J Supercomput
DOI 10.1007/s11227-008-0221-1

QoS-aware, access-efficient, and storage-efficient replica placement in grid environments

Chieh-Wen Cheng · Jan-Jan Wu · Pangfeng Liu

Abstract In this paper, we study the quality-of-service (QoS)-aware replica placement problem in grid environments. Although there has been much work on the replica placement problem in parallel and distributed systems, most of them concern average system performance and have not addressed the important issue of quality of service requirement. In the very few existing work that takes QoS into consideration, a simplified replication model is assumed; therefore, their solution may not be applicable to real systems. In this paper, we propose a more realistic model for replica placement, which consider storage cost, update cost, and access cost of data replication, and also assumes that the capacity of each replica server is bounded.

The QoS-aware replica placement is NP-complete even in the simple model. We propose two heuristic algorithms, called *greedy remove* and *greedy add* to approximate the optimal solution. Our extensive experiment results demonstrate that both *greedy remove* and *greedy add* find a near-optimal solution effectively and efficiently. Our algorithms can also adapt to various parallel and distributed environments.

Keywords Replica placement · Data grids · Greedy heuristics

C.-W. Cheng · P. Liu
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan

P. Liu
Graduated Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan
e-mail: pangfeng@csie.ntu.edu.tw

J.-J. Wu (✉)
Institute of Information Science, Academia Sinica, Taipei 115, Taiwan
e-mail: wuj@iis.sinica.edu.tw

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. 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 [7, 12, 13]. 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 [2, 4–6, 12–15]. We believe that to obtain maximum benefits from replication, a strategic placement of the replicas is essential.

Although there has been much work on the replica placement problem [10, 11, 17, 18, 20], very few of them concerns quality of service. A large part of these works concern the average system performance; for example, minimizing the total accessing cost or minimizing 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 [16] considered the quality of service in addition to minimizing 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 graphs. They also propose a dynamic programming solution for tree topology [16]. Although the time complexity of l-`Greedy-Insert` and l-`Greedy-Delete` is a polynomial function of the nodes, it is not practical to apply these two algorithms in realistic environments due to their long execution time (several minutes when $l = 1$).

In this paper, we study the QoS-aware replica placement problem for general graphs with a more realistic model, which in addition to storage and update cost, also take access cost of replicas into account, and assumes that the workload capacity of a replica server is bounded. Our goal is to make sure that each request be serviced by a replica server within its quality requirement *and* without violating the capacity limits of the replica server. We provide two heuristic algorithms to decide the positions of the replicas to minimize the sum of update, storage, and access costs, and satisfy the quality requirements specified by the user and the capacity limit that each replica server can service. Our algorithm computes near-optimal solutions efficiently, so that it can be deployed in various realistic environments.

The rest of this paper is organized as follows. Section 2 describes previous work about replica placement. Section 3 describes our system model and defines some notations. Section 4 presents our algorithms and gives time complexity analysis. Section 5 presents our experiment results. Section 6 summarizes our research results and major contributions.

2 Related works

The 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 referred to as p-median problem in operations research, or database location problem on Internet, and file allocation problem in computer science. Wolfson and Milo [20] proved that replica placement problem is NP-Complete for general graphs when read and update costs are simultaneously considered. They also provide optimal solutions for special topologies, including complete graph, tree, and ring. Tu et al. [17] 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 $O(n^3k^2)$ dynamic programming algorithm for k replicas placed in n incapacitated servers, and an $O(n^3k^2\wedge^2_{\max})$ algorithm for capacitated servers, where $\wedge_{\max}$ denotes the maximum capacity among all servers. Unger and Cidon [18] provide a more efficient algorithm to find the optimal placement under similar model, with only $O(n^2)$ time, where n is the number of servers. However, the algorithm in [18] cannot deal with server capacity limits. There are other algorithms that provide optimal solutions under simpler models for tree topology [3, 9].

An early effort by Tang and Xu [16] 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 minimizes the update and storage cost at the same time. They show that QoS-aware replica placement problem is NP-Complete for general graphs, and provide two heuristic algorithms, called l-`Greedy-Insert` and l-`Greedy-Delete`, for general graph, and a dynamic programming solution for tree topology.

l-`Greedy-Insert` starts with an empty replication set R, and inserts replicas into R until all servers' QoS requirements are satisfied. l-`Greedy-Delete` works the opposite way as the l-`Greedy-Insert`. It begins 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.

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 $l > 0$ [16]. 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 [8]. The model did not consider update cost and assumed that each server has identical storage cost. The goal was to minimize the number of replicas in the system. They gave a proof of NP-Completeness for this problem, which is a variation of set covering. Let A be the all-to-all shortest path matrix and entry (i, j) of A denotes the shortest path distance between server i and server j. Let B be another matrix and the entries in the i-th row indicate quality of service requirement of server i. We then construct another matrix C according to $A - B$. If an entry of $A - B$ is less than or equal to 0, we set the corresponding entry of C to 1. Otherwise, we set the entry to 0. The nonzero elements of the j-th column of C represents the servers that are covered by server j. If we find a set of columns that cover every row in matrix C, we find a replica placement that satisfies all requests within quality of service requirement.

Won, Indranil, and Klara proposed a simpler and quicker algorithm to find a reasonable good solution for this problem. Every iteration in the algorithm, they select the column j (server j) that covers most rows that not yet covered so far. This Greedy Minimum Set Covering (Greedy MSC) is compared with our methods in our simulation testing.

Our model differs from the model in [16] in that we consider not only site construction cost, update overheads, and quality of service, but also the *access costs* of replica data and the *workload capacity constraint* of the replica servers.

3 System model

This section describes our system 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 replica server that can satisfy the data request, 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, $\mathcal{S}(u)$, that denotes the cost to put a replica on server u. The storage cost may vary depending on the nodes. 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.

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 requests. If the client's requests can be served by the server, i.e., the local server has the requested data and the requests will be processed locally. Otherwise, the requests will be directed to a 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

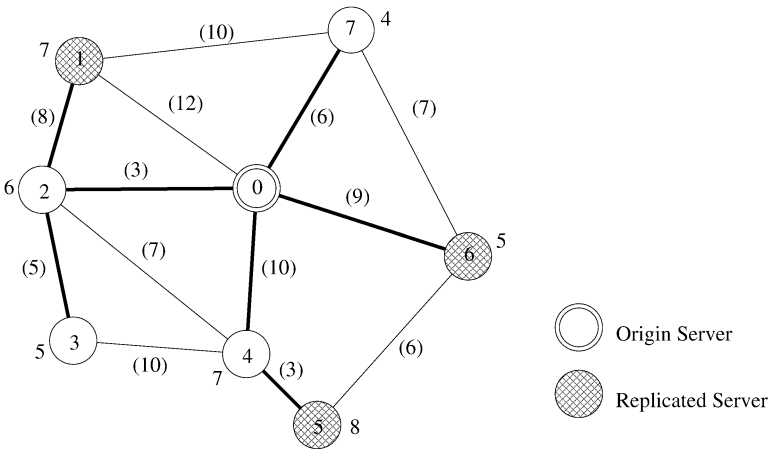


Fig. 1 An example of data replication in connected network

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 loss 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* has two parts: a *replica server set* $R \subseteq V - \{r\}$, and a *service set function* SS. For each server $u \in R \cup \{r\}$, we define a *service set* function $SS(u)$ to be the set of servers that u services. We assume that each server goes to only one server in $R \cup \{r\}$ for service; therefore, for two distinct servers $u, v \in R \cup \{r\}$, the service sets of u and v are disjoint, i.e., $SS(u) \cap SS(v) = \emptyset$.

3.1 Replication cost

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

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

Storage cost The storage cost of a replication strategy is the sum of all storage costs of the replica servers in the replication server set R. Recall that $\mathcal{S}(v)$ is the storage cost to replicate a data on server v:

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

Update cost In order to maintain data consistency, the origin 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 Fig. 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 servers 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 in 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 Fig. 1, if the update rate is 1 and the replication servers R is $\{1, 5, 6\}$, then the update cost is $11 + 13 + 9 = 33$:

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

Access cost Each server v has to communicate with a replica server u when it wishes to access the data from u, where $v \in SS(u)$. The access cost of a replication strategy is the sum of the communication cost that each server v accesses the data from its assigned replica server according to the service set function SS, as in the following equation:

$$A(R) = \sum_{u \in R \cup \{r\}} \sum_{v \in SS(u)} d(u, v). \tag{4}$$

3.2 Service quality requirement

Every server u has a *service quality requirement* $\Pi(u)$. The requirement mandates that all requests generated by u will be serviced by a server within $\Pi(u)$ communication cost. If a request from server u is serviced by a replica server within distance $\Pi(u)$ from u, we say server u is *satisfied*.

3.3 Workload capacity constraint

Each server u has a workload $\mathcal{W}(u)$ and workload capacity constraint $\mathcal{C}(u)$. The workload $\mathcal{W}(u)$ of a server is defined as the number of requests generated by server u. For each server u, when we put a replica on u, it has a workload capacity constraint, $\mathcal{C}(u)$, that denotes the amount of data requests that the replica server u can handle. The origin server also has its workload capacity constraint $\mathcal{C}(r)$. The workload and workload capacity constraint on different server may be different. If the total workload that a server $u \in R \cup \{r\}$ services is greater than its capacity constraint, i.e., $\sum_{v \in SS(u)} \mathcal{W}(v) > \mathcal{C}(u)$, then server u is *overloaded*.

3.4 QoS-aware replica placement with capacity constraint

A replication strategy is *feasible* if all servers are satisfied, and none of the server $u \in R \cup \{r\}$ is overloaded. The problem of QoS-aware replica placement with capacity constraint is to find a feasible replication server set R and determine the service set function $SS(u)$ for each server $u \in R \cup \{r\}$, such that the replication cost in (1) is minimized.

Figure 1 is an example of a feasible replication strategy. We assume that the QoS requirement $\mathcal{Q}$ is 8 for all servers, the workload capacity constraint $\mathcal{C}$ is 20 for each server, and the replication server set R is $\{1, 5, 6\}$. The workload function $\mathcal{W}$ of each server is in Table 1, and the service set of each server $u \in R \cup \{r\}$ is in Table 2. It is easy to verify that the replication strategy R is feasible. The storage cost is 20, the update cost is 33, the access cost is 21, so the replication cost is 74.

4 Heuristic algorithms

In this section, we first describe our two heuristic algorithms: *Greedy-Remove* and *Greedy-Add* for QoS-aware replica placement. Then we analyze the time complexity of these two algorithms.

Table 1 An example of the workload of each server in Fig. 1

Server	Workload
0	10
1	19
2	5
3	5
4	15
5	4
6	9
7	8

Table 2 An example of the service set of each server $u \in R \cup \{r\}$ in Fig. 1

Replica server	Service set
0	{0, 2, 3}
1	{1}
5	{4, 5}
6	{6, 7}

4.1 QoS satisfying set

We define a *QoS satisfying set* $SAT(u)$ of a server u to be the set of servers from which u is located within their QoS distance $\mathcal{Q}$. That means if u become an available replica server, it is able to satisfy those nodes in $SAT(u)$. Formally, we have

$$SAT(u) = \{v \mid d(u, v) \leq \mathcal{Q}(v)\}. \tag{5}$$

Each server has its own QoS satisfying set. If there is a replica on server u, each server $v \in SAT(u)$ may be satisfied by u, if u will not be overloaded by doing so. For a feasible replication strategy, the service set of each server $u \in R \cup \{r\}$ must be a subset of its QoS satisfying set $SAT(u)$; that is, $SS(u) \subseteq SAT(u)$ for all $u \in R \cup \{r\}$.

4.2 Greedy remove

The algorithm *Greedy-Remove* starts with having a replica on *every* server. This replication strategy is feasible since every server can serve itself locally so any QoS constraint is satisfied. Therefore, the service set of each server $u \in R \cup \{r\}$ has only itself. *Greedy-Remove* then repeatedly adjusts the service sets SS of a pair of replica servers and tries to remove replicas in order to reduce the replication cost (see (1)). While removing replicas, *Greedy-Remove* must simultaneously maintain the feasibility of the replication strategy. We consider two cases while adjusting the service sets for any two servers $u, v \in R \cup \{r\}$.

Case 1: The first case is to try to remove the replica from a nonoriginal server v by shifting all servers in the service set of v to the service set of another replica server u. This is only possible when u is within the QoS requirement of every server in $SS(v)$, and this shifting of workload will not cause u overloaded. That is, $SS(v) \subseteq SAT(u)$,

and the sum of all workload of the servers in $SS(u) \cup SS(v)$ does not exceed u's capacity constraint $\mathcal{C}(u)$. The replica will be removed from v if both conditions are satisfied

$$\sum_{w \in SS(u) \cup SS(v)} \mathcal{W}(w) \leq \mathcal{C}(u), \tag{6}$$

$$v \neq r, \quad SS(v) \subseteq SAT(u). \tag{7}$$

For any two servers $u, v \in R \cup \{r\}$, we define a cost reduction function c_1 to be the reduced cost by shifting the workload from v to u. For each server $v \in R$, we let $rm(v)$ be the reduced storage and update cost due to removing the replica from v. For ease of presentation, let AN_v denote the ancestors of server v in the update distribution tree. The rm function can be defined as follows:

$$rm(v) = \begin{cases} \mathcal{S}(v) + \sum_{w \in AN_v \cup \{v\}, T_w \cap R - \{v\} = \emptyset} d(w, p(w)) & \text{if } T_v \cap R - \{v\} \text{ is empty,} \\ \mathcal{S}(v) & \text{otherwise.} \end{cases} \tag{8}$$

The function $c_1(u, v)$ is formally defined as follows:

$$c_1(u, v) = \begin{cases} rm(v) + \sum_{w \in SS(v)} (d(w, v) - d(w, u)) & \text{Condition (6) and (7) are true,} \\ -\infty & \text{otherwise.} \end{cases} \tag{9}$$

Case 2: The second case is to try to shift a portion of the workload from a non-original server v to another server u. Similar to the first case, we can move a server w from $SS(v)$ to $SS(u)$ only when w is in $SS(v)$ and the communication cost between w and u is less than the communication cost between w and v, and w is in the QoS satisfying set $SAT(u)$ of u. Formally, we have $w \in SS(v)$, $d(w, u) < d(w, v)$, and $w \in SAT(u)$.

After identifying the possible servers that could be moved from $SS(v)$ to $SS(u)$, we must determine exactly which server must be moved. We sort all the serves in $SS(v)$ that could be moved according to their *distance reduction* (denoted as $d(w, v) - d(w, u)$). The intuition is that those servers with larger distance reduction should be moved first so that the total cost is reduced. Now we start to move servers from $SS(v)$ to $SS(u)$ one at a time according to their distance reduction (in nondecreasing order). The iterative process stops when either there is no more server in $SS(v)$ that could be moved to u, or server u may become overloaded. Let $CS(u, v) \subset SS(v)$ denote the set of servers that are moved from $SS(v)$ to $SS(u)$.

For two servers $u, v \in R \cup \{r\}$, we define a c_2 function to denote the reduction cost of the second case. The $c_2(u, v)$ function is formally defined as the following:

$$c_2(u, v) = \begin{cases} \sum_{w \in CS(u,v)} (d(w, v) - d(w, u)) & \text{if } CS(u, v) \text{ is nonempty,} \\ -\infty & \text{otherwise.} \end{cases} \tag{10}$$

We define a *cost reduction* function $\mathcal{R}(u, v)$ for any two servers $u, v \in R \cup \{r\}$ to be the maximum reduction cost of the above two cases:

$$\mathcal{R}(u, v) = \max\{c_1(u, v), c_2(u, v)\}. \tag{11}$$

We now describe our *Greedy-Remove* method in details. The *Greedy-Remove* algorithm is iterative. Initially, we put a replica on every server in $V - \{r\}$, so the replication server set R is $V - \{r\}$. In each iteration, *Greedy-Remove* examines each pair of servers u, v in $R \cup \{r\}$ and computes the cost reduction function $\mathcal{R}(u, v)$. Then *Greedy-Remove* selects the maximum $\mathcal{R}(u, v)$ and adjusts the service sets of u and v, accordingly. *Greedy-Remove* repeats this process until it is impossible to reduce the total replication cost.

4.2.1 Time complexity analysis

We now analyze the time complexity of *Greedy-Remove*. The first part of the costs is a preprocessing to find a shortest path between any two servers. That is, we must build an all-pair shortest path to check if one server is within the QoS requirement of another. This takes $O(|V|^3)$ time to calculate.

Given the servers u and v, it takes $O(|V|)$ to find out the cost reduction of moving all servers from v to u for the first case. For the second case, it takes $O(|V| \log |V|)$ time to sort the servers, then examine them one by one and move them from u to v if its distance to v is longer than the distance to u.

We first compute the cost reduction function $\mathcal{R}$ of each pair of replica servers. This takes $O(|V|^3 \log |V|)$ time.

In each iteration, we choose the largest cost reduction $\mathcal{R}$. If the corresponding case is the first case, we need to consider two situations. The first is that there is no replica in the subtree rooted at v after removing the replica from v. We have to recompute the cost reduction function of v's parent replica server with the other replica servers. Because its reduced update cost in *rm* function has been changed. The other situation is that there is only one replica in the subtree T_v after taking off the replica from v. We have to recompute the cost reduction function of this replica server with the other replica servers. This is because the reduced update cost of this replica server is increased. Hence, if the corresponding case is the first case, it needs $O(|V|^2)$ time. Otherwise, if the corresponding case is the second case, we only need to recompute the cost reduction function of the two replica servers that we selected with the other replica servers. This takes $O(|V|^2 \log |V|)$. Thus, each iteration needs $O(|V|^2 \log |V|)$ time.

In each iteration, we remove a replica from a server or move at least one server from one service set to another. The number of replica removal is at most $|V|$ since a replica can be removed at most once. In addition, each server without placing the replica can be moved at most $|V|$ times because it can only be moved from a service set to another once, and there are $|V|$ servers in the network system. Therefore, there are $O(|V|^2)$ iterations. As a result, the time complexity of *Greedy-Remove* is $O(|V|^3 + |V|^3 \log |V| + |V|^2 \cdot |V|^2 \log |V|) = O(|V|^4 \log |V|)$.

4.3 Greedy Add

The *Greedy-Add* algorithm works the opposite way as *Greedy-Remove* does. It begins with an empty replication server set R, and add replicas to R one at a time.

Greedy-Add first finds the QoS satisfying set $SAT(u)$ for every server u in V. For ease of notation, we also define $rep(u)$ of a server u to be the server in $R \cup \{r\}$ that

serves u. For ease of processing, we also assume that the servers in a QoS satisfying set $SAT(u)$ are sorted according to their communication cost to u, i.e., $d(u, v)$, where v is any server in $SAT(u)$.

The *Greedy-Add* algorithm has two stages. In the first stage, *Greedy-Add* repeatedly adds the replicas into the replication strategy R until the replication strategy R is feasible. In the second stage, *Greedy-Add* adds replicas into the replication server set R until it is impossible to reduce the replication cost.

At the beginning of the first stage, *Greedy-Add* selects servers from $SAT(r)$ to form $SS(r)$. Recall that $SAT(r)$ is the set of servers r can satisfy in terms of QoS. *Greedy-Add* selects servers in $SAT(r)$ in a nondecreasing order of their distance to r, i.e., the server v with the smallest $d(r, v)$ will be considered first. The selection repeats until there is no capacity available on r or no server available from $SAT(r)$.

In each following step, *Greedy-Add* examines every server $u \in V - R \cup \{r\}$ and computes the replication cost of placing a replica on u. When *Greedy-Add* places a replica on a server $u \in V - R \cup \{r\}$, u must also serve itself, so *Greedy-Add* moves u from the SS set of whoever serving u to $SS(u)$. Then *Greedy-Add* can start selecting servers from its $SAT(u)$ and move them into $SS(u)$. The selection starts from the first server v in $SAT(u)$ that has not been served by any server in $R \cup \{r\}$ until there is no workload capacity available on u, or every server in $SAT(u)$ has already been served by $R \cup \{r\}$.

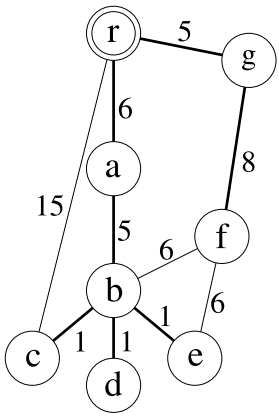
If we put a replica on a server $u \in V - R \cup \{r\}$ and determine $SS(u)$ by the above selection process, the replication cost may increase or decrease. The decreasing of the replication cost is due to the decreasing of access cost, despite the increased storage cost. If putting the replica on some servers $u \in V - R \cup \{r\}$ can decrease the replication cost, *Greedy-Add* ignores the servers which increase the replication cost and puts the replica on the server that can reduce the most replication cost.

If *Greedy-Add* cannot find a location to place a replica to reduce the total replication costs, it will place a replica on the server with the *highest normalized benefit*. Here, as in [16], we define normalized benefit as the increased number of served servers divided by the increased replication cost due to the extra replicas. *Greedy-Add* repeats this process until all servers are serviced by a server $u \in R \cup \{r\}$.

For example, let us assume that the network in Fig. 2 has QoS requirements 5, storage cost is 2, workload is 3, and capacity constraints is 20 for every server. *Greedy-Add* first inserts g into $SS(r)$, the replication cost becomes 5 due to the access cost of g and the service set of r is $\{r, g\}$. Then *Greedy-Add* examines servers in $V - \{r\}$. Since none of the servers in $u \in V - \{r\}$ can decrease the replication cost if we put a replica on it, *Greedy-Add* selects b to put a replica because b has the highest normalized benefit. The service set of b is $\{a, b, c, d, e\}$ and the replication cost is now 26. *Greedy-Add* continues to examine servers in $V - \{r, b\}$, and finds that when it places a replica on a, the replication cost will actually decrease. Therefore, *Greedy-Add* chooses a to put the next replica and the service set of b becomes $\{b, c, d, e\}$, the service set of a is $\{a\}$, and the total replication cost becomes 23. In the next iteration, *Greedy-Add* places a replica on f, which has the highest normalized benefit. Finally, the replication server set R is $\{a, b, f\}$. This strategy is feasible and the replication cost is 38. Refer to Fig. 3 for an illustration.

In the second stage, *Greedy-Add* repeatedly adds replica in order to decrease the access cost. This stage works also in iterations. In each iteration, *Greedy-Add* exam-

Fig. 2 An example of network



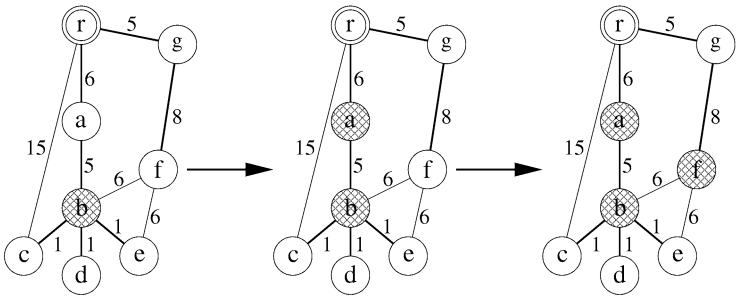


Fig. 3 An example of the first stage of *Greedy-Add*

ines all servers $u \in V - R \cup \{r\}$, and try to place a replica on a server u to determines whether this will reduce the replication cost.

When we put the replica on a server $u \in V - R \cup \{r\}$, u has to serve itself and u must be put into $SS(u)$. Then we start selecting servers from $SAT(u)$ and move them to $SS(u)$. We select servers $v \in SAT(u)$ whose communication cost ($d(v, u)$) is less than the communication to its original server ($d(v, rep(v))$). The selection starts from the server v with the largest $d(v, rep(v)) - d(v, u)$, until there is no available capacity on u, or no server left in $SAT(u)$. We then recompute the replication cost. After trying all servers in $V - R \cup \{r\}$, *Greedy-Add* puts the next replica on a server that reduces the replication cost most. *Greedy-Add* repeats the process until it is impossible to reduce the replication cost.

For example, let us assume that the network on the left of Fig. 4 represents the result from the first stage of *Greedy-Add* and the replication server set $R = \{a, b, f\}$. In the second stage of *Greedy-Add*, it examines all servers in $V - R \cup \{r\}$, and decides to place a replica on g. Although putting the replica on g increases the storage cost, the access cost of g becomes 0 because of the new replica on g. After putting the replica on g, the replication cost becomes 35 and the replication R becomes $\{a, b, f, g\}$. The service set of r becomes $\{r\}$, and the service set of g is $\{g\}$.

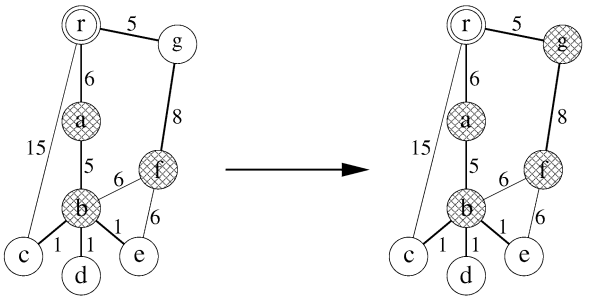


Fig. 4 An example of the second stage of *Greedy-Add*

4.3.1 Time complexity analysis

We analyze the time complexity of *Greedy-Add*. Similar to the analysis of *Greedy-Remove*, the first part of the costs is a preprocessing to find a shortest path between any two servers, which takes $O(|V|^3)$ time to calculate.

In the first stage, *Greedy-Add* inserts the replicas into the network iteratively until all servers are served. In each iteration, *Greedy-Add* examines $O(|V|)$ servers. If we put the replica on a server, it takes $O(|V|)$ time to determine the service set and to compute the normalized benefit. Thus, each iteration takes $O(|V|^2)$ time. In each iteration, *Greedy-Add* puts the replica on a server and there are at most $|V|$ servers in the network. Therefore, in the first stage, *Greedy-Add* requires $O(|V|^3)$ time to determine a feasible replication strategy.

In the second stages, *Greedy-Add* inserts the replicas into the network to reduce the replication cost until further reduction in replication cost is not possible. In each iteration, *Greedy-Add* examines $O(|V|)$ servers. It takes $O(|V|\log|V|)$ time to determine the service set and $O(|V|)$ time to recompute the replication cost. Since there are $|V|$ servers in the network, the second stage of *Greedy-Add* takes $O(|V|^3\log|V|)$ time to finish. As a result, the overall complexity of *Greedy-Add* is $O(|V|^3\log|V|)$.

4.4 Random

We also implement a randomized algorithm as a comparison for the *Greedy-Remove* and *Greedy-Add*. We randomly insert the replica into the network until all servers are served. The servers in a QoS satisfying set $SAT(u)$ are sorted according to their communication cost to u. When we put the replica on a server u, u has to serve itself first. Then we select servers from $SAT(u)$ to form $SS(u)$. The selection starts from the first server v in $SAT(u)$ that has not been served by any server in $R \cup \{r\}$ until there is no workload capacity available on u, or every server in $SAT(u)$ have already been served by $R \cup \{r\}$.

5 Performance evaluation

This section describes our experimental results. Tang and Xu [16] formulate the replica placement as an integer programming problem, then relax the requirements

for an integer solution and consequently transform the integer program into a linear program. The solution from the linear then serves as a comparison basis for solution quality.

We also construct an integer program for the problem of QoS-aware replica placement with capacity constraint, and then we transform it into a linear program by relaxing the integer solution requirements. 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. We compare the solutions by our heuristic algorithms with this super optimal solution. The ratio of cost computed by the heuristic algorithm to the cost computed by the super optimal solution is referred to as *normalized replication cost*.

We now describe the process of how to obtain this super optimal solution. Let $V = \{v_0, v_1, v_2, \ldots, v_{n-1}\}$ be the set of servers and v_0 be the origin server, i.e., $v_0 = r$. The replica placement problem can be expressed as the following integer program.

The 0-1 variable x_i represents whether a replica is placed at server v_i, and the 0-1 variable y_i represents whether y_i will receive data update requests from its parent $p(v_i)$ in the update distribution tree T [16]. Let the 0-1 variable z_{ij}, $0 \leq i < n$ and $0 < j < n$, denote whether v_j will be assigned to v_i. Our goal is to minimize the following objective function:

$$\sum_{n>i>0} \big(\mathcal{S}(v_i) \times x_i + d\big(v_i, p(v_i)\big) \times y_i\big) + \sum_{n>i\geq 0} \sum_{n>j>0} d(v_i, v_j) \times z_{ij}. \tag{12}$$

The minimization of (12) is subject to the following constraints:

$$x_i, y_i \in \{0, 1\}, \quad 0 < \forall i < n, \tag{13a}$$

$$z_{ij} \in \{0, 1\}, \quad 0 \leq \forall i < n \wedge 0 < \forall j < n, \tag{13b}$$

$$y_i \geq x_i, \quad 0 < \forall i < n, \tag{13c}$$

$$y_i \geq y_j, \quad 0 < \forall i, j < n \wedge p(v_j) = v_i, \tag{13d}$$

$$x_i = z_{ii}, \quad 0 < \forall i < n, \tag{13e}$$

$$x_i \geq z_{ij}, \quad 0 < \forall i, j < n \wedge i \neq j, \tag{13f}$$

$$z_{ij} = 0, \quad 0 \leq \forall i < n \wedge 0 < \forall j < n \wedge d(v_i, v_j) > \mathcal{Q}(v_j), \tag{13g}$$

$$\sum_{0\leq i<n} z_{ij} = 1, \quad 0 < \forall j < n, \tag{13h}$$

$$\sum_{0<j<n} z_{ij} \times \mathcal{W}(v_j) \leq x_i \times \mathcal{C}(v_i), \quad 0 < \forall i < n, \tag{13i}$$

$$\sum_{0<j<n} z_{0j} \times \mathcal{W}(v_j) \leq \mathcal{C}(v_0) - \mathcal{W}(v_0). \tag{13j}$$

Constraint (13e) ensures that v_i must be serviced locally if there is a replica placed on v_i, where $0 < i < n$, and constraint (13f) ensures that there must be a replica

placed on v_i if v_j is assigned to v_i. Constraint (13g) prohibits v_j from going to v_i for service if v_i is out of the QoS requirement of v_j, where $0 \leq i < n$ and $0 < j < n$, i.e., $d(v_i, v_j) > \mathcal{Q}(v_j)$, and constraint (13h) ensures that each server v_j is assigned to at least one replica server. Constraints (13i) and (13j) ensure that each replica server and origin server will not exceed its capacity restriction.

If we replace constraint (13a) and (13b) with (14a) and (14b), respectively, we have a linear program instead of an integer program:

$$0 \leq x_i, y_i \leq 1, \quad 0 < \forall i < n, \tag{14a}$$

$$0 \leq z_{ij} \leq 1, \quad 0 \leq \forall i < n \wedge 0 < \forall j < n. \tag{14b}$$

Because the optimal solution of the linear program is as good as the optimal solution for the integer program, it can be used as a "super" optimal solution for the replica placement problem. Note that an optimal solution from the linear program may not even be a 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 to the true optimum. We use normalized replication cost as performance metric. Normalized replication cost is the ratio of cost produced to the linear program solution.

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 the 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 Waxman model parameters. Both α and β are 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 100 and the size of the domain s is 1,000. Waxman model parameters α and β are set to 0.05 and 0.7, respectively. We generate 100 Waxman model graphs using GT-ITM modeling tools [1]. The average number of edges in these 100 graphs 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 1,000, the default QoS requirement is 1,000, the default workload is taken from a uniform distribution over the range $[0, 100]$, and the default server capacity limit is 500.

5.1 The effects of workload

We first compare the normalized replication cost of *Greedy-Remove* with *Greedy-Add* under different range of workload uniform distributions. Figure 5 illustrates the normalized replication cost under a different range of workload uniform distribution when other parameters are set to default values. In Fig. 5, we can see that *Greedy-Remove* always outperforms *Greedy-Add*.

Table 3 shows the average number of replicas that *Greedy-Remove* and *Greedy-Add* put under different range of workload uniform distribution. From Table 3, we observe that when the upper bound of uniform distribution increases, the average number of the replicas that *Greedy-Remove* and *Greedy-Add* put also increases.

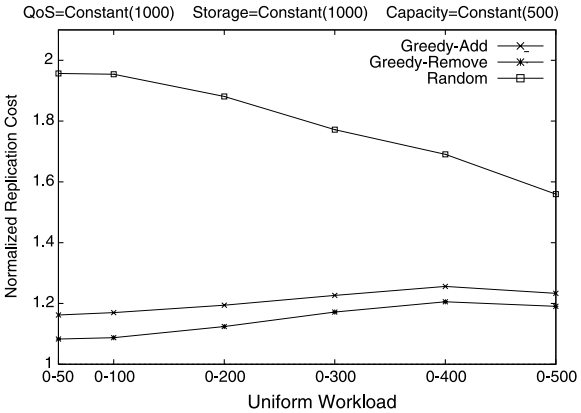


Fig. 5 Performance comparison when workload values are taken form a uniform distribution, QoS requirement = 1,000, storage cost = 1,000, capacity constrain = 500

Table 3 Average number of replicas under different workload values, the distribution of workload is uniform

Load	Greedy-Add	Greedy-Remove
[0, 50]	23.76	19.98
[0, 100]	24.09	21.01
[0, 200]	30.20	28.63
[0, 300]	40.50	39.20
[0, 400]	53.22	51.93
[0, 500]	62.77	61.76

5.2 The effects of capacity constraint

We now consider the normalized replication cost of *Greedy-Remove* and *Greedy-Add* under different capacity constraints. Figure 6 illustrates the normalized replication cost under different constant values of capacity constraints when other parameters are set to default values. In Fig. 6, we can see that the performance of *Greedy-Remove* is always better than *Greedy-Add*.

Table 4 shows the average number of replicas that *Greedy-Remove* and *Greedy-Add* put under different capacity constraints. From Table 4 we see that when the capacity constraint increases, the average number of replicas that *Greedy-Remove* and *Greedy-Add* put decreases.

From Fig. 6 and Table 4 we observe that the normalized replication cost and the average number of replicas become steady when the capacity constraint is greater than 600. This is because QoS requirement and access cost of each server restrict the average number of replicas in the system.

Figure 7 and Table 5 illustrates the normalized replication cost of *Greedy-Remove* and *Greedy-Add* and the average number of replicas that these two methods put under different ranges of capacity constraint from uniform distribution when other parameters are set to default values. From Fig. 7, the performance of *Greedy-Remove* is always better than *Greedy-Add*. From Table 5, the average number of replicas that *Greedy-Remove* and *Greedy-Add* place decreases when the capacity constraint in-

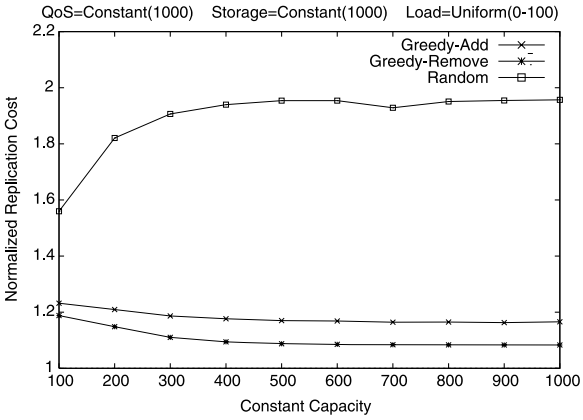


Fig. 6 Performance comparison under different capacity constraints, QoS requirement = 1,000, storage cost = 1,000, workload = [0, 100]

Table 4 Average number of replicas under different capacity constraints, the capacity constraint is constant

Capacity constraint	Greedy-Add	Greedy-Remove
100	62.38	61.13
200	34.70	33.63
300	27.63	25.60
400	24.99	22.38
500	24.09	21.01
600	24.04	20.37
700	23.64	20.16
800	23.70	20.02
900	23.43	20.00
1000	23.63	19.98

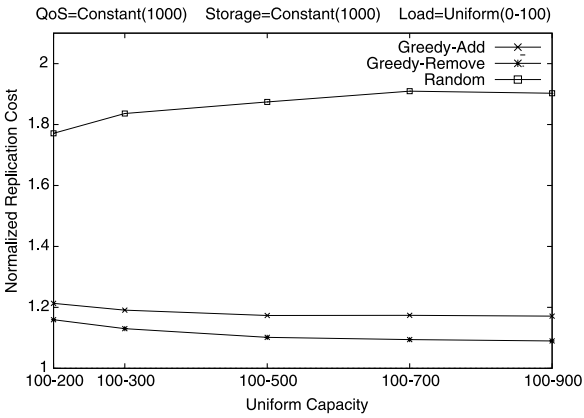


Fig. 7 Performance comparison when capacity constraints are taken form a uniform distribution, QoS requirement = 1,000, storage cost = 1,000, workload = [0, 100]

creases. The normalized replication cost and the average number of replicas also become stable when the upper bound of capacity constraint is greater than 700.

Table 5 Average number of replicas under different capacity constraints, the distribution of capacity constraint is uniform

Capacity constraint	Greedy-Add	Greedy-Remove
[100, 200]	41.38	40.37
[100, 300]	32.63	31.45
[100, 400]	27.41	24.82
[100, 700]	25.84	22.62
[100, 900]	25.34	21.60

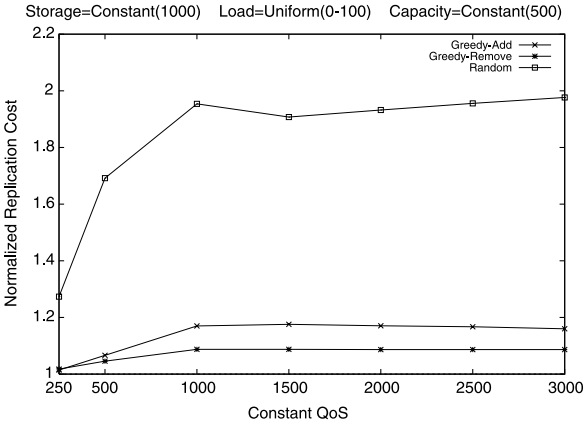


Fig. 8 Performance comparison under different QoS requirements, storage cost = 1,000, workload = [0, 100], capacity constraint = 500

Table 6 Average number of replicas under different QoS requirements, the distribution of QoS is constant

QoS	Greedy-Add	Greedy-Remove
250	68.65	69.64
500	40.33	40.25
1000	24.09	21.01
1500	22.44	18.22
2000	21.67	18.04
2500	21.17	18.03
3000	20.94	18.03

5.3 The effects of QoS

Figure 8 gives the normalized replication cost under different constant values of QoS requirement when other parameters are set to default values. Figure 8 show that the performance of *Greedy-Remove* is worse than *Greedy-Add* only when QoS requirement is set to 250. When the QoS requirement is larger than 500, *Greedy-Remove* outperforms *Greedy-Add*.

Table 6 shows the average number of replicas that these two algorithms put under different QoS requirements. From Table 6, we find that as the QoS requirement increases the average number of replicas that these two algorithms put will decrease. We also can see that the average number of replicas change slightly when QoS requirement is greater than 1,500. This is because the capacity constraint and access

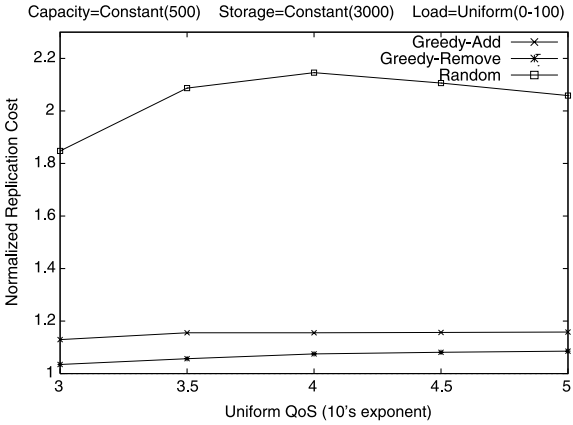


Fig. 9 Performance comparison when QoS requirements are taken form a uniform distribution, storage cost = 1,000, workload = [0, 100], capacity constraint = 500

Table 7 Average number of replicas under different QoS requirements, the distribution of QoS is uniform

QoS	Greedy-Add	Greedy-Remove
$[0, 2 \times 10^3]$	35.82	31.04
$[0, 2 \times 10^{3.5}]$	26.65	22.22
$[0, 2 \times 10^4]$	22.57	19.30
$[0, 2 \times 10^{4.5}]$	21.40	18.44
$[0, 2 \times 10^5]$	21.13	18.08

cost limit the average number of replicas. We also observe that the average number of replicas that *Greedy-Add* puts is less than *Greedy-Remove* when QoS requirement is set to 250. This is because the first stage of *Greedy-Add* tries to find a feasible with less replicas. In most cases, the second stage never runs due to the small QoS requirement.

Figure 9 and Table 7 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 1,000, the network needs more replicas than when QoS is a constant 1,000, and the normalized replication cost is also less than when QoS is a constant 1,000.

5.4 The effects of storage cost

Figure 10 and Table 8 show the normalized replication cost and the average number of replicas under different constant values of storage cost for each server. Table 8 shows that when storage cost increases, the average number of replicas decrease. This is because putting a replica is more expensive than the access cost. Because of the capacity constraint, the average number of replicas change slightly when storage cost is larger than 3,000.

5.5 Execution time

Table 9 and Table 10 illustrates the average execution time of 100 graphs under different QoS requirements and different workload capacity constraints. Table 9 shows

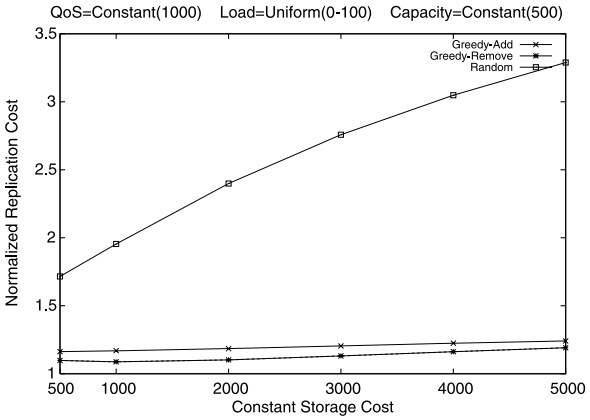


Fig. 10 Performance comparison under different storage cost, QoS requirement = 1,000, workload = [0, 100], capacity constraint = 500

Table 8 Average number of replicas under different storage cost, the distribution of storage cost is constant

Storage cost	Greedy-Add	Greedy-Remove
500	33.62	27.32
1000	24.09	21.01
2000	19.34	18.56
3000	18.05	18.11
4000	17.74	18.04
5000	17.67	18.04
6000	17.51	18.04

Table 9 Average execution time of 100 graphs under different QoS requirement

QoS	Greedy-Add	Greedy-Remove
500	0.004 s	0.007 s
1000	0.004 s	0.009 s
2000	0.006 s	0.011 s
3000	0.007 s	0.012 s

Table 10 Average execution time of 100 graphs under different workload capacity limit, the workload is uniform distribution over the range [0, 100]

Capacity	Greedy-Add	Greedy-Remove
100	0.005 s	0.006 s
300	0.005 s	0.008 s
500	0.005 s	0.009 s
700	0.005 s	0.009 s

that when the QoS requirement of each server increases, the average execution of *Greedy-Add* and *Greedy-Remove* increase. From Table 9 and Table 10, the average execution time of *Greedy-Add* is always less than *Greedy-Remove*.

6 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. This replica placement problem become more complex when the storage for replica on servers is limited. We believe that quality of service and workload capacity should be considered simultaneously for quality assurance, and the access cost must also be taken into account for system performance.

In this paper, we consider QoS requirement, workload capacity restriction and access cost on replica placement problems. We believe that all the key issuers, including storage cost, quality of service, server capacity constraint, access costs, and update costs should be considered. Our proposed algorithms consider all these key issues, and is very simple and easy to adapt to various environments. Experiment results indicate that *Greedy-Remove* and *Greedy-Add* can find near-optimal solutions in all parameter combinations.

References

1. GT Internetwork Topology Models (GT-ITM) (2000) http://www-static.cc.gatech.edu/projects/gtitm/
2. Chervenak A, Schuler R, Kesselman C, Koranda S, Moe B (2005) Wide area data replication for scientific collaborations. In: Proceedings of the 6th international workshop on grid computing, November 2005
3. Cidon I, Kutten S, Soffer R (2001) Optimal allocation of electronic content. In: INFOCOM, pp 1773–1780
4. David WB (2003) Evaluation of an economy-based file replication strategy for a data grid. In: International workshop on agent based cluster and grid computing, pp 120–126
5. David WB, Cameron DG, Capozza L, Millar AP, Stocklinger K, Zini F (2002) Simulation of dynamic grid rdeplication strategies in optorsim. In: Proceedings of 3rd international IEEE workshop on grid computing, pp 46–57
6. Deris MM, Abawajy JH, Suzuri HM (2004) An efficient replicated data access approach for large-scale distributed systems. In: IEEE international symposium on cluster computing and the grid, April 2004
7. Hoschek W, Janez FJ, Samar A, Stockinger H, Stockinger K (2000) Data management in an international data grid project. In: Proceedings of GRID workshop, pp 77–90
8. Jeon WJ, Gupta I, Nahrstedt K (2005) Qos-aware object replication in overlay networks
9. Jia X, Li D, Hu X-D, Du D-Z (2001) Placement of read-write web proxies in the Internet. In: ICDCS, pp 687–690
10. Kalpakis K, Dasgupta K, Wolfson O (2001) Optimal placement of replicas in trees with read, write, and storage costs. IEEE Trans Parallel Distrib Syst 12(6):628–637
11. Krick C, Räcke H, Westermann M (2001) Approximation algorithms for data management in networks. In: SPAA '01: Proceedings of the thirteenth annual ACM symposium on parallel algorithms and architectures. ACM Press, New York, pp 237–246
12. Lamehamedi H, Szymanski B, Shentu Z, Deelman E (2002) Data replication strategies in grid environments. In: Proceedings of 5th international conference on algorithms and architecture for parallel processing, pp 378–383
13. Ranganathan K, Iamnitchi A, Foste IT (2002) Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In: 2nd IEEE/ACM international symposium on cluster computing and the grid, pp 376–381
14. Ranganathana K, Foster I (2001) Identifying dynamic replication strategies for a high performance data grid. In: Proceedings of the international grid computing workshop, pp 75–86

15. Stockinger H, Samar A, Allcock B, Foster I, Holtman K, Tierney B (2001) File and object replication in data grids. In: 10th IEEE symposium on high performance and distributed computing, pp 305–314
16. Tang X, Xu J (2005) Qos-aware replica placement for content distribution. IEEE Trans Parallel Distrib Syst 16(10):921–932
17. Tu M, Li P, Ma Q, Yen I-L, Bastani FB (2005) 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). IEEE Computer Society, Washington, p 14
18. Unger O, Cidon I (2004) Optimal content location in multicast based overlay networks with content updates. World Wide Web 7(3):315–336
19. Waxman BM (1991) Routing of multipoint connections, pp 347–352
20. Wolfson O, Milo A (1991) The multicast policy and its relationship to replicated data placement. ACM Trans Database Syst 16(1):181–205