A QoS-Aware Heuristic Algorithm for Replica Placement

Hsiangkai Wang #1, Pangfeng Liu #2, Jan-Jan Wu *3

#*Department of Computer Science and Information Engineering, National Taiwan University*
Taipei, Taiwan
1pangfeng@csie.ntu.edu.tw
2hsiangkai@gmail.com

**institute of Information Science, Adademia Sinia*
Taipei, Taiwan
3wuj@iis.sinica.edu.tw

Abstract— **This paper studies the QoS-aware replica placement problem. Although there has been much work on replica placement problem, most of them concerns average system performance and ignores quality assurance issue. Quality assurance is very important, especially in heterogeneous environments. We propose a new heuristic algorithm that determines the positions of replicas in order to satisfy the quality requirements imposed by data requests. The experimental results indicate that the proposed algorithm finds a near-optimal solution effectively and efficiently for algorithm can also adapt to various parallel and distributed environments.**

I. 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. 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 is to replicate the data in multiple locations so that a user can access the data from a server in his vicinity. It has been shown that data replication not only reduces access costs, but also increases data availability in many applications. [1], [2], [3]. 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], [5], [6], [7], [3], [8], [2], [9]. [4], [5], [6], [7], [3], [8], [2], [9]. We believe that, to obtain maximum benefits from replication, a strategic placement of replicas is essential.

Although there has been much work on replica placement problem [10], [11], [12], [13], [14], very few of them concerns quality of service. A large part of these work concerns the average system performance, for example, to minimize the total accessing cost, or to minimize the total communication cost, etc. Although these metrics are important in the overall system performance, they cannot meet the individual requirement adequately. Grid computing infrastructure usually consists of various type of resources and the performance of these resources are quite diverse. Moreover, different sites may have different service quality requirements according to the system performance of the sites. Therefore, quality of service is an important factor in addition to overall system performance.

An early work by Tang and Xu [15] considered the quality of service in addition to minimize the storage and update cost. The distance between two nodes is used as a metric for quality assurance. A request must be answered by a server within the distance specified by the request. Every request knows the nearest server that has the replica and the request takes the shortest path to reach the server. Their goal has been to find a replica placement that satisfies all requests without violating any range constraint, and minimize the update and storage cost at the same time. They show that this QoS-aware replica placement problem is NP-Complete for general graphs, and provide two heuristic algorithms – l-`Greedy-Insert` and l-`Greedy-Delete`, for general graph. A dynamic programming solution is given for tree topology [15].

In this paper, we study the QoS-aware replica placement problem and provide a new heuristic algorithm to decide the positions of the replicas to improve system performance and satisfy the quality requirements specified by the user simultaneously. Our algorithm efficiently computes near-optimal solutions, so that it can be deployed in various realistic environments.

The rest of this paper is organized as follows. Section II describes previous work about replica placement. Section III describes the system model and notations. Section IV presents our algorithm and time complexity analysis. Section V presents our experimental results and provides some analysis on the results. Section VI summarizes our research results and major contributions.

II. Related Works

Optimal replica placement problem has been studied extensively in the literature. The same problem has different names in different research areas. For example, it is refereed to as p-median problem in operations research, or database location problem on Internet and file allocation problem in computer science. Wolfson and Milo [14] proved that replica placement problem is NP-Complete for general graphs when read and update cost are simultaneously considered. They also provide optimal solutions for special topologies, including complete graph, tree, and ring. Tu and Xu [12] study the secure data placement problem in the same model and provide a heuristic algorithm for general graphs. Krick et al. [11] consider read, update and storage cost simultaneously in general graph, and provide an polynomial time approximation algorithm that has a constant competitive ratio. They also provide an optimal solution for tree topology in the same paper. Kalpakis, Dasgupta and Wolfson [10] consider read, update and storage cost under tree topology. Their algorithm could cope with the situations even when servers have capacity limits. They describe an $O(n^3p^2)$ dynamic programming algorithm for p replicas placed in n incapacitated servers, and an $O(n^3p^2\wedge_{max}^2)$ algorithm for capacitated servers, where $\wedge_{max}$ denotes the maximum capacity among all servers. Unger and Cidon [13] provide a more efficient algorithm to find the optimal placement under the same model, with only $O(n^2)$ time, where n is the number of servers. However, the algorithm in [13] cannot deal with server capacity limits. There are other algorithms that provide optimal solutions under simpler models for tree topology [16], [17].

Although there has been a lot of work studying the optimal replica placement problem, very few of them concern quality of service. The goal in these efforts is usually to minimize the total replication cost. The replication cost may contain read, update and storage cost, depending on the system model. The objective has usually been to improve the average system performance, without any quality-of-service quarantees. An early effort by Tang and Xu [15] suggested a QoS-aware replica placement problem to cope with the quality-of-service issues. Every edge uses the distance between the two end-points as a cost function. The distance between two nodes is used as a metric for quality assurance. A request must be answered by a server that is within the distance specified by the request. Every request knows the nearest server that has the replica and the request takes the shortest path to reach the server. Their goal has been to find a replica placement that satisfies all requests without violating any range constraint, and minimize the update and storage cost at the same time. They show that this QoS-aware replica placement problem is NP-Complete for general graphs, and they provide two heuristic algorithms, called l-`Greedy-Insert` and l-`Greedy-Delete`, for general graph, and a dynamic programming solution for tree topology.

1) l-`Greedy-Insert`. l-`Greedy-Insert` starts with an empty replication set R, and inserts replicas into R until all servers' QoS requirements are satisfied. In the first step, the algorithm selects $(l+1)$ replicas that maximize the *normalized benefits* among all possible locations. Normalized benefits is defined as the increased number of satisfied servers divided by the increased replication cost due to the selection. In each step, we examine all possible replacement, each of them replaces l replicas with some $(l+1)$ replicas, and choose the one that maximizes the normalized benefits. Note that the removed replicas and the inserted replicas can overlap.
2) l-**Greedy-Delete**. l-`Greedy-Delete` works the opposite way as the l-`Greedy-Insert`. We begin with having a replica in every node, then it deletes replicas whose deletion maximizes the replication cost reduction until there is no replica that can be deleted. In the first step, l-`Greedy-Delete` removes the $(l+1)$ replicas whose deletion maximizes replication cost reduction without violating the QoS requirements. In each subsequent step, the algorithm examines all possibilities of replacing $(l+1)$ replicas with l replicas without violating QoS requirements, and chooses the one that maximizes replication cost reduction. We repeat the process until there is no possible alternative left.

The time complexity of l-`Greedy-Insert` and l-`Greedy-Delete` is $O(|V|^3)$ for $l=0$ and $O(|V|^{2l+2})$ for any $l>0$ [15]. The time complexity for the $l=0$ case is due to shortest path computation. There is a trade-off between the time complexity and the quality of solution on l value. Although the time complexity is a polynomial function of the number of nodes, the execution time of these two algorithms are very slow in practice even when $l=1$.

Since l-`Greedy-Insert` starts by inserting replicas into a empty replica set, and l-`Greedy-Delete` starts by deleting replicas from a full replica set, the execution time of these two algorithms depends heavily on the number of replicas in the optimal solution. If the optimal solution has very few replicas, l-`Greedy-Insert` becomes more efficient than l-`Greedy-Delete`. On the other hand, l-`Greedy-Delete` is much more efficient when the optimal solution contains a lot of replicas.

Won, Indranil and Klara proposed a simpler formulation about QoS-aware replica placement problem [18]. Their goal was to minimize the number of replicas in the system. They did not consider update cost and assumed each server has identical storage cost. They propsed a simpler and quicker algorithm to find the solution and gave another proof of NP-Completeness property of this problem. I describe their algorithm as follows. Let A be the all-to-all shortest path matrix. Entry (i, j) denotes the shortest path distance between server i and server j. B is an equal size matrix as A. Every entry in row i in B has identical value that represent the quality requirement of server i. We then examine every entry of $A - B$. If the entry is less than or equals to 0, set the entry to 1, otherwise, set the entry to 0. Let the 0-1 matrix as C. Column j in C represents which servers are covered by server j. If we find a set of columns which cover all rows in the matrix, we find a replica placement which satisfies all servers' requests. Every iteration in the algorithm, we select the column j (server j) with most rows not covered so far. I call this algorithm `Greedy MSC` (Greedy Minimum Set Covering).

In this paper, we propose a simple heuristic algorithm to find a near optimal placement very efficiently – our algorithm finds a near-optimal solution in less than two seconds even when the number of servers is over 1000.

III. System Model

This section describes our network model. The network is represented by an undirected graph $G = (V, E)$, where V is the set of servers, and $E \subseteq V \times V$ denotes the set of network links among the servers. Each link $(u, v) \in E$ is associated with a cost $d(u, v)$ that denotes the communication cost of the link. We assume that the graph is connected, so that one server can connect to any other server via a path. We define the communication cost of a path as the sum of the communication cost of the links along the path. Because we assume that a server knows where to find the nearest replica, we define $d(u, v)$ between two servers u, v to be the communication cost of the *shortest* path between them. Every server u has a storage cost, $s(u)$, that denotes the cost to put a replica on server u. The storage cost on different nodes may be different. Figure 1 is an example of our model. The numbers in the circles are server indices between 0 and $n-1$, where n is the total number of servers. The number next to a server is its storage cost. The number on a link is the communication cost of the link.

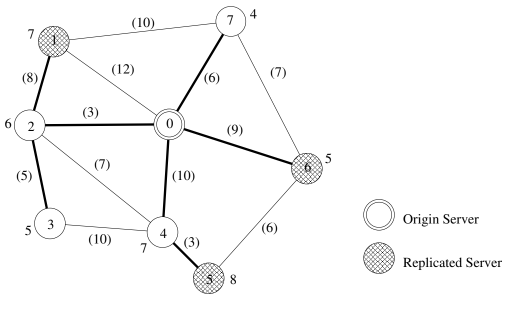

Fig. 1. An example of data replication in connected network.

Each server in the network services multiple clients, although we do not place clients into the network graph. A client sends its requests to its associated server, then the server processes the request. If the client's requests can be served by the server, i.e., the local server has the requested data, the requests will be processed locally. Otherwise, the request will be directed to the nearest server that has the replica. As a result, we assume that all requests are issued from the servers and there are only servers in the network graph. In addition, because the communication cost from the clients to servers does not affect the replication decision, we ignore the communication cost from clients to servers.

There is a special server r, called *origin server*, in the network graph. Without lose of generality, we assume that server 0 is the origin server. Initially only the origin server has the data. A *replica server* is a server that has a copy of the original data. A *replication strategy*, $R \subseteq V - \{r\}$, is a set of replica servers.

We use *replication cost* to evaluate replication strategies. The replication cost $T(R)$ of a replication strategy R is defined as the sum of the *storage cost* $S(R)$ and the *update cost* $U(R)$.

$$T(R) = S(R) + U(R) \tag{1}$$

a) Storage cost: The storage cost of a replication strategy R is the sum of all storage cost of the replica servers.

$$S(R) = \sum_{v \in R} s(v) \tag{2}$$

b) Update cost: In order to maintain data consistency, the original server r issues update requests to every replica server. The update frequency μ denotes the number of update requests issued by r per time period. We assume that there is an *update distribution tree* T, which connects all the servers in the network. For example, in our experiments, we use a shortest path tree rooted at the origin server as the update distribution tree. As in Figure 1, we use bold lines to represent the edges of the shortest path tree. The origin server r multicasts update requests through links on this tree until all the replica servers in R receive the update requests. Every node receives update requests from its parent and relays these requests to its children according to the update distribution tree.

Given the network, the update distribution tree, the update frequency μ, the update cost of a replication strategy R is defined as follows. Let $p(v)$ be the parent of node v in the update distribution tree, and T_v be the subtree rooted at node v. If $T_v \cap R \neq \emptyset$, the link $(v, p(v))$ participates the update multicast. As a result, the update cost is the sum of the communication costs from these links $(v, p(v))$. For example, in Figure 1 if the update rate is 1 and the replication strategy R is $\{1, 5, 6\}$, then the update cost is $11 + 13 + 9 = 33$.

$$U(R) = \mu \times \sum_{v \neq r,\ T_v \cap R \neq \emptyset} d(v, p(v)) \tag{3}$$

A. Service Quality Requirement

Every server u has a service quality requirement $q(u)$. The requirement mandates that all requests generated by u will be serviced by a server within $q(u)$ communication cost. We assume that every server in the network knows the nearest replica server from itself. If a request is serviced by the nearest replica server within $q(u)$, the request is *satisfied*, otherwise, the request is *violated*. If all requests in the system are satisfied, the replication strategy is called *feasible*. The QoS-aware replica placement problem is to find the feasible replication strategy such that the replication cost in Equation 1 is minimized.

For example, in Figure 1, if the quality requirement is 8 for all servers and the replication strategy is $\{1, 5, 6\}$. It is easy to verify that the replication strategy together with origin server can satisfy all requests within the network. The storage cost is $7+8+5 = 20$, the update cost is 33, so the replication cost is 53.

IV. Heuristic Algorithms

In this section, we propose a new heuristic algorithm, called `Greedy-Cover`, that finds good solutions for QoS-aware replica placement problem in general graphs. We start with definitions. The *cover set* $c(u)$ of a server u is the set of servers that are within the QoS requirement $q(u)$ from u.

$$c(u) = \{v | d(v, u) \leq q(u)\} \tag{4}$$

Each server has its own cover set. If a server $w \in c(u)$ has a replica on it, u could be satisfied by the server w. Thus, every server in $c(u)$ is a candidate server to place a replica in order to satisfy server u. We first observe that if $c(u) \subseteq c(v)$, we do not need to consider $c(v)$. If we put a replica on server $w \in c(u)$, the server w can satisfy u and v simultaneously.

We then observe that if $|c(v)| > |c(u)|$, v is more likely to be satisfied than u is. Consequently, v has more chance to be satisfied during processing other cover sets. The reason is that if $c(v)$ has more elements, it is more likely that $c(v)$ will overlap with other cover sets. Consequently v is likely to be covered by other cover sets. Our intuition is that if we place the replica so that the server with the smallest cover set is satisfied first, this replica may satisfy other servers with larger cover sets as well. That is, we may find a replication strategy with less replica servers, and the replication cost may be reduced. Based on these observations and intuition, we propose the `Greedy-Cover` algorithm.

The first step in `Greedy-Cover` is to find the cover set of each server in the network. Second, we remove all super cover sets $c(v)$ that contains some other cover set, $c(u)$. That is, if $c(u) \subseteq c(v), u \neq v$, we remove v from those servers that must be satisfied. In each subsequent step, `Greedy-Cover` chooses the smallest cover set c, examines every server s, and puts a replica on a server s in c with the highest normalized benefit. Normalized benefits is defined as the increased number of satisfied requests divided by the increased replication cost due to the selection [15]. If the newly placed replica satisfies other cover sets, these cover sets are removed. After `Greedy-Cover` updates the set of cover sets, only those unsatisfied cover sets remain. `Greedy-Cover` then chooses the smallest cover set among those remaining cover sets and repeats the process until all cover sets are satisfied. The pseudo code of algorithm `Greedy-Cover` is given in Algorithm 1.

Algorithm 1: The pseudo code of `Greedy-Cover` algorithm

```
Data: $G = (V, E)$, every node's QoS requirement
Result: feasible replication strategy
begin
    find all-pairs shortest path distance
    build shortest path tree rooted at $v_0$
    mark $v_0$ as satisfied        // because $v_0$ has origin copy
    for $i \leftarrow 0$ to $|V| - 1$ do    // this for loop builds cover set
        if $v_i$ was not satisfied by $v_0$ then
            for $j \leftarrow 0$ to $|V| - 1$ do
                if $distance[i, j] \leq v_i$'s QoS then
                    add $j$ to $v_i$'s cover set
                end
            end
        end
    end
    remove super cover sets
    while there exists unsatisfied cover sets do
        select $min\_cover\_set$ from unsatisfied cover sets
        $Max\_N \leftarrow -1$    // initialize maximum normalized benefit
        for $v_i$ in $min\_cover\_set$ do
            put a replica on $v_i$
            $N_i \leftarrow$ normalized benefits of the newly placed replica
            if $N_i > Max\_N$ then
                $Max\_N \leftarrow N_i$
                $best\_server \leftarrow v_i$
            end
            take off the newly placed replica from $v_i$
        end
        mark $best\_server$ is replicated
        remove cover sets satisfied by $best\_server$
    end
end
```

A. Time Complexity

We analyze the time complexity of the three phases of `Greedy-Cover`. In the first phase, `Greedy-Cover` finds the cover set of every server in the network. Every cover set could be identified by checking $|V|$ servers in the network. Since every server has a cover set, it takes $O(|V|^2)$ to find all the cover sets in the network.

In the second phase, `Greedy-Cover` identifies and deletes super cover sets in the network. In order to identify all the super cover sets, `Greedy-Cover` needs to check all pairs of cover sets, which have $O(|V|^2)$ possibilities. It takes $O(|V|)$ to check a pair of cover sets, so it takes `Greedy-Cover` $O(|V|^3)$ time to identify and delete the super cover sets.

In the last phase, `Greedy-Cover` inserts replicas into the network iteratively until all servers are satisfied. First, `Greedy-Cover` selects the smallest cover set, which can be done by an initial round of sorting the cover sets by size. After finding the smallest cover set c, `Greedy-Cover` calculates normalized benefits for all servers in c and puts a replica on the server with the maximum normalized benefits. Both the calculations of increased satisfied servers and increased replication cost due to a newly placed replica take $O(|V|)$ time, so it takes $O(|V|)$ time to compute the normalized benefit for a replica. The size of a cover set is $O(|V|)$ and there are $O(|V|)$ cover sets to consider in the worst case. As a result, it takes `Greedy-Cover` $O(|V|log|V| + |V|^3) = O(|V|^3)$ time to finish the last phase.

V. Performance Evaluation

This section describes our experimental results. For comparison purpose Tang and Xu [15] formulate the replica placement as an integer programming problem. They then relax the requirements for an integer solution, and consequently transform the integer program into a linear program. Since the solution of this linear program is a lower bound for the solution of the original replica placement problem, this "super" optimal solution is used as a performance measurement criteria [15]. We compare the solution from our heuristic algorithm with this super optimal solution. The ratio of cost from the heuristic algorithm to cost from the super optimal solution is referred to as *normalized replication cost.*

We now explain the process of how to obtain this super optimal solution [15]. Let $V = \{r, v_1, v_2, \ldots, v_{n-1}\}$ be the set of servers. The replica placement problem can be expressed as the following integer program.

$$\text{to minimize} \sum_{n>i>0} (s_i \times x_i + d(v_i, p(v_i)) \times y_i) \qquad (5)$$

subject to

$$n > \forall i > 0 \wedge d(v_i, r) > q(v_i), \qquad \sum_{d(v_i,v_j) \le q(v_i)} x_j \ge 1 \qquad (6a)$$

$$n > \forall i > 0, \qquad y_i \ge x_i \qquad (6b)$$

$$n > \forall i, j > 0 \wedge p(v_j) = v_i, \qquad y_i \ge y_j \qquad (6c)$$

$$n > \forall i > 0, \qquad x_i, y_i \in \{0, 1\} \qquad (6d)$$

The variable x_i is 1 if a replica is placed at server v_i, and 0, otherwise, and the variable y_i is 1 if y_i receives data update requests from its parent $p(v_i)$ in the update distribution tree T [15]. If we relax the integer requirement we have a linear program, which has a better optimal solution than the original integer program. Note that an optimal solution from the linear program may not even be a feasible solution for the integer program, but it serves as a lower bound on the total replication cost and could be used to measure how close we are from the optimum. We define *normalized replication cost* to be the the ratio between the cost from an algorithm and the cost from the linear program, and use this ratio as a performance metric.

In our experiments, the network topology was generated according to Waxman model [19]. In this model, N nodes are randomly placed into an s-by-s square. We then repeatedly connect nodes until the network becomes connected. A link is inserted to connect two nodes u and v with probability $p(u, v) = \beta e^{-d(u,v)/\alpha L}$, where $d(u, v)$ is the Euclidean distance between u and v, $L = \sqrt{2}s$ is the largest possible distance between two nodes in the square, and α and β are parameters in the range $(0, 1]$. Larger value of β introduces higher edge density, and the value of α controls the relative ratio of the number of short edges to the number of long edges [19]. The cost of edge (u, v) is set to $d(u, v)$.

In our experiments the number of points N is set to 100 and the size of the domain s is set to 1000. The parameters α and β are set to 0.05 and 0.7 respectively. We generate 100 graphs using GT-ITM modeling tools [20] and the average number of edges in is 332. We assume that server 0 is the origin server, from which we construct an update distribution tree by connecting every server to server 0 by a shortest path. Finally, the default storage cost is set to 1000 and the default QoS requirement is set to 1000.

A. The effects of QoS

First, we compared the normalized replication cost of `Greedy Cover` with l-`Greedy-Insert` and l-`Greedy-Delete` under different QoS requirements. Figure 2 illustrates the normalized replication cost under different QoS requirements when the storage cost is set to 1000. From Figure 2, we observe that 1-`Greedy-Delete` always finds the best placement and the performance of `Greedy Cover` is only second to 1-`Greedy-Delete`. 1-`Greedy-Delete` aggressively tries to reduce the replication cost, so it finds better solution than other algorithms. Initially, `Greedy-Delete` assigns a replica to every server, which ensures a feasible configuration. Starting from this feasible state, in each iteration 1-`Greedy-Delete` searches for a replica set whose deletion causes the maximum reduction in the total cost, so it is more likely to get achieve low cost.

We expected the performance of 1-Greedy-Insert will be similar to 1-Greedy-Delete. However, the experiments show that the cost of 1-Greedy-Insert is nearly twice of 1-`Greedy-Delete` when QoS values are large, e.g., larger than 2000. When the QoS parameter increases to 3000, the network in average needs less than one replica to satisfy requests from all servers. However, if there are servers not satisfied by the origin server, 1-`Greedy-Insert` will always insert at least 2 replicas in the first step, so it has twice storage cost as the other algorithms.

When QoS is less than 2000, `Greedy MSC` has the highest normalized replication cost, even higher than 0-`Greedy Delete`. When QoS is 3000, the cost ratio increases to more than 2. From Table I, we observe that the number of replicas from `Greedy MSC` is almost the same to 1-`Greedy-Delete`, which is the best among all algorithms.

Consequently, the high cost ratio of `Greedy MSC` is not due to the number of replicas, but due to the update cost. In other words, the *position* of these replicas causes high normalized replication cost. This is because that `Greedy MSC` selects the candidate server only according to the number of unsatisfied servers, but fails to consider update cost.

Table I shows the average number of replicas 1-`Greedy-Insert`, 1-`Greedy-Delete`, `Greedy Cover` and `Greedy MSC` under different QoS requirements in 100 Waxman model graphs. From Table I we find similar performance trend as in Figure 2. When QoS is 2500 and 3000, the average number of replicas of 1-`Greedy-Insert` is twice as those of the other two algorithms. This is consistent with our previous observation of 1-`Greedy-Insert` in Figure 2.

From Table I we also find that 1-Greedy-Insert uses slightly more replicas than the other two algorithms. This is because l-`Greedy-Insert` makes decisions based on normalized benefits, not on the number of servers that will be satisfied. A placement having a larger normalized benefit value does not guarantee that more servers will be satisfied by this selection, and a selection with smaller value of normalized benefits may be able to satisfy all unsatisfied servers' requests in the same iteration. This causes 1-`Greedy-Insert` puts more replicas than the other algorithms do.

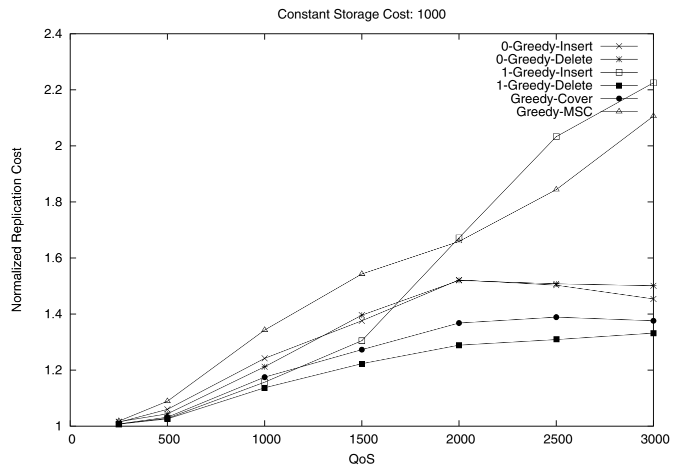

Fig. 2. Performance comparison under different QoS requirement, storage cost = 1000

QoS	`1-Insert`	`1-Delete`	GC	MSC
250	68.15	68.19	68.17	68.42
500	37.21	37.17	37.20	38.34
1000	12.68	12.28	12.63	13.25
1500	5.62	5.00	5.24	5.34
2000	3.11	2.15	2.33	2.26
2500	2.10	1.11	1.22	1.13
3000	1.10	0.53	0.56	0.53

TABLE I

AVERAGE NUMBER OF REPLICAS UNDER DIFFERENT QOS REQUIREMENT, THE DISTRIBUTION OF QOS IS CONSTANT

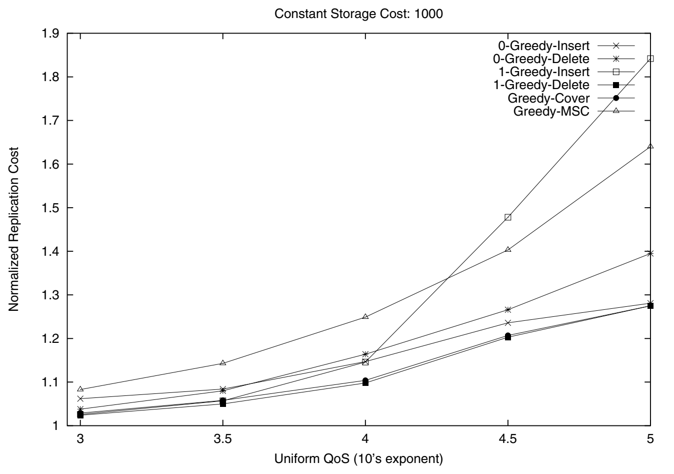

Fig. 3. Performance comparison when QoS values are taken from a uniform distribution.

QoS	`1-Insert`	`1-Delete`	GC	MSC
$[0, 2 \times 10^3]$	26.00	25.96	26.04	26.71
$[0, 2 \times 10^{3.5}]$	10.98	10.87	10.95	11.21
$[0, 2 \times 10^4]$	4.33	4.12	4.14	4.20
$[0, 2 \times 10^{4.5}]$	2.22	1.77	1.77	2.95
$[0, 2 \times 10^5]$	1.12	0.66	0.66	0.66

TABLE II

AVERAGE NUMBER OF REPLICAS UNDER DIFFERENT QOS REQUIREMENT, THE DISTRIBUTION OF QOS IS UNIFORM

Figure 3 and Table II show the normalized replication cost and the average number of replicas when QoS is from a uniform distribution. When the QoS uniform distribution has a mean value of 1000, the network needs more replicas than when QoS is set to a constant 1000. On the other hand, when QoS is from a uniform distribution the normalized replication cost of `Greedy Cover` is closer to 1-`Greedy-Delete` than when the QoS is set to a constant 1000. Finally, the relative order of the normalized replication cost from all algorithms under uniform distribution of QoS is similar to the case when QoS is a constant.

B. The effects of α

α	`1-Insert`	`1-Delete`	GC	MSC
0.05	12.68	12.28	12.63	13.25
0.10	4.15	3.58	3.71	3.83
0.15	2.25	1.27	1.43	1.26
0.20	1.54	0.75	0.81	0.75

TABLE III

AVERAGE NUMBER OF REPLICAS UNDER DIFFERENT α

Figure 4 illustrates the relationship between α and normalized replication cost and Table III shows the average number of replicas. When α increases, both the probability of using longer edges to connect nodes and the number of edges in

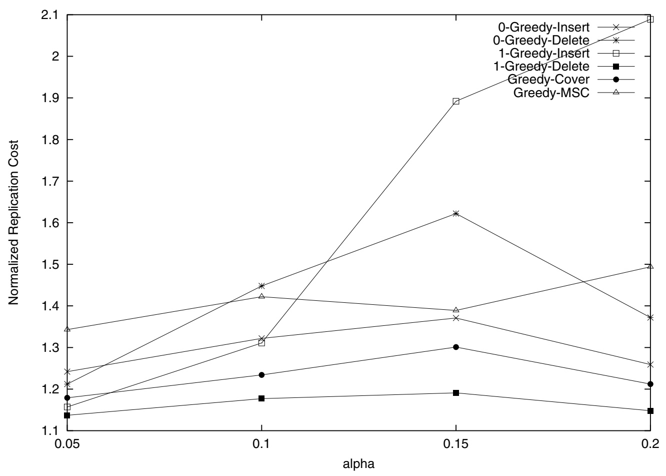

Fig. 4. Performance comparison for different α values.

the graph increase. As a result it is easier to satisfy the QoS requirement with only a few replicas.

The normalized replication cost of 1-`Greedy-Insert` increases abruptly when α is larger than 0.15. When α is 0.15, about half of the 100 graphs can be satisfied by one replica, and when α is 0.2, almost all graphs can be satisfied by one replica. When the average number of replicas required is less than one, the cost introduced by first step of 1-`Greedy-Insert` dominates the total cost. so that the normalized replication cost of 1-`Greedy-Insert` suddenly increases when α reaches 0.15.

The performance trend of varying α (Figure 4) is similar to that of varying QoS requirements (Figure 2). The effect of increasing α is similar to increasing QoS values of every server – both increase the chance that a server is satisfied by replicas. As a result we find similar phenomena in Figure 4 as in Figure 2. For example, when the servers in a network are more easily satisfied, the normalized replication cost of 1-`Greedy-Insert` becomes higher, 1-`Greedy-Delete` is always better than `Greedy Cover`, and a larger l value brings a better solution.

C. Execution time

Table IV compares average execution time from 100 graphs from 1-`Greedy-Insert`, 1-`Greedy-Delete` and `Greedy Cover` under different QoS values. We do not consider 0-`Greedy-Insert` and 0-`Greedy-Delete` because their normalized replication costs are much higher than `Greedy Cover`. The `Greedy Cover` uses only a few millisecond in average to calculate a placement, but 1-`Greedy Insert` or 1-`Greedy Delete` requires a significant amount of time to complete. We conclude that `Greedy Cover` is much more efficient than the two algorithms.

Although the solution quality of 1-`Greedy-Delete` is slightly better than `Greedy Cover`, the difference is very small. In constant and uniform QoS case, the difference is 9% and 1% respectively. `Greedy Cover` is a very efficient and effective algorithm for QoS replica placement since it runs much faster than 1-`Greedy-Insert` and 1-`Greedy-Delete` and produces solutions that are almost as good.

Despite that `Greedy MSC` runs faster than `Greedy Cover`, the difference is small. However, from previous experiments the placements produced by `Greedy MSC` have very high normalized replication cost, due to the fact that it considers only the number of replicas. Although `Greedy MSC` is efficient, it is not effective due to its poor solution quality.

QoS	1-Insert	1-Delete	GC	MSC
250	1.020s	0.421s	0.0059s	0.0027s
1000	0.143s	2.164s	0.0035s	0.0010s
2500	0.008s	4.807s	0.0001s	0.0003s

TABLE IV
AVERAGE EXECUTION TIME OF 100 GRAPHS UNDER DIFFERENT QOS

From Table IV we find that the execution time of 1-`Greedy-Insert` and 1-`Greedy-Delete` are strongly affected by QoS parameters. When QoS requirement is stringent, i.e., the QoS value is small, 1-`Greedy-Insert` requires more iterations to find the solution than 1-`Greedy-Delete`. From Table I we know that when QoS equals 250, the network needs about 70 replicas to satisfy all servers' requests. 1-`Greedy-Insert` starts from a state in which none of servers has a replica and every iteration of 1-`Greedy-Insert` increases the number of replicas by one. On the other hand, 1-`Greedy-Delete` places a replica in every server initially and decreases the number of replicas by one in every iteration. Therefore 1-`Greedy-Insert` spends more time than 1-`Greedy-Delete` when QoS requirement is stringent. On the contrary, when QoS requirement is less stringent, like 2500, the situation is reversed. In Table IV, we could see 1-`Greedy-Insert` just spends eight milliseconds and 1-`Greedy-Delete` needs more than 4 seconds when QoS equals 2500.

Table V illustrates the relationship between the number of servers and execution time of 1-`Greedy-Insert`, 1-`Greedy-Delete` and `Greedy Cover`. We observe that `Greedy Cover` uses less than 2 seconds even when the number of servers is 1100. 1-`Greedy-Insert` uses 88 seconds in the same setting. In addition, from previous experiments `Greedy Cover` produces better solution than 1-`Greedy-Insert` does, so `Greedy Cover` is a superior algorithm to 1-`Greedy-Insert`.

Although 1-`Greedy-Delete` has the best placement quality among three algorithms, it is much slower than 1-`Greedy-Insert` and `Greedy Cover`. When the number of servers is 500, 1-`Greedy-Delete` needs more than 1 hour to complete. Although the execution time of 1-`Greedy-Insert` increases as the number of servers increases, it does not grow so rapidly as 1-`Greedy-Delete` does. The reason is that when the number of servers increases in a fix-sized region, the requests are much easier to be satisfied, consequently 1-Greedy-Insert requires less number of iterations than 1-Greedy-Delete does. Although a larger number of servers makes both 1-Greedy-Insert and 1-Greedy-Delete spend more time per iteration, 1-Greedy-Insert requires less number of iterations than 1-Greedy-Delete does, so the execution time of 1-Greedy-Delete grows much faster than 1-Greedy-Insert.

Table V indicates that both 1-Greedy-Insert and 1-Greedy-Delete are easily influenced by the number of servers in the network. On the contrary, Greedy Cover is very stable and scalable. Greedy Cover can be deployed to large network systems with more than 1000 nodes, and deliver near optimal solution within a reasonable computation overhead.

# of servers	1-Insert	1-Delete	GC	MSC
100	0.586s	1.100s	0.006s	0.001s
300	10.599s	274.900s	0.184s	0.015s
500	25.594s	4198.000s	0.572s	0.027s
700	41.585s	N/A	1.029s	0.038s
900	67.950s	N/A	1.256s	0.051s
1100	88.358s	N/A	1.821s	0.101s

TABLE V

AVERAGE EXECUTION TIME OF 10 GRAPHS UNDER DIFFERENT NUMBER OF NODES, QoS = 500

VI. CONCLUSION

Data replication is an important technique to speed up data access in Data Grid. Grid computing infrastructure usually consists of various type of resources and the performance of these resources are quite diverse. So to provide quality assurance for different data access requirements is more and more important. We consider this problem as QoS-aware replica placement problem.

In this paper, we have presented an effective and efficient algorithm Greedy-Cover to solve the QoS-aware replica placement problem. The algorithm is very simple and easy to adapt to variant environments. Greedy Cover's performance is stable. It is not heavily influenced by QoS values. Experiment results indicate that Greedy-Cover efficiently finds near-optimal solutions in all parameter combinations. Moreover, Greedy-Cover is scalable, and is able to compute a near-optimal solution in two seconds when the number of servers equals 1100. In addition, when the quality of service guarantee becomes more stringent, the performance advantage of Greedy-Cover over other algorithms in the literature becomes more significant.

ACKNOWLEDGMENT

The authors would like to acknowledge the National Center for High-Performance Computing in providing resources under the national project , "Taiwan Knowledge Innovation National Grid".

REFERENCES

[1] 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*, 2000, pp. 77–90.

[2] 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*, 2002, pp. 376–381.

[3] 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*, 2002, pp. 378–383.

[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, 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*, 2002, pp. 46–57.

[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*, 2003, pp. 120–126.

[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] K. Ranganathana and I. Foster, "Identifying dynamic replication strategies for a high performance data grid," in *In Proceedings of the International Grid Computing Workshop*, 2001, pp. 75–86.

[9] 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*, 2001, pp. 305–314.

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

[11] C. Krick, H. Räcke, and M. Westermann, "Approximation algorithms for data management in networks," in *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures.* New York, NY, USA: ACM Press, 2001, pp. 237–246.

[12] M. Tu, P. Li, Q. Ma, I.-L. Yen, and F. B. Bastani, "On the optimal placement of secure data objects over internet," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers.* Washington, DC, USA: IEEE Computer Society, 2005, p. 14.

[13] O. Unger and I. Cidon, "Optimal content location in multicast based overlay networks with content updates," *World Wide Web*, vol. 7, no. 3, pp. 315–336, 2004.

[14] O. Wolfson and A. Milo, "The multicast policy and its relationship to replicated data placement," *ACM Trans. Database Syst.*, vol. 16, no. 1, pp. 181–205, 1991.

[15] X. Tang and J. Xu, "Qos-aware replica placement for content distribution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 10, pp. 921–932, 2005, member-Xueyan Tang and Member-Jianliang Xu.

[16] X. Jia, D. Li, X.-D. Hu, and D.-Z. Du, "Placement of read-write web proxies in the internet." in *ICDCS*, 2001, pp. 687–690.

[17] I. Cidon, S. Kutten, and R. Soffer, "Optimal allocation of electronic content," in *INFOCOM*, 2001, pp. 1773–1780. [Online]. Available: citeseer.ist.psu.edu/cidon01optimal.html

[18] W. J. Jeon, I. Gupta, and K. Nahrstedt, "Qos-aware object replication in overlay networks," 2005.

[19] B. M. Waxman, "Routing of multipoint connections," pp. 347–352, 1991.

[20] "GT Internetwork Topology Models (GT-ITM)," 2000, http://www-static.cc.gatech.edu/projects/gtitm/.