

中央研究院  
資訊科學研究所

Institute of Information Science, Academia Sinica • Taipei, Taiwan, ROC

TR-IIS-07-016

## Optimal Replica Placement in Data Grid Environments with Locality Assurance

Pangfeng Liu, Yi-Fang Lin, Jan-Jan Wu



Oct. 30, 2007 || Technical Report No. TR-IIS-07-016

<http://www.iis.sinica.edu.tw/page/library/LIB/TechReport/tr2007/tr07.html>

# Optimal Replica Placement in Data Grid Environments with Locality Assurance

Pangfeng Liu Yi-Fang Lin

Department of Computer Science

National Taiwan University

Taipei, Taiwan, R.O.C.

pangfeng@csie.ntu.edu.tw

Jan-Jan Wu

Institute of Information Science

Academia Sinica

Taipei, Taiwan, R.O.C.

wuj@iis.sinica.edu.tw

## Abstract

Data replication is typically used to improve access performance and data availability in Data Grid systems. To date, research on data replication in Grid systems has focused on infrastructures for replication and mechanisms for creating/deleting replicas. The important problem of choosing suitable locations to place replicas in Data Grids has not been well studied.

In this paper, we address three issues concerning data replica placement in Data Grids. The first is how to ensure load balance among replicas. To achieve this, we propose a placement algorithm that finds the optimal locations for replicas so that their workload is balanced. The second issue is how to minimize the number of replicas. To solve this problem, we propose an algorithm that determines the minimum number of replicas required when the maximum workload capacity of each replica server is known. Finally, we address the issue of service quality by proposing a new model in which each request must be given a quality-of-service guarantee. We describe new algorithms that ensure *both* workload balance and quality of service *simultaneously*.

**Keywords** Data grid systems, Replica placement, Load balancing, Locality assurance.

## 1 Introduction

Grid computing is an important mechanism for utilizing computing resources that are distributed in different geographical locations, but are organized to provide an integrated service. A grid system provides computing resources that enable users in different locations to utilize the CPU cycles of remote sites. In addition, users can access important data that is only available in certain locations, without the overheads of replicating it locally. These services are provided by an integrated grid service platform, which helps users access the resources easily and effectively.

One class of grid computing, and the focus of this paper, is Data Grids, which provide geographically distributed storage resources for complex computational problems that require the evaluation and management of large amounts of data [3, 10, 16]. For example, scientists working in the field of bioinformatics may need to access human genome databases in different remote locations. These databases hold tremendous amounts of data, so the cost of maintaining a local copy at each site that needs the data would be prohibitive. In addition, such databases are usually read-only, since they contain the input data for various applications, such as benchmarking, identification, and classification. With the high latency of the wide-area networks that underlie most Grid systems, and the need to access/manage several petabytes of data in Grid environments, data availability and access optimization have become key challenges that must be addressed.

An important technique that speeds up data access in Data Grid systems replicates the data in multiple locations so that a user can access it from a site in his vicinity. It has been shown that data replication not only reduces access costs, but also increases data availability in many applications [10, 17, 15]. Although a substantial amount of work has been done on data replication in Grid environments, most of it has focused on infrastructures for replication and mechanisms for creating/deleting replicas [4, 7, 6, 8, 10, 15, 18, 17, 19]. We believe that, to obtain the maximum benefit from replication, strategic placement of the replicas is also necessary.

Thus, in this paper, we address the problem of replica placement in Data Grid systems. We assume a *hierarchical data grid* model because of its resemblance to the hierarchical grid management model found in most grid systems [9, 6, 10, 18]. For example, in the LCG (WorldWide Large Hadron Collider Computing Grid) [9] project, 70 institutes from 27 countries form a grid system organized as a hierarchy, with CERN (the European Organization for Nuclear Research) as the root, or *tier-0* site. There are 11 *tier-1* sites directly under CERN that help distribute data obtained from the Large Hadron Collider (LHC) at CERN. Meanwhile, each tier-2 site in the LCG hierarchy receives data from its corresponding tier-1 site. The entire LCG grid can be represented as a tree structure. In the forthcoming EGEE/LCG-2 Grid, the tree structure will comprise 160 sites in 36 countries.

In our data grid model, data requests issued by a client are served by traversing the tree upwards towards the root until a data copy is found. In many real-world grid systems, like the multi-tier LCG grid system [9], data requests go from tier-2 to tier-1 sites, and then to tier-0 sites if necessary, in the search of requested data. In addition, the grid hierarchy usually reflects the structure of the organization or the geographic locality, so our assumption that data requests travel up towards the root is reasonable.

Our replica placement strategy considers a number of important issues. First, the replicas should be placed in proper server locations so that the workload on each server is balanced. A naive placement strategy may cause “hot spot” servers that are overloaded, while other servers are under-utilized.

Another important issue is choosing the optimal number of replicas. The denser the distribution of replicas, the shorter the distance a client site needs to travel to access a data copy. However, as noted earlier, maintaining multiple copies of data in Grid systems is very expensive; therefore, the number of replicas should be

bounded. Clearly, optimizing the access cost of data requests and reducing the cost of replication are two conflicting goals. Thus finding a suitable trade-off between the two factors is a challenging task.

Finally, we consider the issue of service locality. Each user may specify the minimum distance he/she can allow between him/her and the nearest data server. This is a locality assurance requirement that users may stipulate, and the system must ~~therefore ensure that~~ there is a server within the specified range to handle the request.

In this paper, we study two data grid models. The first is called the “unconstrained model”, since the client can not specify locality requirements. This model provides us with insights into designing algorithms that balance the workload among replicas. The second model is called the “constrained model” in which each client can assign his/her own acceptable *quality of service*, in terms of the number of hops to the root of the tree. This is an important requirement, since different users may require different levels of service quality. We model this as a range limit, so ~~that~~ the placement algorithm must ensure that all the replicas are placed in such a way that all the quality of service (QoS) requirements are satisfied. In the constrained model, the problem of evenly distributing the load of data requests over all the replicas is more challenging than in the unconstrained model.

We develop theoretical foundations for replica placement in the two models (constrained and unconstrained). Furthermore, we devise efficient algorithms that find optimal solutions for the three important problems described above.

The remainder of the paper is organized as follows. Section 2 reviews related works. In Section 3, we describe the unconstrained model, and formally define our replica placement problem. Section 4 presents our replica placement algorithms for the unconstrained model. Section 5 defines the constrained model, while Section 6 presents our replica placement algorithms for the constrained model and provides a theoretical analysis of them. Finally, in Section 7, we present our conclusions, address some as yet unresolved problems, and indicate the direction of our future work.

## 2 Related Work

A number of works have addressed the placement of data replicas in parallel and distributed systems based on regular network topologies, such as hypercubes, tori, and rings. These networks possess many attractive mathematical properties that facilitate the design of simple and robust placement algorithms [2, 21]. However, such algorithms cannot be applied to Data Grid systems directly due to the latter’s hierarchical network structures and special data access patterns, which are not common in traditional parallel systems. An early approach to replica placement in Data Grids, reported in [1], proposed a heuristic algorithm called **Proportional Share Replica** for the placement problem. However, the algorithm does not guarantee that the optimal solution will be found.

Another group of related works that focus on placing replicas in a tree topology can be divided into two types of model. The first ~~type~~ allows a request to go up and down the tree searching for the nearest replica. For example, Wolfson and

Milo [23] proposed a model in which no limit is set on the server’s capacity. There are two cost categories in this model: the *read cost* and the *update cost*. The read cost is usually defined as the number of hops, or the sum of the communication link costs from a request to its server. The update cost, on the other hand, is usually proportional to the sum of the communication link costs of the minimum subtree that spans all the replicas. The goal in [23] is to minimize the sum of the read and update costs, which can be achieved by a greedy method in  $O(n)$  time, where  $n$  is the number of nodes. Kalpakis et. al. [13] developed a model in which each server has a capacity limit and each site has a different building cost, called the *site cost*. The objective is to minimize the summation of the read, update and site costs. The authors showed that the optimal placement could be found in  $O(n^5 C^2)$  time; however, for an incapacitated server model, the cost would be  $O(n^5)$ , where  $n$  is the number of nodes and  $C$  is the maximum capacity of each tree node. Unger and Cidon [22] suggested a similar model to that of Kalpakis, but without the server capacity limit. As a result, the time required to find the optimal placement is reduced to  $O(n^2)$  [22]. Jaeger and Goldberg [11] proposed a model in which there are a known number of servers in the tree, each with equal capacity. There are no read, write, or site building costs, and the goal is to assign the request to a server (not necessarily the nearest one), so that the maximum distance from a client to its assigned server is minimized. The optimal solution can be found by a greedy method in  $O(n^2)$  time. Korupolu et. al. [14] developed a model in which the read cost is slightly different from that of other models. A data request must go from the client to the least common ancestor of the client and the replica, and then to the replica so that the read cost is proportional to the depth of the subtree rooted at the common ancestor. The authors present various approximation algorithms that achieve good replica placement [14].

The second ~~set of models in the literature~~ only allow a request to search for a replica towards the root of the tree. For example, Jia et. al. [12] suggested a model in which neither the server’s capacity nor the site’s building cost are set. The goal is to minimize the sum of the read and update costs while placing  $k$  replicas, which can be achieved by dynamic programming in  $O(n^3 k^2)$  time [12]. Cidon et.al. [5] proposed a similar model in which a replica is associated with a site’s building cost, but there is no update cost. The goal is to minimize the sum of the read and storage costs, which can be achieved by dynamic programming in  $O(n^2)$  time. Tang and Xu [20] subsequently described a model in which there is a range limit on the number of hops a request can make from its assigned replica, but there is no server capacity limit. The objective is to find a feasible solution and minimize the sum of the update and storage costs, which can also be achieved by dynamic programming in  $O(n^2)$  time [20].

Our model focuses on the tree topology in which the requests only travel up towards the root. In real-world grid systems like LCG [9], the requests go from tier-2 to tier-1 and then to tier-0 sites searching for data. The grid hierarchy usually reflects the structure of the organization or geographic locality, so the assumption that requests travel up towards the root is reasonable.

In most works, the objective is to minimize the *sum* of the read/write costs generated by all requests. However, a Data Grid system usually consists of multiple data servers connected by switches that enable concurrent data transfer between

independent pairs of clients and servers. Therefore, in a Data Grid environment, the overall performance of the system is dominated by the performance of the server with the heaviest workload. Although we believe that the load balance of servers is the key optimization criterion for Data Grid systems, we also believe that load balance and quality of ~~service~~ should be considered simultaneously. To date, this aspect has not been addressed in the literature.

### 3 The Unconstrained Model

Before addressing the issue of workload balance among replicas, we describe our unconstrained data grid model in detail. We use a tree  $T$ , to represent a data grid system. The root of the tree, denoted by  $r$ , is the *hub* of the data grid. A database replica can be placed in any tree node, except the hub  $r$ . All the tree's leaves are local sites, where users can access databases stored in the data grid system.

Users of a local site can access a database as follows. First a user request tries to locate the database replica locally. If the replica is not found, the request travels up the tree to the parent node to search for the replica. In other words, the user request goes up the tree and uses the first replica encountered on the path to the root. If after traveling up the path, a replica is not found, the hub will service the request. For example, in Figure 1, a user request at node  $a$  ~~tries to access data~~. As the data is not available at ~~node~~  $a$ , the request goes to the parent node  $b$ , where the data is not available either. Finally the request reaches node  $c$ , where the replica is found.

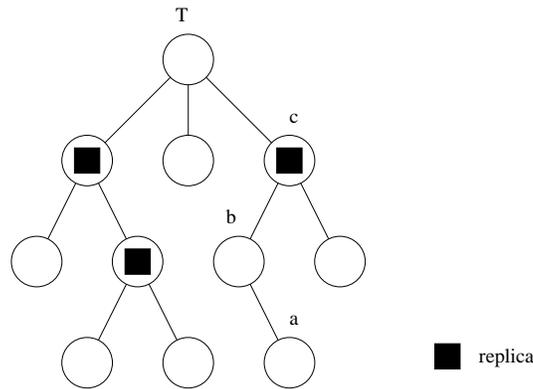


Figure 1: A data grid tree  $T$ .

The goal of our replica placement strategy is to place the replicas so that various objectives can be satisfied. This raises a number of questions. For example, if we can accurately estimate ~~the frequency that~~ a leaf node is used to locate specific data, where do we place a given number of replicas so that the maximum amount of data a replica has to handle is minimized? In addition, if we fix the workload that a replica can handle, how many replicas do we need, and where should we put them?

We now formally define the goals of our replica placement strategy. Let  $l$  be a leaf node of the set of all leaves  $L$ , and let  $w(l)$  be the number of data requests

generated by  $l$ . Note that, for ease of discussion, we focus on the case where only leaves can request data. All the results in this paper can be generalized to cases where all the tree's nodes, including the internal nodes, can request data. Next, based on the data grid access model described above, we define the *workload* of a particular node after the replicas have been placed. Let  $T$  be a data grid tree,  $N$  be the set of nodes in  $T$ , and  $R$  be a subset of  $N$ , with a replica placed on every node of  $R$ . The workload for a node  $n$  of  $N$ , denoted as  $f(n)$ , is defined recursively as follows:

$$f_R(n) = \begin{cases} w(n) & \text{if } n \text{ is a leaf} \\ \sum_c f_R(c) & c \text{ is a child of } n, \text{ and } c \notin R. \end{cases}$$

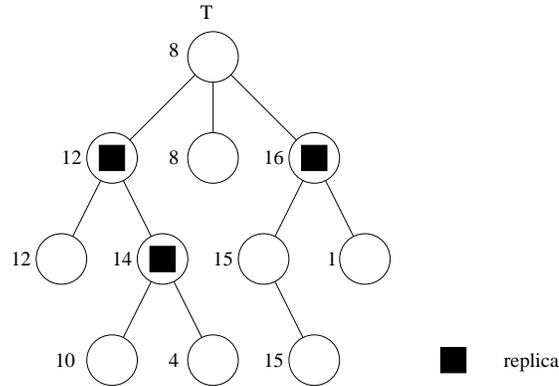


Figure 2: The actual workload of the nodes in a data grid tree  $T$ .

The maximum workload of  $R$  is the maximum workload of all nodes of  $R$  and *the hub*. We include the hub because it services all data requests not serviced by replicas. We now formally define the problems.

- *MinMaxLoad*: given the number of replicas  $k$ , find a subset of tree nodes  $R$  that minimizes the maximum workload.
- *FindR*: given the amount of data  $D$  a replica or the hub can service, find the subset  $R$  with minimum cardinality such that the maximum workload is not greater than  $D$ .

## 4 Algorithms for the Unconstrained Model

In this section, we describe our algorithms for solving the *MinMaxLoad* and *FindR* problems in the unconstrained model. We solve *FindR* first, and use that algorithm to solve *MinMaxLoad*.

### 4.1 FindR

The *FindR* problem can be stated as follows. Given a grid tree and the workload on its leaves, a constant  $k$ , and a maximum workload  $D$ , find a subset of tree nodes  $R$  with cardinality no more than  $k$  in which to place the replica so that the workload



on every  $r \in R$  and on the hub is no more than  $D$ . ~~All these~~  $R$  sets are referred to as “feasible”. A feasible  $R$  is *optimal* if it minimizes the workload on the hub.

To simplify the discussion, we first classify tree nodes into two categories. Suppose there is *no* replica in the tree, the workload on a leaf  $n$  is just  $w(n)$ , and the workload on an internal node is the sum of the workloads of its children. If a tree node has a workload greater than  $D$ , we call it a *heavy* node; otherwise, it is a *light* node. If a light node has a heavy parent, we call it a *critical* node. Figure 3 illustrates the case where  $D$  equals 18. It is obvious that we can always find an optimal  $R$  for the FindR problem; thus  $R$  does not contain any non-critical light nodes.

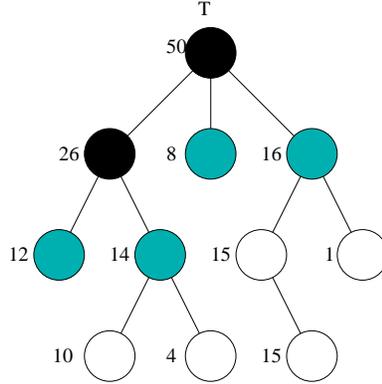


Figure 3: The workload of the nodes in a data grid tree  $T$  when  $D$  is 18. A heavy node is represented by a dark circle, and a critical node is represented by a gray circle.

**Observation 1** *There exists an optimal replica set that does not contain any non-critical light nodes.*

**Proof.** Let  $R^*$  be an optimal replica set that contains  $n$ , a non-critical light node. Consider the unique critical light ancestor  $m$  of  $n$ , and the subtree rooted at  $m$ . All the replicas in this subtree can be replaced by a single replica at  $m$ , without increasing the workload on any of  $m$ 's ancestors. This is a feasible solution, since, by the definition of light node, the total workload of this subtree is at most  $D$ . ■

With Observation 1 in place, we only need to consider heavy and critical nodes in our search for the optimal replica set. The following lemma reduces our search space further.

**Lemma 1** *Let  $T$  be a data grid tree,  $p$  be a heavy node with only critical children, and  $e$  be the child of  $p$  that has the maximum workload. There exists an optimal replica set that contains  $e$ .*

**Proof.** Consider an optimal replica set  $R^*$  that must contain a child of  $p$  (denoted as  $f$ ); otherwise,  $p$  will be flooded with more than  $D$  requests. If  $f$  is not  $e$ , we replace it with  $e$ . The new replica set is feasible, since both  $e$  and  $f$  are light. Also this switch will not increase the workload on  $p$  or any of its ancestors. ■

By Observation 1 and Lemma 1, we derive a baseline algorithm, called **Feasible**, for FindR. Given the tree  $T$ , the replica capacity  $D$ , and the number of replicas allowed  $k$ , the **Feasible** algorithm determines whether there is a feasible replica set with cardinality  $k$  or less by repeatedly picking the critical leaf that has the maximum workload at most  $k$  times. If the tree becomes empty, a solution is found. The pseudo code of **Feasible** is given in Figure 4.

```

Feasible(T, D, k)
{
  Compute the workload of all internal nodes.
  Remove all non-critical nodes.
  Repeat at most k times
  {
    Pick a heavy node that has only critical leaves.
    Pick the critical leaf (e) that has the maximum workload.
    Add e to R.
    Remove e from the tree.
    Adjust the workload for all the ancestors of e.
    If the ancestor becomes non-critical, remove it.
  }
  If the tree is empty,
    we have a solution R;
  else
    we do not have a feasible replica set.
}

```

Figure 4: The pseudo code of the baseline algorithm **Feasible**.

Note that once we pinpoint a critical child  $e$  in which we want to place a replica, we must deduct  $w(e)$  from the workload of *all* of its ancestors, including the hub. This might cause some of the heavy ancestors to become light nodes which are non-critical and should be removed. We repeatedly update the workload towards the root, remove ancestors and their subtrees if they become non-critical, and finally reach a now critical node. We repeat this process by selecting a heavy node with only critical children, as shown in Figure 5.

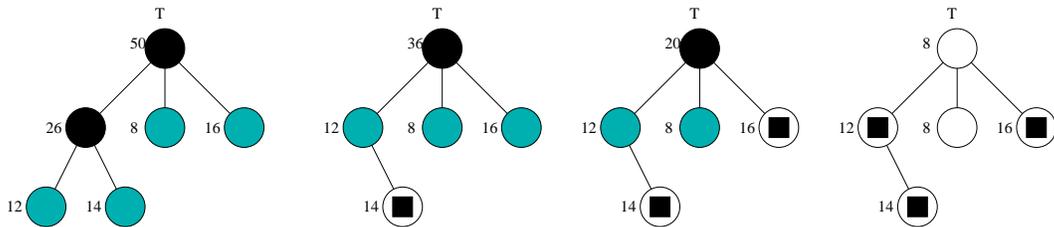


Figure 5: An execution scenario of the baseline algorithm **Feasible**. The capacity  $D$  is set at 18.

We analyze the time complexity as follows. Let  $n$  be the number of tree nodes. It only takes  $O(n)$  time to compute the workload when no replica is placed, so we can determine the category for every node. Also, since a node can only be removed once, the removal cost is bounded by  $O(n)$ . However, the cost of updating the workload of the ancestors could be very expensive. For example, consider a skewed tree of height  $\Omega(n)$ . We may need to update all the ancestors of every child with the maximum workload that we pick from the bottom of the tree such that the total cost could be as high as  $\Omega(kn)$ .

## 4.2 Lazy Updating

We improve our **Feasible** algorithm by introducing a concept called *lazy update*, which ensures that the update cost is not prohibitive. Lazy update assigns a *deduction value* to each internal node  $n$  (denoted by  $d(n)$ ) to keep track of the amount of the workload that should be removed from  $n$  and all of its ancestors.

The lazy update mechanism traverses the heavy nodes depth first. When a heavy node with only critical children is found, lazy update picks the child (denoted as  $c$ ) with the maximum workload on which to place a replica in the same way as the baseline algorithm **Feasible**. It then subtracts  $w(c)$  (the workload of  $c$ ) along the path from  $c$  back to the hub, as shown in Figure 6. If an ancestor becomes non-critical, it is removed, as in **Feasible** (e.g., nodes  $g$  and  $h$  in Figure 6). When the lazy update reaches a heavy node (node  $e$  in Figure 6) that becomes critical after reducing its workload by  $w(c)$ , it does not try to deduct  $w(c)$  from all the ancestors of  $e$ . Instead, it increases the deduction from  $e$ 's parent (denoted by  $f$  in Figure 6) by  $w(c)$ , and starts the traversal from  $f$ . Note that, with the help of this "deduction", we can eliminate the duplication of deducting workloads from those tree nodes that are on the same path from a leaf to the hub. As a result, when the lazy update deducts some workload from  $e$ , it must add the deduction of  $f$  by  $w(c)$ .

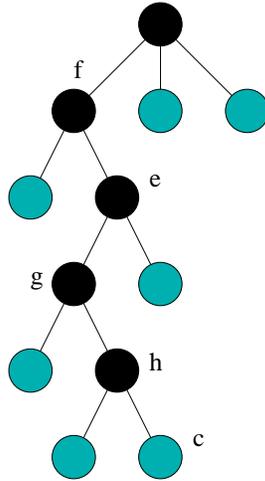


Figure 6: An execution scenario of the lazy update.

The purpose of the depth-first-search is to ensure that all the *deduction* values are kept on the same path of the subtree that the request is traversing to the hub.

Otherwise, one part of the tree may not be aware of the value of a deduction in another part of the tree, and the decision about whether or not a tree node is heavy could be incorrect.

```

LazyUpdate(v)
{
  if v is a heavy node that has only critical leaves
  {
    Pick the critical leaf (e) that has the maximum workload.
    Add e to R.
    Remove e from the tree.

    Adjust the workload of ancestors on the path to the hub.
    If an ancestor becomes non-critical, remove it until a
      critical ancestor c is found.
    Add w(e) to the deduction of c's parent.
    LazyUpdate(c's parent)
  } else {
    For each heavy child h
      LazyUpdate(h)
  }
}
Until all tree nodes have been removed.

```

Figure 7: The pseudo code of the LazyUpdate algorithm for FindR.

Next, we analyze the time complexity of lazy update, especially the deduction component. As each node can only be removed once, the cost of removal is bounded by  $O(n)$ , where  $n$  is the number of tree nodes. When a replica is placed on a critical node  $c$ , the ancestors of  $c$  could be updated in three ways. First, an ancestor could be removed, since, after deducting  $w(c)$ , it becomes a light node (i.e., nodes  $g$  and  $h$  in Figure 6). Second, an ancestor could become a critical node (node  $e$ ) after  $w(c)$  is deducted from its workload. Third, an ancestor could add  $w(c)$  to its deduction value. Because an ancestor can only be removed once, the total cost of the first kind of lazy update is bounded by  $O(n)$ . Also, each replica that is placed will incur the second and third kinds of update once, so their total cost is bounded by  $O(k)$ .

We now analyze the cost of selecting the critical node with the maximum workload among its siblings. It is easy to see that there can be at most  $k$  such selections because we can only pick at most  $k$  replicas. For every internal node, we need to maintain the relative order among its children according to their workload. Once the workload of any child is changed (e.g., due to a replica placed in its subtree), the relative order needs to be recomputed. As a result, we need a data structure that supports fast insertion/deletion, and selection of the maximum workload. Clearly, we only need to keep the  $k$  largest children of every internal node, since we have at most  $k$  replicas. Therefore, we do not need to keep track of all the children; the  $k$  largest children will be sufficient. We achieve this goal with a sorted list containing at most  $k$  elements. The overall list maintenance time is bounded by  $O(k \log k)$ .

Initially the sorted list contains the  $k$  largest children of every parent. Whenever we need to place a replica on the heaviest child (denoted by  $c$  in Figure 6) of a parent node  $h$ , we check if  $h$  remains heavy. If it does, we remove  $c$  from the list of  $h$ , and finish in  $O(1)$  time. If  $h$  becomes light, we remove  $h$  and start moving up the tree until we reach a critical ancestor  $e$ . Now we must update the workload of  $e$ , and find a new position for  $e$  in the sorted list of children of  $f$ , where  $f$  is the parent of  $e$  (Figure 6). It takes  $O(\log k)$  time to insert the new updated child into the sorted list, since the list has at most  $k$  elements. Recall that the overall list maintenance time is bounded by  $O(k \log k)$ , as there are at most  $k$  replicas to place and we make at most one insertion for every replica placed.

The only remaining question is: How can we initialize the sorted list for every internal node? If the value of  $k$  is small, we simply choose the  $k$  largest children of every parent repeatedly, with a total time of  $O(kn)$ . If the value of  $k$  is large, we sort *all* tree nodes with the parent as the primary key, and the workload as the secondary key. Each parent will then be aware of its  $k$  largest children; therefore, the initialization cost is  $O(\min(kn, n \log n))$ .

Finally we aggregate all the costs. The time taken to initialize the sorted list is  $O(\min(nk, n \log n))$ , the time required to maintain the sorted lists is  $O(k \log k)$ , and the time for tree traversal and updating the workload is  $O(n)$ . The total time is bounded by  $O(\min(nk, n \log n) + k \log k + n) = O(n \log n)$  since  $k \leq n$ .

**Theorem 1** *The LazyUpdate algorithm finds the optimal replica set for FindR in  $O(n \log n)$  time, where  $n$  is the number of tree nodes in the data grid.*

### 4.3 MinMaxLoad

With the LazyUpdate algorithm in place, we are ready to derive an algorithm called BinSearch for the MinMaxLoad problem, which can be stated as follows. Given a grid tree, the workload on its leaves, and a constant  $k$ , find a subset of tree nodes  $R$  with cardinality no e than  $k$  in which to place the replica so that the maximum workload on every  $r \in R$  and ~~the workload~~ on the hub is minimized.

The BinSearch algorithm finds the replica set by “guessing” the maximum workload through a binary search. We guess a value  $D$  as the maximum workload on the replica and the hub. If the LazyUpdate algorithm can not find a feasible replica set for this  $D$ , we increase the value of  $D$ ; otherwise, we reduce it. We assume that all the workloads are integers and there is an upper bound  $U$  on the workload of every node; therefore, the total workload is bounded by  $O(nU)$ . Clearly, after  $O(\log n + \log U)$  calls of LazyUpdate, we will be able to find the smallest value of  $D$  such that  $k$  replicas are sufficient.

Next, we analyze the time complexity of BinSearch. Note that in LazyUpdate, we need an initialization phase that computes a sorted list of children for every parent. This task is only performed once in BinSearch, since the tree is the same throughout the binary search. Each iteration of LazyUpdate takes  $O(k \log k + n)$  time, but the total cost of BinSearch is bounded by  $O(\min(nk, n \log n) + (\log n + \log U)(k \log k + n))$ . In a grid system, the number of replicas,  $k$ , is usually bounded by a small constant, ~~meaning that~~  it is very expensive to duplicate data; therefore, we assume that  $k$  is bounded by  $O(\frac{n}{\log n})$ . In addition, the bound on the workload,

$U$ , is usually represented by a 32-bit integer. To summarize, the total execution time becomes  $O(n \log n)$  when  $k$  is bounded by  $O(\frac{n}{\log n})$ .

**Theorem 2** *The BinSearch algorithm finds the optimal replica set for MinMaxLoad in  $O(n(\log n + \log U))$  time, where  $n$  is the number of tree nodes in the data grid,  $U$  is the maximum workload on the leaves, and the number of replicas is  $O(\frac{n}{\log n})$ . If there is a constant bound on  $U$ , the cost is  $O(n \log n)$ .*

## 5 The Constrained Model

The constrained model is similar to the unconstrained model, except that each request has a *range limit*. The range limit serves as a locality assurance, which means the request must be served by a replica, or the hub, within a fixed number of hops towards the root. For example, in Figure 8, if a user at node  $a$  were to specify a range limit 2 or more, he/she would be able to access the data at node  $c$ . If the range limit from  $a$  is 1 instead, the request fails, since a server is not placed at either  $a$  or  $b$ . Formally, we define that a request can *reach* a server if the number of communication links between it and the nearest replica on the path to the hub is no more than its range limit.

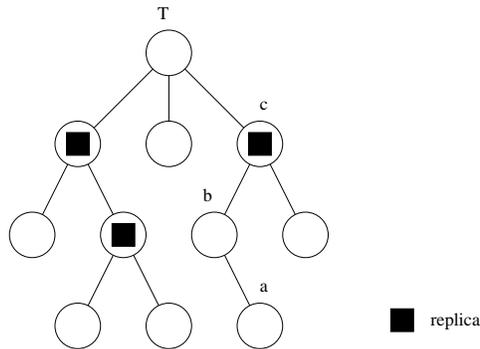


Figure 8: A data grid tree  $T$ .

The added range constraint complicates our goal of developing an efficient replica placement strategy. We must now determine where to place a given number of replicas so that the amount of data a replica server has to handle is minimized, under the constraint that all requests can find a replica within their specified range limits. Similarly, if we fix the total workload a replica server can handle, then we need to decide the minimum number of replicas and the best locations to place them.

Next, we formally define the goals of our replica placement strategy. Let  $v$  be a leaf in a data grid tree; then  $w(v)$  is the workload of  $v$ , and  $l(v)$  is its range limit. For ease of explanation, we only allow the leaves to issue data requests. However, it is trivial to generalize the results of this paper to cases where the internal nodes can also issue data requests.

Let  $R$  be a subset of  $V$ , and let a replica be placed on each node of  $R$ . Based on the data grid access model described earlier, we define the server of each leaf  $v$  as

the first node in  $R$  that  $v$  encounters when its request travels up towards the root of  $T$ . This server node is denoted by  $s_R(v)$ . The *workload* on a tree node  $v$  is defined as the sum of the data requests for which  $v$  is the server, i.e.,  $w_R(v) = \sum_{s_R(l)=v} w(l)$ . However, if the distance between a leaf  $v$  and its server  $s_R(v)$  is greater than its range limit  $l(v)$ , the workload of the server is set to infinity.

From the definitions above, we can define that a replica set  $R$  is *range feasible* if and only if none of the tree nodes has an infinite workload, i.e., every request can reach a server (or the hub) within its range limit. In addition, let the maximum workload induced by  $R$  be the maximum workload on all nodes of  $R$  and the hub. We include the hub because any data requests not serviced by  $R$  will be serviced by the hub. A replica set  $R$  is *workload  $W$  feasible* if and only if the maximum workload of the tree nodes due to  $R$  is no more than  $W$ . Note that this definition implies that a workload  $W$  feasible replica set *is* also range feasible.

Our objective is to solve the two problems, **MinMaxLoad** and **FindR**, in this constrained model.

## 6 Algorithms for the Constrained Model

~~This section describes~~ the algorithms used to solve **MinMaxLoad** and **FindR**. Similar to the case in the unconstrained model, we will solve **FindR** first, and then use that algorithm to solve **MinMaxLoad**.

### 6.1 FindR

The **FindR** problem can be stated as follows. Given a data grid tree  $T$ , the workload and the range limit on its leaves, and the maximum workload  $W$ , find a workload  $W$  feasible replica set  $R$  with *minimum* cardinality. We use  $m(T, W)$  to denote this minimum cardinality. We also define that a replica set  $R$  is *optimal* for  $T$  and workload  $W$  if  $R$  is workload  $W$  feasible, (with only  $m(T, W)$  replicas), and it minimizes the workload *on the hub*. Hereafter, we do not use “for workload  $W$ ” when the context clearly indicates the workload bound.

The complication introduced by the range limit can be easily understood by the fact that Lemma 1 is not valid in the constrained model. In the unconstrained model, we can determine where to put a replica by a greedy method, since Lemma 1 guarantees that it is always safe to select the heaviest request. However, in the constrained model there is a conflict between the workload and the range limit. If the heaviest request has a large range limit, choosing the second heaviest request that has a smaller range limit may be the best solution, since it can travel further up the tree to find a replica.

#### 6.1.1 Contribution function

We first define several terminologies. Consider a data grid tree  $T$  with  $r$  as the root. Let  $v$  be a node in  $T$ ,  $t(v)$  be the subtree rooted at  $v$ , and  $t'(v) = t(v) - v$ , i.e., the forest of subtrees rooted at  $v$ 's children. Also, let  $a(v, i)$  denote the  $i$ -th ancestor of node  $v$  while it is traveling toward the root of  $T$ .

We now define a contribution function  $C$ .  $C(v, i)$  indicates the minimum workload on node  $a(v, i)$  contributed by  $t(v)$ , by placing  $m(t(v), W)$  replicas in  $t'(v)$  and none on  $a(v, j)$  for  $0 \leq j \leq i$ . By definition  $C(v, 0)$  is the workload on a node  $v$  due to an optimal replica set for  $t(v)$ . Note that if there is no replica set for  $t(v)$  with cardinality  $m(t(v), W)$  that could control the workload on  $a(v, i)$  within  $W$ , then  $C(v, i)$  is set to infinity; for example, if  $v$  is a leaf,  $C(v, i)$  is  $w(v)$  when  $i \leq l(v)$ , and infinity when  $i > l(v)$ .

Figure 9 illustrates an example of the  $C$  function. The optimal replica set requires three replicas for  $T$  when the workload  $W$  is 60, i.e.,  $m(t(r), 60) = 3$ . The replicas should be placed at leaves  $a$ ,  $b$ , and  $c$  to minimize the workload on  $r$  to  $C(r, 0) = 35$ . However, if we try to minimize the workload on  $s = a(r, 1)$  by placing just three servers, we can only achieve  $C(r, 1) = 55$  by placing replicas at nodes  $a$ ,  $b$ , and  $e$ . We need to place a replica on node  $e$ , since its range limit is only 1. Now, with regard to the workload on  $t = a(r, 2)$ , as we can not limit its workload to 60 by placing only three replicas in  $t'(r)$ ,  $C(r, 2)$  is set to infinity.

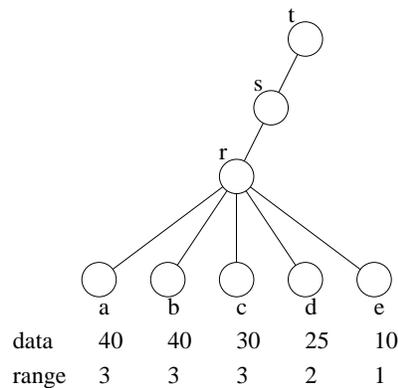


Figure 9: The contribution function.

### 6.1.2 Bottom-up Computation

Here, we describe a bottom-up process for computing the  $C$  and  $m$  functions for every node in a data grid tree. By definition, if  $v$  is a leaf,  $C(v, i)$  is  $w(v)$  when  $i \leq l(v)$ , and infinity otherwise. Now we want to compute the  $C$  and  $m$  functions for an internal node  $v$  that has children  $v_1, \dots, v_n$ . Since the process is bottom-up, we assume that all the  $C$  and  $m$  functions of  $v_1, \dots, v_n$  are known. The following theorem establishes the relation between the optimal replica sets for a tree and any of its subtrees.

**Theorem 3** *Consider a data grid tree  $T$ , a node  $v$  in  $T$ , and a workload  $W$ . There exists an optimal replica set  $R$  for  $T$  with workload limit  $W$  so that  $|R \cap t'(v)| = m(t(v), W)$*

**Proof.** By definition,  $t(v)$  requires at least  $m(t(v), W)$  replicas to be placed in  $t'(r)$  to ensure that the workload on  $v$  is within  $W$ . As a result, there does not exist an optimal replica set  $R$  for  $T$  that could place less than  $m(t(v), W)$  replicas in  $t'(v)$ .

Now, if an optimal replica set for  $T$  places more than  $m(t(v), W)$  replicas in  $t'(v)$ , it places at least  $m(t(v), W) + 1$  replicas. In such a case, when we construct an optimal replica set for  $T$ , we simply place  $m(t(v), W)$  replicas according to an optimal replica set for  $t(v)$ , and place one extra replica on  $v$ . The resulting new replica set for  $T$  does not increase the workload; therefore, it is also an optimal solution for  $T$ . ■

Theorem 3 suggests that for a node  $v$  with children  $v_1, \dots, v_n$ , there exists an optimal replica set  $R$  with workload limit  $W$  such that  $|R \cap t'(v_j)| = m(t(v_j), W)$  for  $1 \leq j \leq n$ . As a result, to find the optimal replica set for  $t(v)$ , we need to place sufficient replicas on  $v_j$ 's nodes such that the sum of their  $C(v_j, 1)$  is at least  $\sum_{1 \leq j \leq n} C(v_j, 1) - W$ . This can be easily done by repeatedly placing a replica on the remaining  $v_j$  that has the largest  $C(v_j, 1)$  until the workload of  $v$  is within  $W$ . Let this set of  $v_j$ 's be  $e(v, 0)$ , the children of  $v$ , each of which must be assigned a replica to minimize the workload on  $a(v, 0) = v$ . We now have  $m(t(v), W) = \sum_{1 \leq j \leq n} m(t(v_j), W) + |e(v, 0)|$ , and  $C(v, 0) = \sum_{v_j \notin e(v, 0)} C(v_j, 1)$ .

After determining  $m(v, W)$ , we want to compute  $C(v, i)$  for  $i > 0$ .

**Theorem 4** Consider a data grid tree  $T$ , a node  $v$  in  $T$  with children  $v_1, \dots, v_n$ , and a workload  $W$ . There exists a replica set  $R$  so that  $|R| = m(T, W)$ ,  $R$  minimizes the total workload induced by  $R$  from  $t'(v)$  on  $a(v, i)$  for  $i \geq 1$ , and  $|R \cap t'(v_j)| = m(t(v_j), W)$ .

**Proof.** This proof is similar to that of Theorem 3. Consider Figure 10. First,  $|R \cap t'(v_j)|$  could not possibly be less than  $m(t(v_j), W)$ ; otherwise, the workload on  $v_j$  would already be greater than  $W$ . If  $R$  has more than  $m(t(v_j), W)$  replicas in  $t'(v_j)$ , we can only place  $m(t(v_j), W)$  according to any optimal replica set for  $t(v_j)$ , and place one replica on  $v_j$ . The resulting replica set would not increase the workload of  $a(v, i)$ . ■

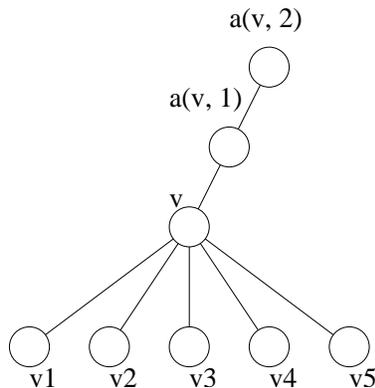


Figure 10: The computation of contribution function  $B(v, i)$  for  $i > 0$ .

An important implication of Theorem 4 is that when we compute  $C(v, i)$  for  $i > 0$ , we can be certain that there exists a set of replicas that can minimize the total workload on  $a(v, i)$  with a *known* number of replicas in each  $t'(v_j)$  (i.e.,

$m(t(v_j), W)$ ). By definition,  $C(v, i)$  is the the minimum workload on the ancestor  $a(v, i)$  contributed by  $t(v)$  by placing  $m(t(v), W)$  replicas in  $t'(v)$ , and none at  $a(v, j)$  for  $0 \leq j < i$  (see Figure 10 for an illustration). Since we know the number of replicas in each  $t'(v_j)$ , we know there exists an  $R$  that can minimize the total workload  $a(v, i)$  by placing extra  $m(t(r), W) - \sum_{1 \leq j \leq n} m(v_j, W)$  replicas among  $r_i$ . Now, it is clear that by choosing  $m(t(r), W) - \sum_{1 \leq j \leq n} m(v_j)$   $v_j$ 's ~~that have~~ the largest  $C(v, i+1)$  (denoted as  $e(v, i)$ ), we can derive  $C(v, i) = \sum_{v_j \in e(v, i)} C(v_j, i+1)$ .

### 6.1.3 Top-down replica placement

We now place replicas from the top to the bottom of the tree recursively. Consider a node  $v$  with  $n$  children  $v_1, \dots, v_n$ . Our goal is to place replicas such that the workload on  $a(v, i)$  is minimized by placing  $m(v, W)$  replicas in  $t'(v)$ ; therefore, our recursion starts from the root with  $i = 0$ . From the discussion in Section 6.1.2, we know we can accomplish this by placing replicas in  $e(v, i)$  – the subset of  $\{v_1, \dots, v_n\}$ , each of which has to be assigned a replica in order to minimize the workload on  $a(v, i)$ .

We consider two cases. In the first, we consider a set of  $v_j$ 's in  $e(v, i)$ . Let  $A$  denote the set of children of these  $v_j$ 's. We can start the recursion from each node of  $A$  with  $i$  set to 0, since we know that every node in  $e(v, i)$  now has a replica. In the second case, we consider the set of  $v_j$ 's that are *not* in  $e(v, i)$ . From Theorem 4, we know that if we want to minimize their contribution to  $a(v, i)$ , we just need to focus on those replica sets that have  $m(v_j, W)$  replicas in  $t'(v_j)$ . In addition, we know we can assume that there will be *no* replica in any  $v_j'$ . As a result, we simply retrieve  $e(v_j, i+1)$  – the subset of  $v_j$ 's children that should be assigned an extra replica to minimize the workload of  $a(v, i)$ .

The pseudo code of this recursive top-down replica placement procedure is given in Figure 11. The parameter  $i$  indicates the level of recursion towards the node whose workload we want to minimize.

```

Place-replica(v, i)
{
    if i is a leaf
        return;

    place a replica on each node of e(v, i);
    for each child c of v {
        if c is in e(v, i)
            Place-replica(c, 0);
        else
            Place-replica(c, i + 1);
    }
}

```

Figure 11: The pseudo code of the recursive top-down replica placement.

Consider the example in Figure 12. The recursion starts at  $a$  with  $i = 0$ . Suppose ~~we know that~~ we need to place replicas on  $b$  and  $c$ . Now we only need to recursively

perform the top-down replica placement at the children of  $b$  and  $c$ , namely  $f, g, h$ , and  $i$ . After placing the replicas on nodes  $b$  and  $c$ , we know that replicas will not be placed at  $d$  and  $e$ . Now consider the three subtrees of  $d$ . We know there exists a replica set that can minimize the contribution to  $a$ , with  $m(j, W)$  replicas in  $t'(j)$ ,  $m(k, W)$  replicas in  $t'(k)$ , and  $m(l, W)$  replicas in  $t'(l)$ . We also know that this replica set has *no* replica in  $d$ . As a result, we simply retrieve  $e(d, 1)$  – the subset of  $\{j, k, l\}$  that should be assigned an extra replica to minimize the workload of  $a$ .

We call this replica-placement algorithm the *PlaceReplica*.

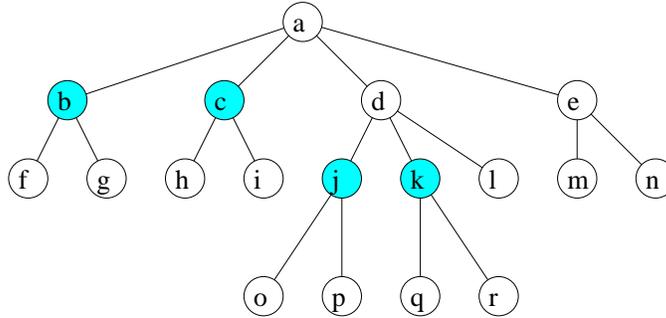


Figure 12: An example of top down replica placement. A shaded tree node indicates a replica.

#### 6.1.4 Time complexity analysis

We analyze the time complexity of *PlaceReplica* by focusing on the bottom-up computation of the  $C$  function, since it dominates the total computation time. For each tree node  $v$ , we need to compute its  $C(v, i)$  up to  $i = L$ , where  $L$  is the maximum range limit of all the nodes. Let the number of children of  $v$  be  $n_v$  then the computation requires  $n_v \log n_v$ , if we sort the  $C$  function values of all of  $v$ 's children. Note that this requires the values to be sorted  $L$  times since a child,  $v_j$ , with a larger  $C(v_j, i)$  function value than another child,  $v_k$ , does not mean that  $v_j$  will have a larger  $C$  function value than  $v_k$  for other values of  $i$ . As a result, the computation cost of  $C$  for  $v$  is  $L n_v \log n_v$ . The total cost of computing the  $C$  functions of all nodes is therefore  $L N \log N$ , where  $N$  is the number of nodes in the tree.

## 6.2 MinMaxLoad

We now derive the *BinSearch* algorithm for the *MinMaxLoad* problem. *BinSearch* finds the replica set by “guessing” the maximum workload  $W$  with a binary search, as in the unconstrained model. Let  $U$  be the workload upper bound for every leaf such that the total workload is bounded by  $O(NU)$ . It is easy to see that after  $O(\log N + \log U)$  *BinSearch* calls, we can find the smallest value of  $W$  such that  $k$  replicas are sufficient. The total execution time of *BinSearch* is therefore  $O(\log N + \log U)(LN \log N)$ . Because the bound on the workload  $U$  is usually represented by a 32-bit integer, the total execution time can be bounded by  $O(LN \log^2 N)$ .

**Theorem 5** *The BinSearch algorithm finds the optimal replica set for MinMaxLoad in  $O(\log N + \log U)(LN \log N)$  time, where  $N$  is the number of tree nodes in the*

data grid,  $L$  is the maximum range limit. and  $U$  is the maximum workload. When  $U$  is a bounded constant, the time complexity is  $O(LN \log^2 N)$ .

## 7 Conclusion

We have addressed three issues related to placing database replicas in Data Grid systems with locality assurance, namely load balancing, the minimum number of replicas, and guaranteed service locality. Each request specifies the workload it requires, and the distance within which a replica must be found. We propose efficient algorithms that 1) select strategic locations to place replicas so that the workload of the replicas is balanced; 2) use the minimum number of replicas if the service capability of each replica is known; and 3) guarantee the service locality specified by each data request.

We have also formulated two problems: **MinMaxLoad** and **FindR**, and derive efficient algorithmic solutions for them. Based on an estimation of data usage and the locality requirement of various sites, our algorithm efficiently determines the locations of replicas if both the number of replicas and the maximum allowed workload for each replica have been determined. Then, another algorithm determines the number of replicas needed to ensure that the maximum workload on every replica is below a certain threshold.

~~One unsolved question about the replica placement problem is:~~ How can the replica location be determined when the network is a general graph, instead of a tree? It is possible that we may need to consider other graphs, (e.g., planar graphs), and derive efficient algorithms for them. Second, in the current hierarchical Data Grid model, all the traffic may reach the root if it is not serviced by a replica. This makes the design of an efficient replica placement algorithm more complex when network congestion is one of the objective functions to be optimized. For example, the network bandwidth in a grid system may be limited. A good replica placement strategy must ensure that, in addition to those issues that have been resolved by this paper, the traffic going through every network link will not exceed its maximum bandwidth. Finally we may need to consider the heterogeneity in server capability. For example, if we would like to place one major replica server and two minor replica servers. A major server can sustain a much heavier workload than a minor server. ~~Then, the question is how to place the replicas such that all the service quality can be guaranteed.~~

**Acknowledgments** The authors wish to thank Mr. Meng-Zong Tsai from National Taiwan University for many helpful discussions, and the National Center for High-Performance Computing for providing resources under the "Taiwan Knowledge Innovation National Grid" project.

## References

- [1] J. H. Abawajy. Placement of file replicas in data grid environments. In *ICCS 2004, Lecture Notes in Computer Science 3038*, pages 66–73, 2004.

- [2] M. M. Bae and B. Bose. Resource placement in torus-based networks. *IEEE Transactions on Computers*, 46(10):1083–1092, October 1997.
- [3] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, (23):187–200, October 2000.
- [4] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe. Wide area data replication for scientific collaborations. In *In Proceedings of the 6th International Workshop on Grid Computing*, November 2005.
- [5] I. Cidon, S. Kutten, and R. Soffer. Optimal allocation of electronic content. *Computer Networks*, 40(2):205–218, 2002.
- [6] W. B. David. Evaluation of an economy-based file replication strategy for a data grid. In *International Workshop on Agent based Cluster and Grid Computing*, pages 120–126, 2003.
- [7] W. B. David, D. G. Cameron, L. Capozza, A. P. Millar, K. Stocklinger, and F Zini. Simulation of dynamic grid rdeplication strategies in optorsim. In *In Proceedings of 3rd Intl IEEE Workshop on Grid Computing*, pages 46–57, 2002.
- [8] M.M. Deris, Abawajy J.H., and H.M. Suzuri. An efficient replicated data access approach for large-scale distributed systems. In *IEEE International Symposium on Cluster Computing and the Grid*, April 2004.
- [9] Worldwide LHC Computing Grid. <http://lcg.web.cern.ch/lcg/>.
- [10] W. Hoschek, F. J. Janez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data grid project. In *In Proceedings of GRID Workshop*, pages 77–90, 2000.
- [11] M. Jaeger and J. Goldberg. Capacitated vertex coveringcapacitated vertex covering. *Transportation Science*, 28(2), 1994.
- [12] X. Jia, D. Li, X.-D. Hu, W. Wu, and D.-Z. Du. Placement of web-server proxies with consideration of read and update operations on the internet. *Comput. J.*, 46(4):378–390, 2003.
- [13] K. Kalpakis, K. Dasgupta, and O. Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):628–637, 2001.
- [14] M. Korupolu, C. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. *Journal of Algorithms*, 38(1):260–302, 2001.
- [15] H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman. Data replication strategies in grid environments. In *In Proceedings of 5th International Conference on Algorithms and Architecture for Parallel Processing*, pages 378–383, 2002.
- [16] R. Moore, C. Baru, R. Marciano, A. Rajasekar, and M. Wan. *I. Foster and C. Kesselman edited, The Grid: Blueprint for a Future Computing Infrastructure*, chapter Data intensive computing. Morgan Kaufmann PUblishers, 1999.

- [17] K. Ranganathan, A. Iamnitchi, and I.T. Foste. Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In *In 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 376–381, 2002.
- [18] K. Ranganathana and I. Foster. Identifying dynamic replication strategies for a high performance data grid. In *In Proceedings of the International Grid Computing Workshop*, pages 75–86, 2001.
- [19] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and object replication in data grids. In *In 10th IEEE Symposium on High Performance and Distributed Computing*, pages 305–314, 2001.
- [20] X. Tang and J. Xu. Qos-aware replica placement for content distribution. *IEEE Transactions on Parallel and Distributed Systems*, 16(10), October 2005.
- [21] N.-F. Tzeng and G.-L. Feng. Resource allocation in cube network systems based on the covering radius. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):328–342, April 1996.
- [22] O. Unger and I. Cidon. Optimal content location in multicast based overlay networks with content updates. *World Wide Web*, 7(3):315–336, 2004.
- [23] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.*, 16(1):181–205, 1991.