree, and the procedure returns $(0, d_{\max})$.

Algorithm 1: (Δ+1)-approx. algorithm for SINGLE.

```
procedure single-gen($j$)
begin
    if $j \in \mathcal{C}$ then
        return $(r_j, d_{\max})$;
    else
        for $j' \in children(j)$ do
            $(req_{j'}, dist_{j'}) =$ single-gen($j'$);
            if $\delta_{j'} > dist_{j'}$ and $req_{j'} > 0$ then
                $\mathcal{R} = \mathcal{R} \cup \{j'\}$;
                $req_{j'} = 0$; $dist_{j'} = d_{\max}$;
            else $dist_{j'} = dist_{j'} - \delta_{j'}$;
        if $\sum_{j'\in children(j)} req_{j'} > W$ then
            for $j' \in children(j)$ do
                if $req_{j'} > 0$ then $\mathcal{R} = \mathcal{R} \cup \{j'\}$;
            return $(0, d_{\max})$;
        else
            if $j = r$ then
                if $\sum_{j'\in children(j)} req_{j'} > 0$ then
                    $\mathcal{R} = \mathcal{R} \cup \{r\}$;
                return $(0, d_{\max})$;
            else return $\Big(\sum_{j'\in children(j)} req_{j'},$
                         $\min_{j'\in children(j)} dist_{j'}\Big)$;
```

3) Otherwise, $\sum_{j'\in children(j)} req_{j'} \leq W$, and we operate differently, depending on whether j is the root of the tree or not.

a) If $j = r$, (root of the tree), we place a server if needed (there is at least one request to process), and the procedure returns $(0, d_{\max})$:
if $\sum_{j'\in children(r)} req_{j'} > 0$, then $\mathcal{R} = \mathcal{R} \cup \{r\}$.

b) If $j \neq r$, there are $\sum_{j'\in children(j)} req_{j'} \leq W$ requests that can be processed at node j or above, and the maximum distance for these requests is $\min_{j'\in children(j)} dist_{j'}$. Therefore, the procedure returns

$$\Big(\sum_{j'\in children(j)} req_{j'}, \min_{j'\in children(j)} dist_{j'}\Big).$$

Theorem 3: The call to **single-gen**(r) is a $(\Delta+1)$-approximation algorithm for SINGLE, and its time complexity is $O(\Delta \times |\mathcal{T}|)$.

Proof: Consider an instance of SINGLE. Let $\mathcal{R}_{opt}$ be the set of servers in an optimal solution, and $\mathcal{R}_{algo}$ is the set of servers returned by the call to **single-gen**(r) (see Algorithm 1). The call to **single-gen**(j) may add some children of j into the set of replicas, but it never adds j to $\mathcal{R}_{algo}$ (except for the root node). Hence, it is always possible, when going up in the tree, to add a replica to a children node, since it is not yet in the set of replicas. Also, the procedure is such that the number of requests going up in the tree is never greater than W, therefore it is always possible, by adding a replica to a child node, to cover all requests in its subtree, and therefore the algorithm always succeeds.

Let $\mathcal{R}_1$ be the set of replicas that are added by the algorithm either at step 1 (line 9 of Algorithm 1) or at step 3a (line 19), while $\mathcal{R}_2$ is the set of replicas that are added at step 2 (line 14). Note that $\mathcal{R}_{algo} = \mathcal{R}_1 \cup \mathcal{R}_2$, and we provide upper bounds on the cardinality of these two sets.

Let $j \in \mathcal{R}_1$ be such that there is no other server in $\mathcal{R}_1$ in the subtree rooted in j, $subtree(j)$. This server is processing some requests that cannot be processed upper in the tree. To process such requests, the optimal solution also needs to place at least one server in $subtree(j)$. Since the algorithm does not let any request from the subtree traverse j, we consider the tree $\mathcal{T} \setminus subtree(j)$, and use the same argument recursively, to prove that $|\mathcal{R}_1| \leq |\mathcal{R}_{opt}|$.

If some servers are added in $\mathcal{R}_2$, it means that at some point, a node j had strictly more than W requests in its subtree, and we add at most Δ servers to process all these requests. Since a server cannot handle more than W requests, the optimal solution must place at least one replica for each of such groups of requests, and therefore $|\mathcal{R}_2| \leq \Delta \times |\mathcal{R}_{opt}|$. Finally,

$$\begin{aligned} |\mathcal{R}_{algo}| = |\mathcal{R}_1| + |\mathcal{R}_2| &\leq |\mathcal{R}_{opt}| + \Delta \times |\mathcal{R}_{opt}| \\ &\leq (\Delta+1)|\mathcal{R}_{opt}| , \end{aligned}$$

which concludes the proof.

Note that the algorithm performs exactly $|\mathcal{N} \cup \mathcal{C}|$ calls to **single-gen**, and that all operations performed in **single-gen** can be done in time $O(\Delta)$, and therefore the algorithm has a time complexity in $O(\Delta \times |\mathcal{T}|)$, which is clearly polynomial in the problem size. ■

We now show that this $\Delta + 1$ factor cannot be improved; in other words, we prove that there does not exist $\varepsilon > 0$ such that Algorithm 1 is a $(\Delta + 1 - \varepsilon)$-approximation. Consider the instance $\mathcal{I}_m$ depicted on Fig. 3 (n_0 is the root and the tree is laid on its side). $\mathcal{I}_m$ is built by the concatenation of $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_m$. We set $d_{\max} = 4m$; in addition, all distances are set to 1, except the distance between $c_{i,\Delta}$ and $n_{i,1}$, for $1 \leq i \leq m$, which is equal to $d_{\max}$. Therefore, all requests from $c_{i,\Delta}$ must be processed either locally by $c_{i,\Delta}$, or by its parent node $n_{i,1}$, while all other requests can be processed anywhere on the path from the node issuing the requests to the root n_0 of the tree. The number of

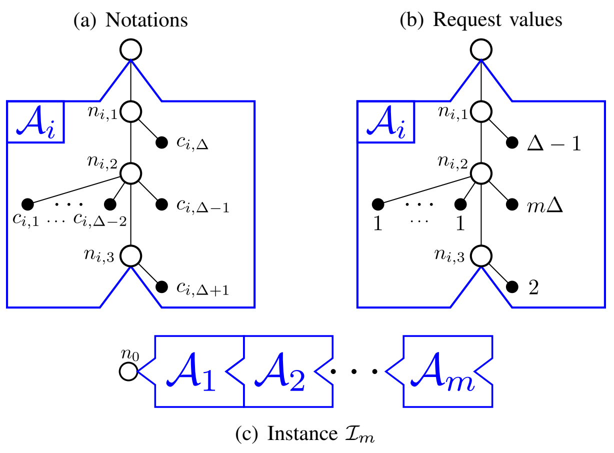

Figure 3. An instance on which Algorithm 1 reaches an approximation ratio of $\Delta + 1$.

requests of each node $c_{i,j}$ in $\mathcal{C}$, for $1 \leq i \leq m$ and $1 \leq j \leq \Delta + 1$, are given in Fig. 3. Finally, we fix $W = m\Delta + \Delta - 1$.

The first servers that are placed by Algorithm 1 correspond to the call to **single-gen**($n_{m,2}$), because the sum of requests of its children nodes is $m\Delta + (\Delta - 2) \times 1 + 2 = m\Delta + \Delta > W$. Therefore, the algorithm adds Δ servers to $\mathcal{R}$, at nodes $c_{m,1}, \dots, c_{m,\Delta-1}$ and $n_{m,3}$. Then, a server is placed on $n_{m,1}$, because of the distance constraint, and there are no requests going up to $\mathcal{A}_{m-1}$. We can therefore reiterate on $\mathcal{A}_{m-1}, \dots, \mathcal{A}_1$, and the algorithm adds $\Delta + 1$ servers at each step. Overall, the algorithm places $m \times (\Delta + 1)$ servers: $\mathcal{R}_{algo} = \{c_{i,j},\ n_{i,1},\ n_{i,3}\}_{1 \leq i \leq m,\ 1 \leq j \leq \Delta - 1}$.

Since the sum $m \times (m\Delta + 2\Delta - 1)$ of all requests is strictly greater than $m \times W = m \times (m\Delta + \Delta - 1)$, we must place at least $m + 1$ servers to handle all the requests. Let $\mathcal{R}_{opt} = \{n_0,\ n_{i,1}\}_{1 \leq i \leq m}$ be a set of $m+1$ servers. Node $n_{i,1}$ is processing the W requests of clients $c_{i,\Delta}$ and $c_{i,\Delta-1}$, for $1 \leq i \leq m$, while the root is processing the requests of $c_{i,1}, \dots, c_{i,\Delta-2}$ and $c_{i,\Delta+1}$, for $1 \leq i \leq m$, hence a total of $m\Delta \leq W$ requests. This is an optimal solution since it involves $m + 1$ servers.

On this instance, the ratio between the solution found by the Algorithm 1 and the optimal solution is

$$ratio^{(m)}_{\textbf{single-gen}} = \frac{m \times (\Delta + 1)}{m + 1} \underset{m \to +\infty}{\longrightarrow} \Delta + 1.$$

This shows that the approximation factor of Algorithm 1 cannot be improved.

Corollary 1: Algorithm 1 is a Δ-approximation algorithm without distance constraints (SINGLE-NOD).

Proof: The previous algorithm can be substantially simplified when there are no distance constraints, since the condition on line 8 is never satisfied, and hence $\mathcal{R}_1$

is either the empty set or it contains only the root of the tree. A set of at most Δ replicas is added each time a node j has strictly more than W requests in its subtree, and we process all these requests. Since the inequality is strict, $|\mathcal{R}_2| < \Delta \times |\mathcal{R}_{opt}|$, and $\mathcal{R}_{algo} \leq 1 + |\mathcal{R}_2| \leq \Delta \times |\mathcal{R}_{opt}|$, which concludes the proof. ■

D. Approx. algorithm without distance constraints

Here we propose a polynomial time algorithm, named **single-nod**, to solve the SINGLE-NOD problem. The solution is obtained by a call to **single-nod**(r), where r is the root of the tree. Initially, the set of servers is empty ($\mathcal{R} = \emptyset$), and the procedure greedily adds servers to this set. The procedure **single-nod**(j) returns a value req corresponding to the number of requests that still need to be processed at or above node j in the tree. The algorithm always returns **single-nod**$(r) = 0$, i.e., all requests are processed by the servers that have been placed.

The call to **single-nod**(i), where $i \in \mathcal{C}$ is a leaf node of the tree, returns the result r_i. For any other node $j \in \mathcal{N}$, we recursively call the procedure on each child node $j' \in children(j)$, and collect the corresponding values $req_{j'}$. Also, we change the structure of the tree during the procedure, and hence we keep an updated list of children for each node j, denoted by C_j, and initially $C_j = children(j)$. We may then add new children to a node. Then several cases occur.

1) If $\sum_{j' \in C_j} req_{j'} > W$, then we place a server on node j. We sort the nodes of C_j by non-decreasing number of requests, and we greedily assign requests to server j (starting with the smallest requests), while the total capacity W is not exceeded. We also add a server at node $j_{\min}$, where $j_{\min}$ is the first node of C_j whose requests could not be processed by j. The procedure returns 0, and we consider two cases:

a) if j is not the root of the tree, and if some requests of C_j have not yet been handled by the two servers j and $j_{\min}$, we add the corresponding nodes to $C_{parent(j)}$;
b) otherwise, if j is the root r of the tree, we add all nodes of C_r whose requests are not yet processed in the set of servers $\mathcal{R}$.

2) Otherwise, $\sum_{j' \in C_j} req_{j'} \leq W$, and those requests can be processed either at node j or upper in the tree. Therefore,

a) if j is not the root of the tree, the procedure returns $\sum_{j' \in C_j} req_{j'}$;
b) otherwise, if $j = r$, we add r in $\mathcal{R}$ and it can handle all remaining requests; the procedure returns 0.

Note that the pseudo-code for this algorithm can be found in the companion research report [20].

Theorem 4: The call to **single-nod**(r) is a 2-approximation algorithm for SINGLE-NOD, and its time complexity is $O((\Delta \log \Delta + |\mathcal{C}|) \times |\mathcal{T}|)$.

Proof: Consider an instance of SINGLE-NOD. Let $\mathcal{R}_{algo}$ be the set of servers returned by the call to **single-nod**(r). We denote by $\mathcal{R}_1$ the set of servers j added at step 1 of the algorithm; $\mathcal{R}_2$ is the set of servers added as extra servers ($j_{\min}$): to each $j \in \mathcal{R}_1$, we can associate $j' \in \mathcal{R}_2$ and $proc(j) + proc(j') > W$, where $proc(j)$ is the number of requests processed by a server $j \in \mathcal{R}_{algo}$ (with the allocation done by the algorithm). Note that $|\mathcal{R}_1| = |\mathcal{R}_2|$. Finally, $\mathcal{R}_3$ is the set of servers that are eventually added at step 1b.

If $\mathcal{R}_3 = \emptyset$ and we finished the procedure at step 2b, then we have $\mathcal{R}_{algo} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{r\}$, and hence $|\mathcal{R}_{algo}| = 2 \times |\mathcal{R}_1| + 1$. Moreover, the total number of requests in the tree is strictly greater than $|\mathcal{R}_1| \times W$ by construction, and therefore the optimal solution needs at least $|\mathcal{R}_1| + 1$ servers, hence the 2-approximation.

Otherwise, we have $\mathcal{R}_{algo} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$ (note that $r \in \mathcal{R}_1$ in this case), and we aim at proving that any solution must use at least $|\mathcal{R}_1| + |\mathcal{R}_3|$ servers. Our aim is to provide a lower bound on the number of servers that must be added in a subtree to cover all requests, by building a *relaxed* set of servers that can process all requests in the tree. First, we formally define the lower bound, and we establish a few properties. Note that if a client has no requests ($r_i = 0$), we suppress it from the tree, as well as any internal node that becomes a leaf of the tree.

Definition 1. *For all $j \in \mathcal{N}$, $nb(j)$ is a lower bound on the number of nodes with requests that cannot be grouped together in $subtree(j)$ (and hence, either they will be grouped with other nodes higher in the tree, or it will be necessary to cover each of these nodes with a server).*

Property 1. *For $j \in \mathcal{N}$, if there are less than W requests in $subtree(j)$, i.e., $0 < \sum_{i \in subtree(j) \cap \mathcal{C}} r_i \leq W$, then there is an optimal solution with no replica in $subtree(j) \setminus \{j\}$, and $nb(j) = 0$.*

If there was a replica in $subtree(j) \setminus \{j\}$, we could always move it to node j, and this replica would process the whole subtree, hence letting no requests go up in the tree. All requests can be grouped at node j, and therefore $nb(j) = 0$. In this case, we *aggregate* $subtree(j)$ at node j, hence node j becomes a client node: $j \in \mathcal{C}'$, where $\mathcal{C}'$ is the set of clients generated by aggregating requests. For such nodes, we define $r_j = \sum_{i \in subtree(j) \cap \mathcal{C}} r_i$. We prove later that aggregation can always be done in the relaxed solution.

Property 2. *Let us consider $j \in \mathcal{N}$, such that $\forall i \in$*

children(j)*,* $i \in \mathcal{C} \cup \mathcal{C}'$*, and* $\sum_{i \in children(j)} r_i > W$*. Let* $i_1, \dots, i_{n_j}$ *be the* n_j *children of* j*, ordered by non-decreasing values of* r_{i_k} *(i.e.,* $0 < r_{i_1} \leq r_{i_2} \leq \dots \leq r_{i_{n_j}} \leq W$*). Let* $m_j \geq 2$ *be the index such that* $\sum_{k=1}^{m_j - 1} r_{i_k} \leq W$ *and* $\sum_{k=1}^{m_j} r_{i_k} > W$*. Then* $nb(j) = n_j - m_j + 1$*.*

In this case, requests can only be covered together by a server if node j is this server, and therefore the aim is to cover as many clients as possible with a server at node j. This is done by covering the clients with the least number of requests, hence clients $i_1, \dots, i_{m_j - 1}$. There remain $n_j - m_j + 1$ nodes to be covered.

Note that thanks to Property 1, even if a child node i is an aggregated node ($i \in \mathcal{C}'$), then we do not benefit of covering only a subset of the requests in $subtree(i)$. Indeed, if node j is covering only some of the requests of $subtree(i)$ (but not all of them), then there is at least one node i' in $subtree(i)$ that cannot be grouped in $subtree(j)$. We can rather assume that no request from $subtree(i)$ is covered by j, hence decreasing the load at node j and keeping the same (or even decreasing) number of nodes that remain to be grouped higher in the tree (by replacing i', and eventually other nodes in $subtree(i)$ that were not covered, with i).

Next we need to extend this property recursively when going up in the tree.

Property 3. *Let us consider a node* j *that has only children satisfying Property 1 (children in* P_1*) or Property 2 (children in* P_2*):* children$(j) = P_1 \cup P_2$*. Let* $C_j = P_1$ *be the set of client nodes of* j*. For all* $j' \in P_2$*, we add nodes* $m_{j'} + 1$ *to* $n_{j'}$ *(as defined in Property 2) to the set* C_j*, and we order the* n_j *nodes in* C_j *by non-decreasing values of* r_{i_k}*. If* $\sum_{k=1}^{n_j} r_{i_k} \leq W$ *(***Property 3a***), then* $nb(j) = 0$*. Otherwise (***Property 3b***), let* $m_j \geq 2$ *be the index such that* $\sum_{k=1}^{m_j - 1} r_{i_k} \leq W$ *and* $\sum_{k=1}^{m_j} r_{i_k} > W$*. Then* $nb(j) = n_j - m_j + 1$ *.*

Property 3a comes directly from the fact that all requests can be grouped at node j, similarly to Property 1.

Now we consider that we are in the case of Property 3b. Requests can be grouped in $subtree(j)$ by placing servers at each node $j' \in P_2$, and a server at node j. First consider a node $j' \in P_2$. By definition of P_2, there are strictly more than W requests coming from j', and hence potentially W of these requests could be covered by a single server placed at node j'. Let X be the set of children of node j': $X = \{i_1, \dots, i_{n_{j'}}\}$, where the children are ordered by non-decreasing number of requests as before. Consider that the set $X_1 \subseteq X$ of children is covered by j', and that the set $X_2 \subseteq X$ is covered by j. There remain $|X \setminus (X_1 \cup X_2)|$ servers in $subtree(j')$ that cannot be grouped in $subtree(j)$. First note that if $X_1 \cup X_2$ does not contain the smallest children of X, i.e., there is $i_a \in X_1 \cup X_2$ and $i_b \notin X_1 \cup X_2$, with $b < a$, then we can exchange i_a and i_b: we cover i_a instead of i_b, hence decreasing the amount of requests to be handled by j or j', and keeping identical the number of servers requested to cover the subtree. Next, if X_1 does not contain the smallest children of $X_1 \cup X_2$, we change the assignment to process children 1 to $m_{j'}$ with server j' (this is actually not possible since these children have a total of strictly more than W requests, but we perform *relaxed* grouping so as to obtain a lower bound on the remaining nodes that cannot be grouped). Since the sum of the other requests from $X_1 \cup X_2$ do not exceed the sum of the requests initially in X_2, j can cover these requests together with the requests that are coming from other children of j (even if these requests may be individually larger than the previous requests assigned to j).

We still need to discuss if it would not be beneficial to cover individually some requests that have been aggregated in our reasoning. Let us consider an optimal solution (that may not have necessarily aggregated nodes as we are doing with Property 1 and 3a). Then the sum of the requests that can be grouped at node j' is less or equal to W (since we are not considering a relaxed solution anymore, but a valid solution). Even if this optimal solution is covering only some of the requests of $subtree(i)$, where i is a children node of j' and $i \in P_1$, the number of nodes that cannot be grouped in $subtree(j')$ cannot be lower than $nb(j')$. Moreover, the sum of the requests is greater than the sum of the requests that comes from the relaxed solution. Since we cover always more than W requests with the relaxed solution, any optimal solution cannot perform more grouping, even by exploiting smaller requests that have not been aggregated.

If we proceed similarly with all children $j' \in P_2$, we decrease the amount of requests to be processed by j, hence computing a lower bound on the number of nodes with requests that cannot be grouped in $subtree(j)$. Similarly to the proof of Property 2, it is then easy to see that j should cover the smallest remaining children in order to minimize this number, hence the result.

Building a *relaxed* replica set $\mathcal{R}_{rel}$. Thanks to the properties that have just been presented, we build an equivalent tree corresponding to a *relaxed* solution where there is no problem to group requests, since servers can process more than W requests. Indeed, as we have said at the beginning of the proof, it is easy to see that the algorithm is a 2-approximation if $\mathcal{R}_3 = \emptyset$, but otherwise the algorithm has failed to group requests in $\mathcal{R}_3$, and we need to prove that any solution could not have done better.

Initially, $\mathcal{R}_{rel} = \emptyset$, and we build recursively the tree $\mathcal{T}'$ of requests that cannot be processed by a common node, starting with $\mathcal{T}' = \mathcal{T}$. First we replace $subtree(j)$ by j on nodes following Property 1, i.e., we aggregate these nodes. Then, if a node j follows Property 2, following the insight of Property 3, the relaxed solution groups the m_j smallest children of j and processes them at node j, hence we let $\mathcal{R}_{rel} = \mathcal{R}_{rel} \cup \{j\}$ and we remove node j and the processed children from $\mathcal{T}'$. The remaining children of node j are then attached to $parent(j)$, which is the next node where these children may be potentially grouped together. We go up in the tree until we reach the root r. Note that when we encounter a node j following Property 3b, we have already added the $|P_2|$ children of j to $\mathcal{R}_{rel}$ at a previous step, and then we keep only a single node j with the sum of all requests that are remaining in its subtree (similarly to the aggregation performed with Property 1).

At the end, since the algorithm follows the construction of $\mathcal{R}_{rel}$, we have $\mathcal{R}_{rel} = \mathcal{R}_1$, and the difference is that the algorithm has added two replicas instead of one at each node j following Property 2, in order to cover the same amount of requests (strictly greater than W). Note that when going up in the tree recursively, each node will either follow Property 1 (identical to Property 3a) or Property 2 (identical to Property 3b).

Even in the relaxed solution that groups requests in the best possible way, there may remain some requests to cover in $\mathcal{T}'$, and the nodes that are in $\mathcal{T}'$ at the end of the tree transformation are by construction the nodes of $\mathcal{R}_3$ of the algorithm. Therefore, any solution must use at least $|\mathcal{R}_{rel}| + |\mathcal{R}_3|$ replicas, and finally

$$|\mathcal{R}_{opt}| \geq |\mathcal{R}_{rel}| + |\mathcal{R}_3| = |\mathcal{R}_1| + |\mathcal{R}_3| \geq \frac{1}{2}|\mathcal{R}_{algo}|,$$

which concludes the proof.

The algorithm performs exactly $\mathcal{C} \cup \mathcal{N}$ calls to **single-nod**, and all operations performed in **single-nod** can be done in time $O(\Delta \log \Delta + |\mathcal{C}|)$: in the worst case, there are Δ children nodes whose requests need to be sorted (in $O(\Delta \log \Delta)$). We do not sort C_j, but rather keep sorted lists and merge two sorted lists when moving nodes in the tree. The cost of the merge procedure can be in $O(|\mathcal{C}|)$ at each call of **single-nod**. All together, the time complexity of the algorithm is $O((\Delta \log \Delta + |\mathcal{C}|) \times |\mathcal{T}|)$. ■

We now show that this factor of 2 cannot be improved. Consider the instance depicted on Fig. 4, with $W = K$. Nodes $n_1, \ldots, n_K$ are satisfying Property 2

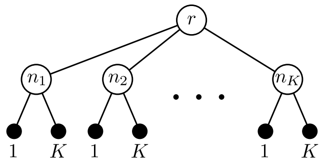

Figure 4. An instance on which **single-nod** reaches an approximation ratio of 2.

and hence they are added by the algorithm in set $\mathcal{R}_1$. Their client node with K requests are added in set $\mathcal{R}_2$. Therefore, $|\mathcal{R}_{algo}| = 2K$. However, the optimal solution processes exactly K requests at each node n_i (with $1 \leq i \leq K$), and places one extra server at the root that can process the requests of all K clients with only one request, hence $|\mathcal{R}_{opt}| = K + 1$, and the ratio of 2.

IV. Multiple policy

In this section, we target the problems with the *Multiple* policy. While it is already known that Multiple-NoD can be solved in polynomial time and that Multiple is NP-hard (see [3]), we prove that, surprisingly, Multiple-Bin can also be solved in polynomial time, by exhibiting an involved polynomial time algorithm. Note that this result holds only when all the r_i's are smaller or equal to W, i.e., all the requests of a client $i \in \mathcal{C}$ can always be served locally by adding a replica at node i. Otherwise, we prove that the problem remains NP-hard.

A. *NP-completeness of* Multiple-Bin

Theorem 5: Multiple-Bin is NP-hard.

Proof sketch: Due to space limitation, we give only a sketch of the proof, and the detailed proof can be found in the companion research report [20]. The completeness comes from a reduction from 2-PARTITION-EQUAL [15]. We consider an instance $\mathcal{I}_5$ of 2-PARTITION-EQUAL: given $2m$ positive integers $a_1, a_2, \ldots, a_{2m}$, does there exist a subset $I \subset \{1, \ldots, 2m\}$ of cardinal m such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. Let $S = \sum_{i=1}^{2m} a_i$, $W = \frac{S}{2} + 1$ and $b_i = \frac{S}{2} - 2a_i$ for $1 \leq i \leq 2m$. We build the instance $\mathcal{I}_6$ of our problem depicted in Fig. 5, and we ask whether there

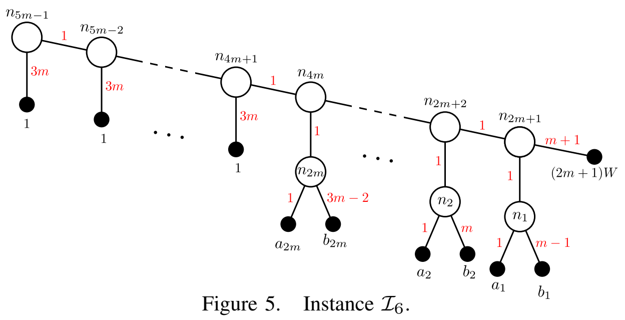

Figure 5. Instance $\mathcal{I}_6$.

exists a solution with $4m$ servers. We set $d_{\max} = 3m$, and distances are indicated on the edges in the figure.

Because of the constraints, any solution is forced to place a replica on nodes $n_{2m+1}, n_{2m+2}, \dots, n_{5m-1}$, and then the 2-partition comes from the fact that we need to place m replicas between nodes $n_1, \dots, n_{2m}$. Details can be found in [20].

B. A polynomial-time optimal algorithm

Theorem 6: MULTIPLE-BIN can be solved in polynomial time if $r_i \leq W$ for all $i \in \mathcal{C}$ (i.e., each client can entirely be served locally).

Proof: We exhibit a polynomial time algorithm, **multiple-bin**, which returns the optimal solution to MULTIPLE-BIN, for the special case where for all $i \in \mathcal{C}$, we have $r_i \leq W$. The solution is obtained by a call to **multiple-bin**(r), where r is the root of the binary tree. For each node $j \in \mathcal{N} \cup \mathcal{C}$, we keep a list of requests that are currently at this node and should still be processed, $req(j)$, and also a list of requests that are assigned to this node if it has been provided with a replica, $proc(j)$. These lists consist of triples (d, w, i), where an amount w of requests is issued by client $i \in \mathcal{C}$, and these requests can be served at a distance $d_{\max} - d$ from node j. Each list is sorted by non-increasing values of d. Moreover, the total number of requests in a list is never exceeding W, the server capacity.

The procedure is recursively updating these lists: initially, all lists are empty, and at the end, all client requests should appear in a list $proc(j)$. Initially, the set of servers is empty: $\mathcal{R} = \emptyset$. The call to **multiple-bin**(j) may lead to several cases.

1) If $j \in \mathcal{C}$ is a client node (i.e., a leaf node), and if $\delta_j > d_{\max}$, then the requests must be processed locally, therefore we place a server at node j (i.e., $\mathcal{R} = \mathcal{R} \cup \{j\}$) and we set $proc(j) = \{(0, r_j, j)\}$ and $req(j) = \emptyset$. Otherwise, we do not place any server yet, and we rather set $req(j) = \{(0, r_j, j)\}$. We will then attempt to process these requests further up in the tree.

2) Otherwise, if $j \in \mathcal{N}$ is an internal node, we first recursively call the procedure on both children nodes (recall that the tree is binary), denoted respectively $lchild(j)$ and $rchild(j)$, and therefore update all lists for nodes in $subtree(j)$, excluding j. We merge $req(lchild(j))$ and $req(rchild(j))$ as a temporary list $temp$ (remember that the lists are sorted by non-increasing d). Note that we add $\delta_{lchild(j)}$ (resp. $\delta_{rchild(j)}$) to the distances of the list $req(lchild(j))$ (resp. $req(rchild(j))$). If the first element of $temp$ is such that $d + \delta_j > d_{\max}$, some requests of the subtree cannot be processed upper in the tree, and therefore we add a server at node j. We also add a server at node j if there are more than W requests in $temp$, so that we keep less than W requests going up in the tree. In both cases, this server will be processing the first requests of the list, up to the server capacity, hence processing the requests that are the most constrained by distance. Because of the *Multiple* policy, we can process only a subset of the requests of a client to reach the exact capacity W, if there are more than W requests in the $temp$ list. We define $proc(j)$ as the first (at most) W requests of the list, and $req(j)$ contains the remaining requests. Note that there are no more than W requests in $req(j)$. Then, several cases occur.

a) If $req(j)$ is empty, we were able to process all requests of $subtree(j)$ at node j.
b) Otherwise, if the first request of the list $req(j)$ is such that $d+\delta_j > d_{\max}$, then there are more requests that cannot be processed upper in the tree. In this case, we place a new server in $subtree(j)$ and we may need to modify the assignment of requests to servers. This step is detailed below, it is done through a call to the procedure **extra-server**(j).
c) Otherwise, all requests of $req(j)$ may be processed by $parent(j)$, and we are ready to handle node $parent(j)$.

The procedure **extra-server**(j) works as follows. It adds a server on the first node that has not yet a server on the rightmost path of $subtree(j)$. To do so, we assign requests in a different way as was done at the beginning of step 2. Since all requests need to be processed in $subtree(j)$, and all requests in $req(lchild(j))$ and $req(rchild(j))$ can be processed at node j (otherwise, they would have been processed directly at the child node), we assign all requests from $req(lchild(j))$ to node j (i.e., we add these requests to $proc(j)$). If $rchild(j) \notin \mathcal{R}$, we just need to place a server at this node. Otherwise, we perform the recursive call **extra-server**($rchild(j)$), which will eventually move requests in $subtree(rchild(j))$. Note that the requests that may then be moved to $rchild(j)$ were going upper in the tree, and therefore they can be processed at node $rchild(j)$ without violating the distance constraint. Since the client nodes are leaves of the tree, with no more than W requests, we eventually reach a node that has no server and that can handle the remaining requests (it might be the rightmost client in $subtree(j)$).

This algorithm is formalized as Algorithm 2. We now prove that it returns an optimal solution to MULTIPLE-BIN. Given a problem instance, let $\mathcal{R}_{opt}$ be the set of servers chosen by an optimal solution, and let $\mathcal{R}_{algo}$ be the set of servers returned by our algorithm. Moreover,

Algorithm 2: Optimal algorithm for MULTIPLE-BIN.

```
procedure **multiple-bin**($j$)
begin
    $proc(j) = \emptyset$; $req(j) = \emptyset$;
    if $j \in \mathcal{C}$ then
        if $\delta_j > d_{\max}$ then $\mathcal{R} = \mathcal{R} \cup \{j\}$; $proc(j) = \{(0, r_j, j)\}$;
        else $req(j) = \{(0, r_j, j)\}$;
    else
        multiple-bin($lchild(j)$); multiple-bin($rchild(j)$);
        $temp =$ merge(add-dist($req(lchild(j)), \delta_{lchild(j)}$), add-dist($req(rchild(j)), \delta_{rchild(j)}$));
        Let $temp = \{(d, w, i), temp'\}$ and $wtot = \sum_{(d',w',i') \in temp} w'$;
        if $d + \delta_j > d_{\max}$ or $wtot > W$ then
            $\mathcal{R} = \mathcal{R} \cup \{j\}$; $wproc = 0$;
            while $temp \neq \emptyset$ and $wproc < W$ do
                Let $temp = \{(d, w, i), temp'\}$;
                if $wproc + w \leq W$ then
                    $wproc = wproc + w$;
                    $temp = temp'$; $proc(j) = \{proc(j), (d, w, i)\}$;
                else
                    $w' = W - wproc$; $wproc = W$;
                    $temp = \{(d, w - w', i), temp'\}$; $proc(j) = \{proc(j), (d, w', i)\}$;
        $req(j) = temp$;   /* At this point, $\sum_{(d',w',i') \in req(j)} w' < W$ */
        if $req(j) \neq \emptyset$ then
            Let $req(j) = \{(d, w, i), req'\}$;
            if $d + \delta_j > d_{\max}$ then extra-server($j$); $req(j) = \emptyset$;

procedure **merge**($req_1, req_2$)
begin
    if $req_1 = \emptyset$ then return $req_2$;
    else if $req_2 = \emptyset$ then return $req_1$;
    else
        Let $req_1 = \{(d_1, w_1, i_1), req_1'\}$ and $req_2 = \{(d_2, w_2, i_2), req_2'\}$;
        if $d_1 \geq d_2$ then
            return $\{(d_1, w_1, i_1), (d_2, w_2, i_2),$ merge($req_1', req_2'$)$\}$;
        else return $\{(d_2, w_2, i_2), (d_1, w_1, i_1),$ merge($req_1', req_2'$)$\}$;

procedure **add-dist**($req, dist$)
begin
    if $req = \emptyset$ then return $\emptyset$;
    else
        Let $req = \{(d, w, i), req'\}$;
        return $\{(d + dist, w, i),$ add-dist($req', dist$)$\}$;

procedure **extra-server**($j$)
begin
    $proc(j) = req(lchild(j))$;
    if $rchild(j) \notin \mathcal{R}$ then $\mathcal{R} = \mathcal{R} \cup \{rchild(j)\}$; $proc(rchild(j)) = req(rchild(j))$;
    else extra-server($rchild(j)$);
```

let $W_{tot} = \sum_{i \in \mathcal{C}} r_i$ be the total number of requests. Finally, let $\mathcal{K}$ be the set of servers that follow one of these properties: (i) it has been added at line 5 of the algorithm; (ii) it has been added at line 12 and it is processing strictly less than W requests ($wproc < W$ at the end of the loop); (iii) it leads to a call to **extra-server** at line 24. In all cases, if $j \in \mathcal{K}$, then $req(j) = \emptyset$.

First note that, for all $j \in \mathcal{C} \cup \mathcal{N}$, we have $\sum_{(d,w,i) \in req(j)} w \leq W$. This property is true for $j \in \mathcal{C}$ (see line 6) since $r_j \leq W$ by definition. For $j \in \mathcal{N}$, if there are too many pending requests arriving to the node, we place a server at node j processing W requests, and since the tree is binary, the remaining requests that are in $temp$, and then $req(j)$ (see line 21) are not exceeding W. Moreover, requests in $req(j)$ can always be processed by $parent(j)$ because of the distance constraint. Otherwise, either a server is placed line 5, or the procedure **extra-server** is called line 24. Note that at the root of the tree ($j = r$), we have $\delta_r = +\infty$, and therefore $req(j) = \emptyset$ at the end.

We partition the servers according to the set $\mathcal{K}$: we assign each server $j \in \mathcal{R}$ to its closest ancestor $k \in \mathcal{K}$. Let $serv(k)$ be the set of servers whose closest ancestor in $\mathcal{K}$ is k. Then, $p(k) = \cup_{j \in serv(k)} proc(j)$ is the set of requests processed by servers in $serv(k)$, and $r(k) = \sum_{(d,w,i) \in serv(k)} w$ is the total number of requests processed in the subtree of k from which we have removed subtrees rooted in nodes $k' \in \mathcal{K}$. By definition of $\mathcal{K}$, no request is going through a node $k \in \mathcal{K}$. We have therefore partitioned the tree: $W_{tot} = \sum_{k \in \mathcal{K}} r(k)$, and $|\mathcal{R}_{algo}| = \sum_{k \in \mathcal{K}} |serv(k)|$.

Given a node $k \in \mathcal{K}$, we now show that an amount of requests strictly greater than $(|serv(k)| - 1)W$ included in $r(k)$ could not have been handled by a node upper in the tree than k, even by an optimal solution. We differentiate two cases.

- If k has been added on line 5 or 12, all servers in $serv(k)$ (excepting k) are processing exactly W requests, and by construction, these requests are more constrained by distance than the requests processed by node k. Since some of the requests processed by node k cannot be handled by a node upper in the tree because of the distance constraint, it is also the case for those $(|serv(k)| - 1)W$ requests.
- If k leads to a call to **extra-server**(k), the reasoning is the same before the call to **extra-server**. At this moment we had $|serv(k)| - 1$ servers, including k, each of them processing exactly W requests, but some more requests could not be handled higher in the tree. The total amount of requests processed in the subtree (from which we remove subtrees rooted in nodes in $\mathcal{K}$) is therefore strictly greater than $(|serv(k)| - 1)W$.

We further need to prove that the **extra-server** procedure always succeeds in re-arranging requests and adding an extra server. The key is that we never violate a distance constraint. Indeed, we consider the requests in $req(j')$, and these requests can always be processed by $parent(j')$. Therefore, when calling **extra-server**(j), it is always possible to process at node j all the requests from $req(lchild(j))$. We do not need to fill server j to W requests, since we already have a lower bound on the total number of requests in the subtree rooted in $k \in \mathcal{K}$. If there is already a server at node $rchild(j)$, by construction it is processing W requests, but we can iterate the procedure. We will eventually reach the case $rchild(j) \notin \mathcal{R}$: if $rchild(j) \in \mathcal{C}$, then it is not in $\mathcal{R}$, otherwise we would have $req(rchild(j)) = \emptyset$, and therefore j could process all the requests (coming only from $req(lchild(j))$), and therefore we would not need an extra server in the subtree, hence having at most $(|serv(k)| - 1)W$, which leads to a contradiction.

Finally, we proceed to a recursive bottom up analysis: consider the lowest node $k \in \mathcal{K}$ in the tree $\mathcal{T}$. In the subtree rooted at k, an amount of requests strictly greater than $(|serv(k)| - 1)W$ cannot be handled upper in the tree. Thus, even an optimal solution requires the use of at least $|serv(k)|$ servers to be put in this subtree. Since no requests are sent from k to a node upper in the tree ($req(k) = \emptyset$), we can now consider for our analysis the tree $\mathcal{T}$ without the subtree rooted at k, and apply the same argument recursively to prove that an optimal solution requires at least as many servers as Algorithm 2.

Note that the algorithm performs exactly $|\mathcal{C} \cup \mathcal{N}|$ calls to **multiple-bin**. The complexity of the **merge** and **add-dist** procedures is in $O(|\mathcal{C}|)$ (at most one triple per client node), and similarly the *while* loop takes no more than $O(|\mathcal{C}|)$ iterations. Finally, the call to **extra-server** cannot be made several times on a same node, so there are no more than $|\mathcal{C} \cup \mathcal{N}|$ calls to this procedure. Overall, the complexity of this algorithm is in $O(|\mathcal{T}|^2)$, and therefore it is polynomial in the problem size, which concludes the proof. ■

V. Conclusion

In this paper, we have investigated the problem of replica placement in tree network. While several instances of this problem have already been studied, there were still some complexity gaps. We have focused on two policies: either all requests of a client must be served by a single server in the tree, on the path between the client and the root of the tree (*Single* policy), or the requests of a given client can be processed by multiple servers (*Multiple* policy), still on this path. Moreover, we have considered problem instances with distance constraints, hence expressing guarantees on the quality of service.

For the *Single* policy, we have established the NP-completeness in the strong sense of the simplest problem instance, namely SINGLE-NOD-BIN. Moreover, we have shown that unless P=NP, for all $\varepsilon > 0$, there is no $(\frac{3}{2} - \varepsilon)$-approximation algorithm for this problem. Therefore, we have designed two approximation algorithms for SINGLE. The first one solves the most general problem instance, and it is a $(\Delta + 1)$-approximation algorithm, where Δ is the arity of the tree. Then we have refined the previous algorithm in the case without distance constraints, hence proposing a 2-approximation algorithm for SINGLE-NOD. While the algorithms are greedy and easy to implement, the proofs of the approximation ratios are quite involved. Then, focusing on the *Multiple* policy, we proposed a sophisticated polynomial time algorithm that optimally solves the MULTIPLE-BIN problem, while MULTIPLE is known to be NP-hard. In fact, this algorithm works only when each request can be processed entirely by a single server, i.e., it does not exceed the server capacity. Otherwise, we prove that the problem remains NP-hard.

As future work, there remain some complexity gaps to fill. In particular, we believe that we can design a $3/2$-approximation algorithm for SINGLE-NOD-BIN, hence closing the gap, but we have not yet been able to prove the approximation ratio. A greedy algorithm is unlikely to be good enough, and we rather envision to *push* servers towards the root of the tree, whenever possible. As for MULTIPLE, we plan to design approximation algorithms for the general NP-hard problem.

Acknowledgment: A. Benoit is with the Institut Universitaire de France. This work was supported in part by the ANR *RESCUE* project.

REFERENCES

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

[2] J.-J. Wu, Y.-F. Lin, and P. Liu, "Optimal replica placement in hierarchical Data Grids with locality assurance," *J. Parallel and Distributed Computing*, vol. 68, no. 12, pp. 1517–1538, 2008.

[3] A. Benoit, V. Rehn-Sonigo, and Y. Robert, "Replica placement and access policies in tree networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1614–1627, 2008.

[4] K. Kalpakis, K. Dasgupta, and O. Wolfson, "Optimal placement of replicas in trees with read, write, and storage costs," *IEEE Trans. Parallel and Distributed Systems*, vol. 12, no. 6, pp. 628–637, 2001.

[5] 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 CS Press, 2006.

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

[7] G. Rodolakis, S. Siachalou, and L. Georgiadis, "Replicated server placement with QoS constraints," *IEEE Trans. Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1151–1162, 2006.

[8] M. Pál, E. Tardos, and T. Wexler, "Facility location with nonuniform hard capacities," in *Proceedings of FOCS'01, the 42nd IEEE symp. on Foundations of Computer Science*, 2001, p. 329.

[9] D. Shmoys, E. Tardos, and K. Aardal, "Approximation algorithms for facility location problems," in *Proceedings of the 29th Symp. on Theory of Computing*, 1997.

[10] F. A. Chudak and D. P. Williamson, "Improved approximation algorithms for capacitated facility location problems," in *Proceedings of IPCO*, 1999, pp. 99–113.

[11] M. Kao, C. Liao, and D. Lee, "Capacitated domination problem," *Algorithmica*, pp. 1–27, 2009.

[12] J. E. G. Coffman, M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: a survey," in *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1997, pp. 46–93.

[13] W. F. de la Vega and G. Lueker, "Bin packing can be solved within $1 + \varepsilon$ in linear time," *Combinatorica*, pp. 1:349–355, 1981.

[14] O. Beaumont, N. Bonichon, and H. Larchevêque, "Modeling and Practical Evaluation of a Service Location Problem in Large Scale Networks," in *Proceedings of ICPP'11*, 2011, pp. 482–491.

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

[16] T. Loukopoulos, I. Ahmad, and D. Papadias, "An overview of data replication on the Internet," in *Proceedings of ISPAN'02, the Int. Symp. on Parallel Architectures, Algorithms and Networks*, 2002.

[17] J. Bar-Ilan, G. Kortzars, and D. Peleg, "How to allocate network centers," *J. Algorithms*, pp. 15:385–415, 1993.

[18] S. Khuller and Y. Sussmann, "The Capacitated K-Center Problem," *SIAM Journal on Discrete Mathematics*, vol. 13, p. 403, 2000.

[19] B. Codenotti, G. D. Marco, M. Leoncini, M. Montangero, and M. Santini, "Approximation algorithms for a hierarchically structured bin packing problem," *Inf. Process. Lett.*, vol. 89, no. 5, pp. 215–221, 2004.

[20] A. Benoit, H. Larchevêque, and P. Renaud-Goud, "Optimal algorithms and approximation algorithms for replica placement with distance constraints in tree networks," INRIA, France, Research Report, Sep. 2011. [Online]. Available: http://graal.ens-lyon.fr/~abenoit/