Optimal Replica Placement Strategy for Hierarchical Data Grid Systems

Pangfeng Liu
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 Sincia
Taipei, Taiwan, R.O.C.
wuj@iis.sinica.edu.tw

Abstract

Grid computing is an important mechanism for utilizing distributed computing resources. These resources are distributed in different geographical locations, but are organized to provide an integrated service. In order to speed up data access efficiency data grid systems replicate essential data in multiple locations, so that a user can access the data from a site in his vicinity. This paper studies replica placement in Data Grid systems, taking into account several important issues described below. First, the replicas should be placed in proper server locations so that the workload on each server is balanced. Second, we choose the optimal number of replicas to balance the data access efficiency, and the expensive maintenance costs for multiple copies of data. Clearly, optimizing access cost of data requests and reducing the cost of replication are two conflicting goals. Finding a good balance between them is a challenging task. We propose efficient algorithms for selecting optimal locations for placing the replicas so that the workload among these replica is balanced. Also when given the data usage from each user site and the maximum workload allowed for each replica server, our algorithm efficiently determines the minimum number of replicas required, as well as their locations.

1 Introduction

Grid computing is an important mechanism for utilizing distributed computing resources. These resources are distributed in different geographical locations, but are organized to provide an integrated service. A grid system can provide computing resources so that users at different locations can utilize the CPU cycles of remote sites. In addition, users can access important data that are available only in several locations, without the overheads of replicating them locally. These services are provided by an integrated grid service platform so that user can access the resource transparently and effectively.

One class of grid computing and the focus of this paper is Data Grids that provide geographically distributed storage resources to large computational problems that require evaluating and managing large amount of data [3, 8, 11]. For example, the scientists working on bioinformatics may need to access human gnome databases on different remote locations. These databases have tremendous amount of data, so the cost of maintaining a local copy on each site that needs the data is extremely expensive. In addition, these databases are mostly read-only, since they are the input data to the applications for various purposes, such as benchmarking, identification, and classification. With the high latency of wide-area network that underlies most Grid systems, and the need to access/manage several petabytes of data in Grid environments, data availability and access optimization becomes key challenges to be addressed.

An important technique to speed up data access for Data Grid systems is to replicate the data in multiple locations, so that a user can access the data from a site in his vicinity. It has been shown that data replication not only reduces access costs, but also increase data availability in many applications [8, 12, 10]. There is a fair amount of work on data replication in Grid environments. However, most of the existing work focused on infrastructures for replication and mechanisms for creating/deleting replicas [4, 6, 5, 7, 8, 10, 13, 12, 14]. We believe that, in order to obtain maximum gains of replication, a strategic placement of the replicas is necessary.

A number of early works address placement of data replicas in parallel and distributed systems with regular network topologies such as hypercubes , torus, rings, and trees. These networks posses many attractive mathematical properties that enable the design of simple and robust placement algorithms [2, 9, 15]. These algorithms, however, cannot be directly applied to Data Grid systems due to hierarchical network structures and special data access patterns in Data Grid systems that are not common in traditional parallel systems. An initial work on replica placement

Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)
0-7695-2585-7/06 $20.00 © 2006 IEEE

for Data Grids was reported in [1]. The author proposed a heuristic algorithm, named `Proportional Share Replication`, for the placement problem. However, the algorithm does not guarantee to find the optimal solution.

In this paper, we study replica placement in Data Grid systems, taking into account several important issues described below. First, the replicas should be placed in proper server locations so that the workload on each server is balanced. Another important issue is choosing the optimal number of replicas. The denser the distribution of replicas is, the shorter the distance a client site needs to travel to access a data copy. However, maintaining multiple copies of data in Grid systems is expensive, and therefore, the number of replicas should be bounded. Clearly, optimizing access cost of data requests and reducing the cost of replication are two conflicting goals. Finding a good balance between them is a challenging task.

We propose efficient algorithms for selecting optimal locations for placing the replicas so that the workload among these replica is balanced. Also when given the data usage from each user site and the maximum workload allowed for each replica server, our algorithm efficiently determines the minimum number of replicas required, as well as their locations.

The rest of the paper is organized as follows. Section 2 describes our data grid model, and formally define our replica placement problem. Section 3 presents our replica placement algorithms, and provides theoretical analysis for them. Section 4 concludes and addresses several open questions and future works.

2 Model

We first describe our data grid model. We will consider hierarchical Data Grid model in this paper, due to its simplicity and close resemblance to the hierarchical management, usually found in a grid system. 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 nodes except the hub r. All the tree leaves are local sites where user can access databases stored in this data grid system.

A user of a local site at the leaf access a database as follows. First he tries to locate the database replica locally. If the database replica is not present, he goes to the parent node up the tree to find if a replica is there. Namely the user request goes up the tree and uses the first replica encountered along the path towards the root. If there is no replica along the way, the hub will server this request.

Now we formally define the goals of our replica placement strategy. Let l be a leaf node from the set of all leaves L. Let $w(l)$ be the amount of data requests from 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 the cases where all the tree nodes, including the internal nodes, can request data. According to the data grid access model above we define the *workload* on a particular tree node after the replicas are placed as follows. Let T be a data grid system tree, N be the set of nodes in T, and R be a subset of N, with a replica placed in every node of R. The workload for a node n of N, denoted as $f(n)$, is defined recursively as follows.

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

The maximum workload of R is the maximum workload from all nodes of R and *the hub*. The reason that we include the hub is that all the data requests unanswered by replicas will eventually be answered by the hub. Now we formally define our problems.

- *MinMaxLoad*: given the number of replica k, find a R so that the maximum workload is minimized.
- *FindR*: given the amount of data a replica or the hub can serve (D), find the minimum cardinality R so that the maximum workload is no more than D.

3 Algorithms

The problem `FindR` 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 R of tree nodes with cardinality no more than k to place the replica so that the workload on every $r \in R$ and the hub is no more than D. All these R sets are refereed to as "feasible". A feasible R is *optimal* if it minimizes the workload on the hub.

To make the discussion easier, 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 workload of its children. If a tree node has a workload greater than D, we call it a *heavy* node, or it is a *light* node. If a light node has a heavy parent, we call it a *critical* node.

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

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.*

By Observation 1 and Lemma 1, we derive a baseline algorithm `Feasible` for `FindR`. Given the tree T, the replica capacity D, and the number of replicas allowed k, `Feasible` determines whether there is a feasible replica

set with cardinality k or less, by repeatedly picking the critical leaf that has the maximum workload for at most k times. If the hub becomes light, a solution is found.

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 turn some of the heavy ancestors into light nodes. These ancestors now are non-critical and should be removed.

The time complexity analysis is as follows. Let n be the number of tree nodes. It only takes $O(n)$ to compute the workload when no replica is placed, so that we can determine the category for every tree node. Also since a node can be removed only 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 for every maximum child we pick from the bottom of the tree. The total cost could be as high as $\Omega(kn)$.

We improve our baseline algorithm `Feasible` so that the updating cost is not prohibitive. We introduce a concept called *lazy update* for this purpose. The idea of lazy update is to use a *deduction value* on an internal node n (denoted by $d(n)$) to keep track of the amount of workload that should be removed from n, and all of its ancestors.

The lazy updating works as a depth first traversal on the heavy nodes. When the traversal reaches a heavy node with only critical children, it picks the child (denoted as c) with the maximum workload to place a replica, as the baseline algorithm `Feasible` does, and starts subtracting $w(c)$ (the workload of c) along the path from c back to the hub. See Figure 1 for an illustration. If any of the ancestors along the way becomes non-critical, it is removed as in the baseline algorithm `Feasible` (node g and h in Figure 1). When the lazy update reaches a heavy node (node e in Figure 1) that now becomes critical after reducing its workload by $w(c)$, it will not try to deduct $w(c)$ from all ancestors of e. Instead the lazy update will increases the deduction on e's parent (denoted by f in Figure 1) by $w(c)$, and starts the traversal from f. Note that with the help of this "deduction", we eliminate the duplicated work of deducting workloads from those tree nodes that are along the same path from a leaf to the hub. As a result when lazy updating deducts workload from e, it must add the deduction of f by $w(c)$.

Now we analyze the time complexity of lazy update, especially on the deduction part. Each node can only be removed once, so the cost of removal is bounded by $O(n)$, where n is the number of tree nodes. When a replica is placed at a critical node c, there could be three kinds of value updating on the ancestors of c. First, an ancestor could be removed since after deducting $w(c)$ it becomes light nodes (node g and h in Figure 1). Second, an ancestor could become critical (node e) after $w(c)$ is deducted from its workload. Third, an ancestor will add $w(c)$ to its deduction value. Since an ancestor can only be removed once, considering all the replicas, the total costs of the fist kind of updating is bounded by $O(n)$. Also each replica placing will incur the second and the third kinds of updating once, so the total cost from them is bounded by $O(k)$.

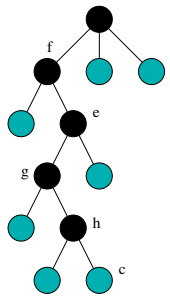

Figure 1. An execution scenario of the lazy update.

We now analyze the cost of picking the critical node with the maximum workload among its siblings. It is easy to see that there could be at most k such selections since 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. It is easy to see that we only need to keep the k largest children for every internal node, since we only have at most k replicas. We achieve this goal with a sorted list with at most k elements. The overall list maintenance time is bounded by $O(k \log k)$.

The only remaining question is how to initialize the sorted list for every internal node. If the value of k is small, we simply choose the maximum k children repeatedly for every parent, with a total time $O(kn)$. 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. This gives every parent the k largest of its children, therefore the initialization cost is $O(\min(kn, n \log n))$.

Now we add all the costs together. The time to initialize the sorted list is $O(\min(nk, n \log n))$, the time to maintain the sorted lists is $O(k \log k)$, and the time for tree traversal and updating 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 algorithm* `LazyUpdate` *finds the optimal replica set for* `FindR` *in time $O(nlogn)$, where n is the number of tree nodes in the data grid.*

We now derive an algorithm `BinSearch` for the `MinMaxLoad` problem. The algorithm `BinSearch` finds the replica set by "guessing" the maximum workload with a binary search. We assume that all the workload are integers and there is an upper bound U on the workload for every

node, therefore the total amount of workload is bounded by $O(nU)$. It is easy to see that after $O(logN + logU)$ calls of `LazyUpdate`, we will be able to find the smallest value of D by which only k replicas suffice.

Now 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 done once in `BinSearch`, since the tree is the same throughout the binary search. Each iteration of `LazyUpdate` takes $O(k \log k + n)$, and 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 algorithm* `BinSearch` *finds the optimal replica set for* `MinMaxLoad` *in time* $O(n(\log n + \log U))$*, where* n *is the number of tree nodes in the data grid,* U *is the maximum workload on the leaves, and the number of replica is* $O(\frac{n}{\log n})$*. If there is a constant bound on* U*, the cost is* $O(n \log n)$*.*

4 Conclusion

This paper addresses the issues of placing database replica in a data grid system. In particular, we give efficient algorithms for selecting strategic locations for placing the replica so that the workload among these replica is balanced. We formulate two problems: MinMaxLoad and FindR, and derive efficient algorithmic solutions for them. Based on the estimation of data usage from various sites, our algorithm efficiently determines the locations of replica if both the number of replica and the maximum workload for each replica have been determined. Another algorithm can determine the number of replica needed to ensure that the maximum amount of workload on every replica is below a certain threshold. Both algorithms run in time $O(n \log n)$, where n is the number of sites in the data grid system.

One open question for the replica placement problem is how to determine the replica location 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.

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] 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.

[6] 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.

[7] M. Deris, A. J.H., and H. Suzuri. An efficient replicated data access approach for large-scale distributed systems. In *IEEE International Symposium on Cluster Computing and the Grid*, April 2004.

[8] 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.

[9] K. Kalpakis, K. Dasgupta, and O. Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):628–637, June 2001.

[10] 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.

[11] 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.

[12] K. Ranganathan, A. Iamnitchi, and I. 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.

[13] 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.

[14] 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.

[15] 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.