A Distributed Algorithm for Web Content Replication

Sharrukh Zaman
Department of Computer Science
Wayne State University
Detroit, Michigan 48202-3929
Email: sharrukh@wayne.edu

Daniel Grosu
Department of Computer Science
Wayne State University
Detroit, Michigan 48202-3929
Email: dgrosu@cs.wayne.edu

***Abstract*—Web caching and replication techniques increase accessibility of Web contents and reduce Internet bandwidth requirements. In this paper, we are considering the replica placement problem in a distributed replication group. The replication group consists of servers dedicating certain amount of memory for replicating objects. The replica placement problem is to place the replica at the servers within the replication group such that the access time over all objects and servers is minimized. We design a distributed 2-approximation algorithm that solves this optimization problem. We show that the communication and computational complexity of the algorithm is polynomial in the number of servers and objects. We perform simulation experiments to investigate the performance of our algorithm.**

I. Introduction

In Internet content distribution, a distributed replication group stores object replicas to better serve users by guaranteeing lower access costs. The servers in the replication group provide storage to replicate popular Internet content for their users with the goal of minimizing the overall access cost. A request for a particular object by a user is served with minimum access cost if it is stored at the user's local server. If that object is not replicated at the local server but it is stored in a remote server instead, the access cost is higher. The users incur maximum access cost when no peer replicators store a replica of the requested object, in which case, the local server has to fetch it from the origin server.

There are several approaches to solve the replica placement problem and hence different solutions exist in the literature. In the context of Internet, a distributed solution is more acceptable than a centralized one. We desire an optimal solution obtainable by a polynomial time algorithm which has reasonable running time and communication overhead. When optimal solutions cannot be achieved in polynomial time we consider approximation algorithms. The replica placement problem we are considering here is a generalized version of the multiple knapsack problem in that it allows multiple copies of the objects to be placed in the bins, and object profits that vary with the bin and the items already inserted in the bins. Since the multiple knapsack problem is NP-hard [1] it follows that the replica placement problem is NP-hard. In this paper, we design an approximation algorithm that guarantees that the total access time is within a factor of two from the optimal. The algorithm runs in polynomial time and has a communication cost that is polynomial in the number of servers, objects and total server capacities.

Related work. The web replication problem we are considering has some similarities with several other optimization problems such as, the generalized assignment problem [2], the multiple knapsack problem [1], the facility location problem [3], and the transportation problem [4]. Moscibroda and Wattenhofer [3] developed a distributed approximation algorithm for the facility location problem. Leff and Wolf [5] provided an optimal solution for their model of Remote Caching Architecture, reducing it to a capacitated transportation problem. Complexity results for problems ranging from the knapsack to the generalized assignment problem (GAP) are given in [1]. Baev *et al.* [6] proposed approximation algorithms for data placement problems that take into account the storage costs. Recently, several researchers considered the replica placement problem in systems with selfish replicators [7], [8].

Our contribution. We design a distributed 2-approximation algorithm for the replica placement problem in distributed systems. We characterize its computational and communication complexity. We show that it provides a 2-approximation to the optimal replica placement problem. We investigate by simulation the performance of the algorithm.

II. Replica Placement Problem

In this section, we define the replica placement problem we are solving. We use the system model described in [8], with different notation. We consider that the replication group is composed of m servers $s_1, \ldots, s_m$ with capacities $c_1, \ldots, c_m$. There are n unit-sized objects $o_1, \ldots, o_n$ that will be placed in the server caches in order to achieve minimum possible access cost over all objects. The access costs are determined by the location of the object and by the request rates of the objects. We assume that a server can access an object with a cost of t_l, if it is stored in its own cache. The cost becomes t_r, when it has to access another replicator's cache to fulfill its client's request. The highest access cost is t_s, if that particular object is not stored in any server in the group and it is to be accessed from the origin or source of that object. Obviously, $t_l \leq t_r \leq t_s$. A server s_i knows the request rates r_{ij}, $j = 1, \ldots, n$, of its local users for all objects. We denote by $\mathbf{r}_i =$

$(r_{i1}, r_{i2}, \ldots, r_{in})$, the vector of request rates of the users at server s_i, and by $\mathbf{r} = (r_1, r_2, \ldots, r_m)$, the $m \times n$ matrix of request rates of all the objects at all servers. We denote by X the placement matrix, an $m \times n$ matrix whose entries are given by:

$$X_{ij} = \begin{cases} 1, & \text{if object } o_j \text{ is replicated at server } s_i \\ 0, & \text{otherwise} \end{cases}$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

The system goal is to minimize the access time at each server over all objects. We define the system goal using the following arguments. If no objects are replicated, the total access time at server s_i for object o_j is $r_{ij}t_s$. The access time reduces to $r_{ij}t_r$ if object o_j is replicated somewhere in the replication group but not at server s_i. In this case, the gain in access time is $r_{ij}(t_s - t_r)$. When object o_j is replicated at server s_i, the access time is $r_{ij}t_l$. In this case, the gain is $r_{ij}(t_s - t_l)$ or equivalently, $r_{ij}(t_s - t_r) + r_{ij}(t_r - t_l)$. Based on these arguments we formally define the optimization problem as:

$$\max \sum_{i=1}^{m} \left(\sum_{j:rc_j>0} r_{ij}(t_s - t_r) + \sum_{j:X_{ij}=1} r_{ij}(t_r - t_l) \right) \quad (1)$$

subject to:

$$X_{ij} \in \{0, 1\}, \quad i = 1, \ldots, m; j = 1, \ldots, n \quad (2)$$

$$\textstyle\sum_{j=1}^{n} X_{ij} \leq c_i, \quad i = 1, \ldots, m \quad (3)$$

where $rc_j = \sum_{i=1}^{m} r_{ij}$, is the 'replica count' of object o_j.

The first term of the objective function represents the gain obtained by replicating objects within the replication group, while the second term represents the additional gain obtained by replicating objects locally at server s_i. The first constraint says that an object j is either allocated or not allocated to server s_i. The second constraint, which is the capacity constraint, says that the number of objects allocated to server s_i should not exceed the capacity c_i of server s_i.

III. Distributed Replica Placement Algorithm

A. Preliminaries

We propose a distributed approximation algorithm, called DGR (Distributed Greedy Replication), that solves the replica placement problem. The algorithm has as input five parameters, $\mathbf{r}$, $\mathbf{c}$, t_s, t_r, and t_l. The first parameter, $\mathbf{r}$, is the matrix of request rates as defined in the previous section. The second parameter, $\mathbf{c} = (c_1, \ldots, c_m)$ is the m-vector of server's capacities. The last three parameters, are the access costs of the objects from source, remote and local replicas, respectively. In order to describe the algorithm we define two additional parameters, 'insertion gain' and 'eviction cost'. The *insertion gain* for object o_j and server s_i is defined as follows:

$$ig_{ij} = \begin{cases} p_j(t_s - t_r) + r_{ij}(t_r - t_l), & \text{if } rc_j = 0 \\ r_{ij}(t_r - t_l), & \text{if } X_{ij} = 0, rc_j > 0 \\ 0, & \text{if } X_{ij} = 1 \end{cases} \quad (4)$$

where $p_j = \sum_{i=1}^{m} r_{ij}$ is the 'popularity' of object o_j.

As can be seen from the definition of ig_{ij}, it represents the increase in overall gain the system would experience if it replicates object o_j in server s_i's cache. The highest insertion gain is for an object which does not have a replica in the group. It reduces to only the local gain of a server when that object is already replicated elsewhere. Otherwise, it is zero.

The *eviction cost* of object o_j at server s_i is defined as:

$$ec_{ij} = \begin{cases} 0, & \text{if } X_{ij} = 0 \\ r_{ij}(t_r - t_l), & \text{if } X_{ij} = 1, rc_j > 1 \\ p_j(t_s - t_r) + r_{ij}(t_r - t_l), & \text{if } X_{ij} = 1, rc_j = 1 \end{cases} \quad (5)$$

The eviction cost, ec_{ij} is the decrease in the system gain that would happen if object o_j is evicted from server s_i's space. The eviction cost has the highest value for an object that has only one replica in the group, since evicting this object will cause all servers to access it from the origin.

The insertion gain and the eviction cost are used to characterize each 'local' decision of replicating or evicting an object from a server. In making these decisions the algorithm considers the effect of replicating the objects on the overall system gain.

B. The Proposed Algorithm

The proposed distributed approximation algorithm for replica placement is given in Algorithm 1. The algorithm is executed by each server within the replication group. It starts with an initialization phase (lines 2 to 7) in which the servers initialize their local variables and compute the popularity of each object. In order to compute the popularity, p_j, of each object o_j, all servers participate in a collective communication operation called all-reduce-sum [9]. This collective communication operation is defined by the communication primitive all-reduce-sum($\mathbf{r}_i$, $\mathbf{p}$) which works as follows. Before executing the primitive each server has a vector $\mathbf{r}_i = (r_{i1}, \ldots r_{in})$ of size n, and as the final result of the primitive execution each server will contain a vector $\mathbf{p} = (p_1, \ldots, p_n)$ whose entries are given by $p_j = \sum_{i=1}^{m} r_{ij}$. Thus, all-reduce-sum computes the popularity of each object, and the result (the popularity vector) is made available at each server. In line 4, the algorithm initializes row i of the allocation matrix X to zero, that means no objects are allocated. It also initializes the available capacity e_i to c_i, the capacity of server s_i. The insertion gain for each object is initialized to the maximum value which corresponds to the case in which no replica exists in the replication group. The eviction cost and the replica count for each object are initialized to 0.

The second phase of the algorithm is the iterative phase, consisting of the while loop in lines 13 to 51. Before entering the loop, the global maximum insertion gain, ig_{max}, is computed through another collective communication operation called all-reduce-max ($send_msg$, $recv_msg$) (lines 8 to 11). The parameters are the send buffer and the receive buffer, respectively. Both are ordered lists of four variables (ig_{max}, i, j, j'), where ig_{max} is the maximum insertion gain, i and j are the indices of the corresponding server and object that gives the maximum insertion gain, and j' is the object to be evicted, if necessary. To participate in this operation each server s_i determines its highest insertion gain ig_{max} and the

Algorithm 1 DGR($\mathbf{r}_i$, c_i, t_s, t_r, t_l)

1: {Server s_i:}
2: {Initialization}
3: **all-reduce-sum**($\mathbf{r}_i, \mathbf{p}$)
4: $X_i \leftarrow \mathbf{0}$; $e_i \leftarrow c_i$
5: **for** $j := 1$ to n **do**
6: $\quad ig_{ij} \leftarrow r_{ij}(t_r - t_l) + p_j(t_s - t_r)$; $ec_{ij} \leftarrow 0$; $rc_j \leftarrow 0$
7: **end for**
8: $ig_{max} \leftarrow \max_k ig_{ik}$; $j \leftarrow \arg\max_k ig_{ik}$
9: $send_msg \leftarrow (ig_{max}, i, j, 0)$
10: **all-reduce-max**($send_msg, recv_msg$)
11: $(ig_{max}, i', j, j') \leftarrow recv_msg$
12: {i' is the server that has ig_{max} for object j;
j' is the object to be evicted from server i' (0 if none)}
13: **while** $ig_{max} > 0$ **do**
14: $\quad$ **if** $i' = i$ **then**
15: $\quad\quad$ {this server has the maximum insertion gain}
16: $\quad\quad X_{ij} \leftarrow 1$
17: $\quad\quad ec_{ij} \leftarrow ig_{ij}$; $ig_{ij} \leftarrow 0$; $e_i \leftarrow e_i - 1$; $rc_j \leftarrow rc_j + 1$
18: $\quad\quad$ **if** $j' \neq 0$ **then**
19: $\quad\quad\quad X_{ij'} \leftarrow 0$
20: $\quad\quad\quad ig_{ij'} \leftarrow ec_{ij'}$; $ec_{ij'} \leftarrow 0$;
21: $\quad\quad\quad e_i \leftarrow e_i + 1$; $rc_{j'} \leftarrow rc_{j'} - 1$
22: $\quad\quad$ **end if**
23: $\quad$ **else**
24: $\quad\quad$ {another server has the maximum insertion gain}
25: $\quad\quad rc_j \leftarrow rc_j + 1$
26: $\quad\quad$ **if** $X_{ij} = 0$ **then**
27: $\quad\quad\quad ig_{ij} \leftarrow r_{ij}(t_r - t_l)$
28: $\quad\quad$ **else**
29: $\quad\quad\quad ec_{ij} \leftarrow r_{ij}(t_r - t_l)$
30: $\quad\quad$ **end if**
31: $\quad\quad$ **if** $j' \neq 0$ **then**
32: $\quad\quad\quad rc_{j'} \leftarrow rc_{j'} - 1$
33: $\quad\quad\quad$ **if** $X_{ij'} = 1$ and $rc_{j'} = 1$ **then**
34: $\quad\quad\quad\quad ec_{ij'} \leftarrow r_{ij'}(t_r - t_l) + p_{j'}(t_s - t_r)$
35: $\quad\quad\quad$ **end if**
36: $\quad\quad$ **end if**
37: $\quad$ **end if**
38: $\quad$ {prepare the next iteration}
39: $\quad ig_{max} \leftarrow \max_k ig_{ik}$; $j \leftarrow \arg\max_k ig_{ik}$
40: $\quad ec_{min} \leftarrow \min_k(ec_{ik} : ec_{ik} > 0)$; $j' \leftarrow \arg\min_k(ec_{ik} : ec_{ik} > 0)$
41: $\quad$ **if** $e_i = 0$ or $c_i - e_i \geq n$ **then**
42: $\quad\quad$ **if** $ig_{max} \leq ec_{min}$ **then**
43: $\quad\quad\quad ig_{max} \leftarrow 0$; $j' \leftarrow 0$
44: $\quad\quad$ **end if**
45: $\quad$ **else**
46: $\quad\quad j' \leftarrow 0$
47: $\quad$ **end if**
48: $\quad send_msg \leftarrow (ig_{max}, i, j, j')$
49: $\quad$ **all-reduce-max**($send_msg, recv_msg$)
50: $\quad (ig_{max}, i', j, j') \leftarrow recv_msg$
51: **end while**

object o_j that gives this highest gain (lines 8-9). There is no object for eviction at this point, so $j' = 0$. We shall discuss more about j' later in this subsection. In line 9, each server s_i prepares the buffer $send_msg$ with ig_{max} and the indices i, j, and 0 for j'. The primitive all-reduce-max will return the $send_msg$ with highest ig_{max} to each server through the output buffer $recv_msg$ (line 10). After all-reduce-max execution each server s_i knows the global maximum insertion gain ig_{max} and the server and the object that has this ig_{max}. It also knows the index j' of the object to be evicted if needed. At this point the servers are ready to enter the main loop (line 13) of the algorithm.

During each iteration, if server s_i has the maximum global gain for an object, it performs allocation and, if necessary, deallocation of objects (lines 14 to 22). If s_i does not have the maximum global gain, it only updates some local values to keep track of the changes that resulted from allocation/deallocation of objects at other servers (lines 24 to 37). These updates are performed according to equations 4 and 5. Allocation (deallocation) of object o_j at server s_i is performed by setting the X_{ij} entry of the allocation matrix to 1 (0). Replica count, rc_j, and available capacity, e_i, are incremented, respectively decremented, in case of allocation. The reverse is done for deallocation. In case of allocating an object, the ig value before the allocation becomes the new ec value for that object (this is according to equations 4 and 5). For example, if before the allocation, object o_j does not have any replica in the group (i.e., $rc_j = 0$), the value of ig_{ij} is equal to the first entry in equation 4. After the allocation, $X_{ij} = 1$ and $rc_j = 1$, so the value of ec_{ij} is equal to the third entry in equation 5. This holds true for all other cases and, therefore, we can assign ig_{ij} to ec_{ij} when we allocate o_j to s_i and do the reverse when we evict o_j from s_i. Obviously, the insertion gain becomes zero after an insertion and the eviction cost becomes zero after an eviction. The eviction happens if $j' \neq 0$, when object o_j is evicted from the server s_i's cache (lines 18-22).

Lines 24 to 37 simply update rc, ig and ec, since another server $s_{i'}$ ($i' \neq i$) performed the allocation and s_i needs to keep track of it. rc is incremented or decremented and equations 4 and 5 are used to update ig and ec. If another server replicates o_j, server s_i updates the values of its insertion gain and eviction cost for o_j (lines 26-30). If object $o_{j'}$ was evicted from another server (i.e., $j' \neq 0$), the replica count is decremented and the insertion gain and the eviction cost corresponding to $o_{j'}$ are updated. If object $o_{j'}$ is replicated only at s_i, then server s_i updates only the eviction cost, $e_{ij'}$. If the object is not replicated at any server in the group, then s_i updates only the insertion gain, $ig_{ij'}$. Then, each server participates in another all-reduce-max operation that determines the next candidate server and object(s). Each server prepares the $send_msg$ as follows. The maximum insertion gain ig_{max} and j are determined as before. Server s_i also determines a candidate object $o_{j'}$ for eviction. This is the object that has the minimum eviction cost at s_i. A server is eligible to be considered for an allocation only if one of the following holds: it has available capacity to store more

objects, or it is full but some inserted object $o_{j'}$ has its eviction cost less than the insertion gain of some uninserted object o_j. Otherwise, it reports its ineligibility by setting ig_{max} to 0. In line 40, ec_{min} and j' are determined, and ec_{min} is compared with ig_{max} only when the available capacity, $e_i = 0$. If both eligibility conditions fail, ig_{max} is set to zero. If $e_i > 0$ then server s_i has space for new objects and, hence, no eviction is necessary ($j' = 0$). The algorithm terminates when each server reports $ig_{max} = 0$.

IV. Analysis of DGR

In this section, we analyze the complexity of the DGR algorithm and determine its approximation ratio. Due to the space limitation, we present only the results without proofs. The proofs will be provided in an extended version of this paper. We first present the complexity results. One can easily show that the running time of DGR is $O(n + C \log n)$ and that its communication complexity is $O((n+C) \log m)$, where $C = \sum_{i=1}^{m} c_i$ is the total capacity of the replication group. The approximation guarantee provided by DGR is given by the following theorem.

Theorem 1. *DGR is a 2-approximation algorithm for the replica placement problem.*

The following is a sketch of the proof of this theorem. We show that $OPT/DGR \leq 2$, where OPT and DGR denote the total gain by the optimal and DGR allocations, respectively. An equivalent expression is $(OPT - DGR)/DGR \leq 1$ or, $OPT - DGR \leq DGR$. To determine $OPT - DGR$ we characterize the difference in allocation by simple 'operations' like allocating or deallocating objects, etc. We show that any DGR allocation can be converted into an optimal one by a finite set of such operations. An operation increases, decreases, or does not affect the total system gain. We term their effect on the system gain as the 'gain' of the operations. Let OP be the set of operations that converts a particular DGR allocation into the optimal one. Therefore,

$$\sum_{op \in OP} G_{op} = OPT - DGR$$

where G_{op} is the gain of the operations that change DGR into OPT. It is sufficient to show that

$$\sum_{op \in OP} G_{op} \leq DGR \tag{6}$$

to prove that DGR is a 2-approximation algorithm. The details of the proof will be given in an extended version of this paper.

V. Experimental results

In this section, we study the proposed algorithm by simulation. We wrote a simulation program in Java that implements the DGR algorithm, and an exhaustive search algorithm that finds the optimal solution for the replica placement problem. An instance of this problem is defined by the number of servers m, the number of objects n, the object access costs t_s, t_r, t_l, the upper and lower limits of server capacities c^{max}, c^{min}, and the upper and lower limits of the request rates r^{max}, r^{min}. We generate the request rates and server capacities randomly in our experiments. We examined the effects of these parameters by varying m, n and c^{max} while we keep $t_s = 7$, $t_r = 3$, $t_l = 1$ fixed. Table I shows the gain in access time obtained by DGR and the optimal algorithm in these experiments. We see that the values are pretty close to each other and that in one case the DGR algorithm yielded the optimal outcome. Unfortunately, we could not test with more combinations of the values since the search space explodes even for small values of the parameters. The expected size of the search space is $\binom{n}{c^{avg}}^m$, where $c^{avg} = (c^{max} - c^{min})/2$.

TABLE I
Comparison of Results: Varying Servers and Objects

m	n	c^{max}	DGR	OPT	OPT/DGR
8	6	2	69566	69714	1.002127476
8	5	3	53538	53538	1
4	12	4	32808	32908	1.003048037
4	15	3	52450	53100	1.012392755
4	25	2	38532	38590	1.001505242
8	6	4	62502	62648	1.002335925

From these results, we observe that our approximation ratio is far below 2, and we expect this to happen for the majority of cases. Thus, we expect our algorithm to perform very well in practice providing solutions close to the optimum. As future work, we plan to implement the proposed algorithm on a real system and perform extensive evaluation of its performance.

Acknowledgment. This research was supported in part by NSF grant DGE-0654014.

References

[1] C. Chekuri and S. Khanna, "A PTAS for the multiple knapsack problem," in *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000, pp. 213–222.

[2] D. B. Shmoys and E. Tardos, "An approximation algorithm for the generalized assignment problem," *Math. Program.*, vol. 62, no. 3, pp. 461–474, 1993.

[3] T. Moscibroda and R. Wattenhofer, "Facility location: distributed approximation," in *PODC '05: Proceedings of the twenty-fourth annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2005, pp. 108–117.

[4] D. P. Bertsekas and D. A. Castanon, "The auction algorithm for the transportation problem," *Ann. Oper. Res.*, vol. 20, no. 1-4, pp. 67–96, 1989.

[5] A. Leff, J. L. Wolf, and P. S. Yu, "Replication algorithms in a remote caching architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 11, pp. 1185–1204, 1993.

[6] I. Baev, R. Rajaraman, and C. Swamy, "Approximation algorithms for data placement problems," *SIAM Journal on Computing*, vol. 38, no. 4, pp. 1411–1429, 2008.

[7] B. Chun, K. Chaudhuri, H. Wee, M. Barreno, C. H. Papadimitriou, and J. Kubiatowicz, "Selfish caching in distributed systems: a game-theoretic analysis," in *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2004, pp. 21–30.

[8] N. Laoutaris, O. Telelis, V. Zissimopoulos, and I. Stavrakakis, "Distributed selfish replication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 12, pp. 1401–1413, Dec. 2006.

[9] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, January 2003, ch. 4.