Implementing Replica Placements: Feasibility and Cost Minimization

Thanasis Loukopoulos[1], Nikos Tziritas[1], Petros Lampsas[2], and Spyros Lalis[1]

[1]*Department of Computer and Communications Engineering*
University of Thessaly
Glavani 37, 38221 Volos, Greece
{luke, nitzirit, lalis}@inf.uth.gr

[2]*Department of Informatics and Computer Technology*
Technological Educational Institute of Lamia
3rd km Old Ntl. Rd., 35100 Lamia, Greece
plam@teilam.gr

Abstract

Given two replication schemes X^{old} and X^{new}, the Replica Transfer Scheduling Problem (RTSP) aims at reaching X^{new}, starting from X^{old}, with minimal implementation cost. In this paper we generalize the problem description to include special cases, where deadlocks can occur while in the process of implementing X^{new}. We address this impediment by introducing artificial (dummy) transfers. We then prove that RTSP-decision is NP-complete and propose two kinds of heuristics. The first attempts to replace dummy transfers with valid ones, while the second minimizes the implementation cost. Experimental evaluation of the algorithms illustrates the merits of our approach.

1. Introduction

Data replication is commonly used in distributed systems to increase performance and availability [22]. Some well-known problems that must be addressed in this context: replica placement (i.e. deciding which data object to duplicate on which servers) [17], replica discovery and request redirection [12], and maintaining consistency between replicas [1]. In this paper we focus on a related scheduling problem, referred to as the Replica Transfer Scheduling Problem (RTSP) [14]. A generic description of RTSP is as follows: given *M* servers, *N* objects, an existing replication scheme X^{old} and a new scheme X^{new} we wish to implement (the latter being presumably the output of a replica placement algorithm), find a sequence of object transfers and deletions for transforming X^{old} into X^{new} with the minimum cost.

1-4244-0910-1/07/$20.00 ©2007 IEEE

Although a significant amount of work exists on replica placement, little has been done to tackle RTSP (see related work). In this paper we formulate RTSP as a cost optimization problem and prove that the relevant decision problem is NP-complete. We identify special cases of interest where reaching X^{new} cannot be guaranteed due to a deadlock, and extend our formulation to address this problem. Based on our observations we develop heuristics for producing the minimum cost schedule. Experiments demonstrate the efficacy of our approach.

The rest of the paper is organized as follows. Section 2 illustrates the motivation behind the problem and discusses related work. Section 3 presents the system model and gives the problem formulation. Section 4 describes the heuristics which are experimentally evaluated at Section 5. Finally, Section 6 concludes the paper.

2. Motivation and previous work

2.1. Motivation

RTSP arises in distributed systems that employ replication to minimize the overall client access cost. Consider for instance a distibuted video server system [5]. If only a single copy of each popular movie exists, the respective host servers (as well as the network for accessing them) will most likely be overloaded from user requests. It is thus meaningful to create multiple copies for the most popular movies and distribute them to different servers. However, user preferences change with time: a previously popular movie gradually becomes (relatively) unpopular after most users have viewed it, while new movies are constantly added to the system. This means that the replication scheme for the movies must be changed after a certain time period has elapsed (e.g. on a daily basis). The aim of RTSP is to minimize the transition cost when moving from one replication scheme to another.

The transition from the previous replication placement X^{old} to the next one X^{new} involves object transfers towards new servers and deletions of old replicas. Depending on the target system, different optimization parameters and goals can be considered. We adopt a network distance metric which has been used in the past to describe access cost in content distribution networks (CDNs) and distributed Web servers [9], [13], [22]. Hence, the goal of an RTSP algorithm is to produce a schedule for implementing this transition with the minimum possible network/communication cost due to object transfers between servers.

2.2. Previous work

RTSP has been mostly tackled in the past as part of the placement problem [10], [13]. With this approach the implementation cost of each replica creation is incorporated in the target function that is evaluated when deciding for the placement scheme. However, a placement algorithm is inferior when deciding upon the transfer schedule compared to an RTSP algorithm, the reason being that the former must decide without knowing the final X^{new} it will reach. In [13] we motivated the case and in [14] demonstrated the benefits of tackling RTSP separately. In this paper we provide a general RTSP formulation that tackles deadlock cases and develop new heuristics to deal with the problem. These heuristics are evaluated together with the winner algorithms of [14] (i.e. GOLCF and OP1 illustrated in Sec. 4.2).

Considerable work has been done in replica placement under various contexts e.g., video and Web [22] servers, content distribution networks [20] and the Grid [8]. For comprehensive summaries the interested reader is referred to [10], [18]. Furthermore, a large literature exists on problems related to scheduling, e.g. task scheduling in parallel systems [6], [11] and vehicle sequencing [3], [7]. Of particular interest are works tackling the problem of scheduling tasks that require data transfers [4], [8], [19], [21]. Although data transfers are involved, the described algorithms in these papers are not directly applicable here, since they operate over a fixed task graph. In contrast no fixed task graph exists in RTSP since (among other things) transfers might be done towards arbitrary intermediate nodes). Furthermore, RTSP aims at minimizing the cost of replica transfers instead of meeting time criteria. However, we believe that research in the above described task scheduling problem and on RTSP are rather complimentary even if stemmed from different research areas. Therefore as part of our future work we plan to study RTSP when X^{new} must be reached within a time deadline, as well as its possible applications to task scheduling problems.

3. Problem formulation

3.1. System model

Consider a generic distributed system with M servers and N data objects. Let S_i and $s(S_i)$ denote the name and the storage capacity (measured in abstract data units, e.g. bytes) of the ith server, $1 \le i \le M$. Also, let O_k and $s(O_k)$ denote the kth data object and its size, $1 \le k \le N$. Let X be a $M \times N$ *replication matrix* used to encode a replication placement as follows: X_{ik} equals 1 if S_i is a replicator of O_k , else X_{ik} equals 0. Servers communicate via (virtual) point-to-point links. Let l_{ij} denote the communication cost (per data unit) between S_i and S_j. We assume that l_{ij} is fixed and $l_{ij} = l_{ji}$. Let $S_{N(i,k,X)}$ and $S_{N2(i,k,X)}$ denote the “nearest” and “second-nearest” (cheapest in terms of communication cost) replicator of O_k for S_i in the replication scheme X . Note that $N(i,k,X)$ and $N2(i,k,X)$ are defined only if there exist at least one and two replicators of O_k , respectively.

3.2. RTSP definition

Let T_{ikj} denote the transfer of object O_k to server S_i using S_j as the source, and D_{ik} denote the deletion of O_k at S_i. Let $H = \{A_1, A_2, ..., A_t\}$ denote a *schedule* of t such *actions*. Also, let X^u and X^{u+1} denote the replication matrix before and after A_u, respectively.

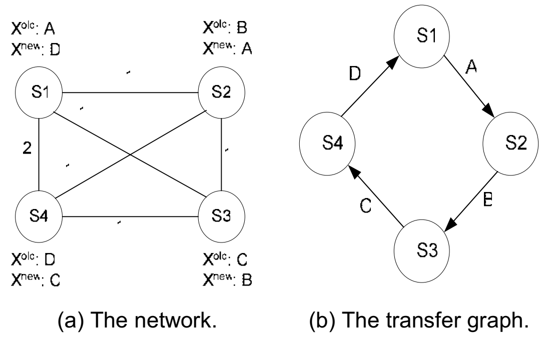
A transfer action $A_u = T_{ikj}$ is valid iff S_j is a replicator of O_k ($X^u_{jk} = 1$), S_i is not a replicator of O_k ($X^u_{ik} = 0$) and has free storage for hosting a copy thereof ($s(S_i) - \sum_{\forall k'} X^u_{ik'} s(O_{k'}) \ge s(O_k)$). Similarly, a delete action $A_u = D_{ik}$ is valid iff S_i is a replicator of O_k ($X_{ik}^u = 1$). Each action A_u transforms the (current) replication matrix X^u to X^{u+1} depending on its type: $A_u = T_{ikj} \Rightarrow X_{ik}^{u+1} = 1$ and $A_u = D_{ik} \Rightarrow X_{ik}^{u+1} = 0$. A schedule $H = \{A_1, A_2, ...A_t\}$ is valid with respect to (X^1, X^{t+1}) if it defines a sequence of valid actions that transform X^1 into X^{t+1} in a stepwise fashion.

Finally, let $C(H_u)$ be the cost of the uth action in schedule H: $C(H_u) = s(O_k)l_{ij}$ if $A_u = T_{ikj}$, or 0 if $A_u = D_{ik}$. Hence the *implementation cost* of schedule $H = \{A_1, A_2, ...A_t\}$: $I^H(X^1, X^{t+1}) = \sum_{u=1}^{t} C(H_u)$ (1).

RTSP can then be stated as: *given two replication schemes X^{old} and X^{new}, find a schedule H that is valid with respect to (X^{old}, X^{new}) and minimizes the implementation cost (1).*

3.3. Feasibility issues

As defined, RTSP does not always have a solution. Consider the example of Fig. 1(a) involving 4 servers and 4 objects (A, B, C and D). All objects are of equal size and all the servers have enough capacity to hold only one object. Here there exists no valid schedule for implementing X^{new} based on X^{old}. This is more clearly presented by drawing the transfer graph, i.e. a directed graph with nodes depicting system servers and arcs named after objects, where the following holds: for each *outstanding* replica (that needs to be created according to the new replication placement), there are arcs from each potential source towards the destination. Fig. 1(b) depicts the transfer graph of the network of



(a) The network. (b) The transfer graph.

Figure 1. Example of an infeasible RTSP problem statement.

Fig. 1(a). Observe that the transfers form a circle and a deadlock-like situation occurs. For instance, in order to transfer object D to S_1, S_1 must first delete object A to free space for the transfer, which results in S_2 not being able to obtain A since the only source of it is deleted.

To tackle feasibility issues such as this, we extend the RTSP formulation to include an artificial source holding copies of all objects. We call this artificial source the *dummy server* (S_d) and refer to the transfers that use this server as a source (e.g. T_{ikd}) as *dummy transfers*. To be consistent with the rest of the formulation we treat the dummy server just as any other server with the exception that the cost for using it as a source is the largest among all servers. More specifically, we set the link cost between it and any other server to be $a \cdot (\max(l_{ij}) + 1)$ where $a \geq 1$ is a constant. In its extended version, RTSP is guaranteed to have a solution (as long as X^{new} does not violate the storage constraints of the servers). The worst case obviously corresponds to the following scenario: First delete every replica in all servers but the dummy, and then perform all required object transfers from the dummy server.

We note that the existence of a dummy server which contains all objects but is (very) expensive to access is not far from reality. For instance when a new (or a very old / unpopular) movie is added in a distributed video server system, the first replica is most probably created by reading some (very slow) deep archival system. The fact that we treat such cases as normal server-to-server transfers is merely a convenient abstraction. By making the cost of dummy transfers sufficiently large (as a function of constant a) any algorithm that minimizes the implementation cost will also minimize the number of dummy transfers, replacing them with transfers from other (normal) servers. By allowing a to take values less than 1, it is also possible to model situations where creating a replica using other means, instead of proper network transfers, is more efficient. In this paper we do not investigate such cases.

3.4. Proof of NP-completeness

Clearly, RTSP-decision belongs to NP since given a schedule *H*, checking whether it is valid with respect to (X^{old}, X^{new}), as well as calculating its cost can be done in polynomial time. Following, we prove that RTSP-decision is NP-complete by reducing the (0,1) Knapsack-decision to it.

The (0,1) Knapsack-decision problem can be defined as follows [15]: Given *n* objects, having benefit values $b_1, b_2, ..., b_n$ and sizes $s_1, s_2, ..., s_n$, is there a subset W of the objects, such as $\sum_{i \in W} s_i \le S$ and $\sum_{i \in W} b_i \ge K$ (b_i, s_i, S, K are positive integers $\forall i$). Assuming an instance of the (0,1) Knapsack-decision we construct an equivalent RTSP instance as follows: We consider a network of $M = n+3$ servers $S_1, ..., S_{n+3}$ and $N = n+1$ objects $O_1, ..., O_{n+1}$. Objects $O_1, ..., O_n$ correspond to the n Knapsack objects ($s(O_i) = s_i$), while O_{n+1} is a dummy object of size $s(O_{n+1}) = \sum_{1 \le i \le n} s(O_i)$. For each Knapsack object O_i we define S_i to be a replicator of it. $S_1, ..., S_n$ store nothing else (other than the relevant object replica) in X^{old}. S_{n+1} is a server of capacity $s(S_{n+1}) = S + \sum_{1 \le i \le n} s(O_i)$ (where S is the Knapsack size). S_{n+1} only stores O_{n+1} in X^{old}. S_{n+2} is a server of capacity $s(S_{n+2}) = \sum_{1 \le i \le n} s(O_i)$ storing the Knapsack objects in X^{old}. Finally, S_{n+3} is a server only holding a replica of O_{n+1}. The following links exist: (i) a link between S_{n+1} and S_{n+2} with link cost 1, (ii) links between $S_1, ..., S_n$ and S_{n+1}, each of link cost $l_{in+1} = b_i^{'}$, $1 \le i \le n$, where $b_i^{'} = \frac{b_i \prod_{1 \le i \le n} s(O_i)}{s(O_i)}$, (iii) a link between S_{n+3} and S_{n+2} of cost $\sum_{1 \le i \le n} (b_i^{'} + 1)$. Fig. 2 explains the above.

X^{new} is set to be identical to X^{old}, with the exception that S_{n+1} and S_{n+2} must interchange objects. Consider, an optimal order of actions *H-OPT*, implementing X^{new}. We observe that the transfer of O_{n+1} from S_{n+3} cannot belong to *H-OPT* since the involved cost $\sum_{1 \le i \le n} (b_i^{'} + 1) \sum_{1 \le i \le n} s(O_i)$ is larger than the total cost of the schedule (let H') that starts with the transfer of O_{n+1} from S_{n+1} and continues with the transfers of all Knapsack objects from $S_1, ..., S_n$, for a total cost of $\sum_{1 \le i \le n} s(O_i) + \sum_{1 \le i \le n} b_i^{'} s(O_i)$. Thus, *H-OPT* must contain the transfer of O_{n+1} from S_{n+1}.

We also observe that $H' \neq$ *H-OPT*, since S_{n+1} starts with *S* unused storage space, which can be used to transfer at least one Knapsack object (let O_i) at a cost lower than performing the transfer from S_i ($s(O_i)$ compared to $b_i^{'} s(O_i)$). Therefore, *H-OPT* begins with a sequence of Knapsack object transfers from S_{n+2} to S_{n+1}, followed by the transfer of O_{n+1} from S_{n+1} to S_{n+2}, followed by the transfers of the remaining Knapsack objects from $S_1, ..., S_n$ to S_{n+1}. Let $W^{'}$ be the set of objects that appear in the *H-OPT* schedule before the transfer of O_{n+1}. The cost of *H-OPT* is thus given by:

$$\sum_{i \in W^{'} \wedge i \neq n+1} s(O_i) + \sum_{1 \le i \le n} s(O_i) + \sum_{i \notin W^{'} \wedge i \neq n+1} b_i^{'} s(O_i)$$

But *H-OPT* is the schedule of minimum possible cost, meaning that $W^{'}$ is selected (the exact order with which $W^{'}$ is selected has no impact at the cost) such that the following is minimized:

$$\sum_{i \in W^{'} \wedge i \neq n+1} s(O_i) + \sum_{1 \le i \le n} s(O_i) + \sum_{i \notin W^{'} \wedge i \neq n+1} b_i^{'} s(O_i) \quad (2)$$

After substitutions (2) gives: min($\sum_{i \in W^{'} \wedge i \neq n+1} s(O_i) + \prod_{1 \le i \le n} s(O_i) \sum_{i \notin W^{'} \wedge i \neq n+1} b_i$), since $\sum_{1 \le i \le n} s(O_i)$ is constant. But notice, that the following holds:

$$\prod_{1 \le i \le n} s(O_i) b_i \ge s(O_i) \forall 1 \le i \le n \quad (3)$$

Thus, (2) reduces to min($\prod_{1 \le i \le n} s(O_i) \sum_{i \notin W^{'} \wedge i \neq n+1} b_i$), which since $\prod_{1 \le i \le n} s(O_i)$ is constant, gives min($\sum_{i \notin W^{'} \wedge i \neq n+1} b_i$). Therefore, we conclude that *H-OPT* minimizes $\sum_{i \notin W^{'} \wedge i \neq n+1} b_i$ that is equivalent to maximizing $\sum_{i \in W^{'} \wedge i \neq n+1} b_i$, which is the problem statement of the (0,1) Knapsack optimization problem.

The following concludes the reduction: given a (0,1) Knapsack-decision instance, we create a network as above and ask whether there exists a valid schedule *H*: $I(X^{old}, X^{new}) \le \sum_{\forall i} s_i + (\sum_{\forall i} b_i - K) \prod_{\forall i} s_i + S$ (by (2)

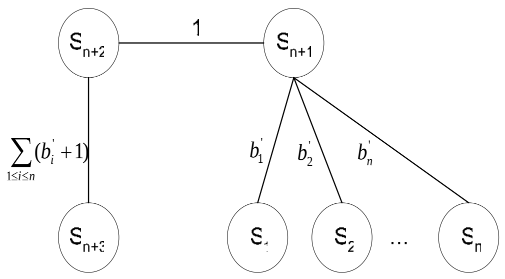


Figure 2. Network structure for reducing Knapsack to RTSP.

and (3)). If H exists so does a solution to the (0,1) Knapsack-decision instance.

4. Scheduling heuristics

In the sequel we describe our heuristics. We start by describing heuristics that aim at minimizing the number of dummy transfers and proceed with algorithms that optimize the implementation cost of the transfer schedule.

4.1. Minimizing dummy transfers

The algorithms here are divided into two categories. The first one takes as input X^{old}, X^{new} and attempts to build a schedule where no dummy transfers occur. As discussed in the previous section, this may not always be feasible. Furthermore, since these are heuristics rather than exhaustive search algorithms, they may produce a schedule with dummy transfers even though one without a dummy transfer may be (theoretically) feasible. The second category takes as input an existing schedule H^{old} that is valid with respect to (X^{old}, X^{new}) and produces a new schedule H^{new} with a smaller number of dummy transfers. Ideally, these heuristics may completely eliminate the number of dummy transfers.

Build initial schedule: Random Deletions First (RDF). RDF starts with an empty schedule and performs first all the deletions of *superfluous* replicas, i.e. replicas for which $X_{ik}^{new} = 0$ and $X_{ik}^{old} = 1$, followed by all transfers of *outstanding* replicas, i.e. replicas for which $X_{ik}^{new} = 1$ and $X_{ik}^{old} = 0$. The order in which both deletions and transfers are performed is random, while the nearest source is selected in each transfer. Note that since all deletions happen at the beginning of the schedule it is possible that the last replica of an object for which an outstanding transfer exists is deleted. In this case the corresponding object transfer action will use the dummy server as a source.

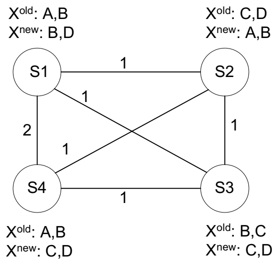


Figure 3. A network example.

To better understand the process, consider the network of Fig. 3 with 4 servers and 4 objects named after A, B, C, D. All objects are of equal size and all servers have enough capacity to store 2 objects. The figure shows the link costs between servers, and for each server what objects it holds in X^{old} and what objects it should hold in X^{new}. By applying the algorithm, the schedule starts with all deletions in random order. For instance one possible outcome is: $\{D_{1A}, D_{4B}, D_{3B}, D_{2C}, D_{4A}, D_{2D}\}$. The algorithm then randomly adds transfers as long as there are outstanding replicas. One possible resulting schedule (where servers with outstanding replicas are considered in the order of S_1, S_4, S_3, S_2, S_2, S_4) is the following: $\{D_{1A}, D_{4B}, D_{3B}, D_{4A}, D_{2D}, D_{2C}, T_{1Dd}, T_{4C3}, T_{3D1}, T_{2B1}, T_{2Ad}, T_{4D3}\}$. Notice that the transfer sequence begins with a dummy transfer of object D at server S_1 (T_{1Dd}). Subsequent transfers of object D (e.g. T_{3D1}) can be done from the newly created source. Furthermore, the transfer of D to S_4 (last action in the schedule) uses S_3 as source instead of S_1 since $l_{34} = 1 < l_{14} = 2$.

Build initial schedule: Grouped by Server Deletions First (GSDF). GSDF builds a valid schedule as follows. It selects servers randomly and for each one it performs deletions for its superfluous replicas followed by transfers for its outstanding replicas. Again, transfers are done from the nearest source or the dummy server if no source is available at that point. For example in the network of Fig. 3 one possible schedule built by GSDF could be the following (servers are considered in the order of S_2, S_3, S_4, S_1): $\{D_{2C}, D_{2D}, T_{2A1}, T_{2B1}, D_{3B}, T_{3Dd}, D_{4A}, D_{4B}, T_{4C3}, T_{4D3}, D_{1A}, T_{1D3}\}$. The rationale of GSDF is that by performing all the outstanding transfers of a single server just after all its superfluous replicas are deleted and before considering the outstanding replicas for other servers, the resulting schedule will have less dummy transfers compared to RDF. Also notice that the server which is selected first by the algorithm will never need any dummy transfers since the only replicas deleted at this point are its own superfluous replicas.

Initial schedule exists: Move dummy transfers prior to deletions (H1). H1 works on top of an existing schedule with the aim of eliminating dummy transfers. More specifically it scans the schedule from left to right and whenever a dummy transfer is identified it attempts to move it before some deletion of a respective replica. Consider for instance the schedule produced by RDF in the example of Fig. 3 : { D_{1A}, D_{4B}, D_{3B}, D_{4A}, D_{2D}, D_{2C}, T_{1Dd}, T_{4C3}, T_{3D1}, T_{2B1}, T_{2Ad}, T_{4D3}}. H1 will try first to restore T_{1Dd} by moving the transfer before the first deletion of object D it comes across as it scans the schedule starting from T_{1Dd} and moving towards its beginning. In the example, the resulting schedule after T_{1Dd} is restored to validity will be as follows: {…, D_{4A}, T_{1D2}, D_{2D}, D_{2C}, T_{4C3}, T_{3D1}, ...}.

Restoring dummy transfers by moving them before respective deletions is not always possible though. Consider for instance what happens when attempting to restore T_{2Ad}. According to the above method the resulting schedule will be: { D_{1A}, D_{4B}, D_{3B}, T_{2A4}, D_{4A}, T_{1D2}, D_{2D}, D_{2C}, T_{4C3}, T_{3D1}, T_{2B1}, T_{4D3}}. However this schedule is invalid since when T_{2A4} is reached, the capacity constraint of server S_2 will be violated (neither object C nor D has been deleted from S_2 prior to T_{2A4} thus a transfer is done towards a server with no adequate capacity to hold the object). Such capacity violations can be repaired by moving adequate deletions before the shifted dummy transfer. In the example this amounts to moving either D_{2C} or D_{2D} before T_{2A4}, resulting in schedule { D_{1A}, D_{4B}, D_{3B}, D_{2C}, T_{2A4}, D_{4A}, T_{1D2}, D_{2D}, T_{4C3}, T_{3D1}, T_{2B1}, T_{4D3}}. Notice that a further case of interest rises, if instead of D_{2C} we decided to move D_{2D}. In this case T_{1D2} should be moved as well (otherwise it will lead to a dummy transfer) and the resulting schedule will be the following: { D_{1A}, D_{4B}, D_{3B}, T_{1D2}, D_{2D}, T_{2A4}, D_{4A}, D_{2C}, T_{4C3}, T_{3D1}, T_{2B1}, T_{4D3}}. As already discussed, moving T_{1D2} might result in capacity violation at S_1 if it overpasses any deletions involving S_1 (in the example it does not).

Summarizing, the algorithm attempts to move a dummy transfer before a deletion of a respective replica so that it can be changed into a valid transfer, e.g. before: H ={.., D_{jk}, G_1, T_{ikd}, ...}, after: H'={.., T_{ikj}, D_{jk}, G_1, ...}, where G_1 represents a sub-schedule containing deletions and transfers. In doing so, it checks for the following cases:

(i) *No deletions at* G_1 *involving* S_i: If G_1 contains no deletions of the form $D_{ik'}$ then H' is valid and the algorithm proceeds with the next dummy transfer.

(ii) *Storage capacity violation - repairable with standalone deletions*: If G_1 contains deletions of the form $D_{ik'}$ then by moving T_{ikd} the capacity constraint of S_i might be violated. In this case the algorithm checks whether by moving the standalone deletions of S_i (i.e. deletions of the form $D_{ik'}$ that are not preceded by any transfer of the form $T_{i''k'i}$ in G_1) before T_{ikj} the capacity constraint is met, e.g. G_1={ $G_{1.1}$, $D_{ik'}$, $G_{1.2}$ } before: H ={.., D_{jk}, $G_{1.1}$, $D_{ik'}$, $G_{1.2}$, T_{ikd}, ...}, after H'={.., $D_{ik'}$, T_{ikj}, D_{jk}, $G_{1.1}$, $G_{1.2}$, ...}. If so, the algorithm accepts the change and proceeds.

(iii) *Storage capacity violation - non repairable with standalone deletions*: In case no standalone deletions for S_i exist or they do not suffice to restore the capacity constraint, the algorithm considers moving deletions together with the transfers that precede them, e.g. before: H ={.., D_{jk}, $G_{1.1}$, $T_{i''k'i}$, $D_{ik'}$, $G_{1.2}$, T_{ikd}, ...}, after H'={.., $T_{i''k'i}$, $D_{ik'}$, T_{ikj}, D_{jk}, $G_{1.1}$, $G_{1.2}$, ...}. If the move of $T_{i''k'i}$ occurs without capacity violation at $S_{i''}$, H' is accepted. Otherwise, H1 recursively attempts to restore validity by considering the equivalent to H' schedule H''={.., $D_{ik'}$, T_{ikj}, D_{jk}, $G_{1.1}$, $T_{i''k'd}$, $G_{1.2}$, ...}, i.e. by treating $T_{i''k'i}$ as a dummy transfer and checking for it the conditions (i)-(iii). Notice that after each recursion the sub-schedule G_1 that separates the dummy transfer from the respective deletion will decrease (e.g. becomes $G_{1.1}$ after the first call). When it reaches Ø the recursion will terminate and the algorithm will keep the resulting schedule if it is valid otherwise it will backtrack to the initial schedule H, leaving the original dummy transfer (T_{ikd}) as is and proceeding with the next dummy transfer.

Initial schedule exists: Create superfluous replicas (H2). H2 uses a complimentary approach to H1 in restoring dummy transfers. Namely, it creates additional free storage space or takes advantage of available storage to introduce superfluous replicas that will act as proper sources for dummy transfers. The algorithm takes as input a schedule H which is scanned from left to right until a dummy transfer $T_{i'kd}$ is encountered. It then identifies the first deletion $D_{i''k}$

of O_k preceding $T_{i'kd}$ and attempts to inject a new transfer of O_k at a server S_i immediately before the deletion $D_{i''k}$. Notice that H1 would attempt to move $T_{i'kd}$ before $D_{i''k}$ which might not be possible if the capacity at $S_{i'}$ is violated. Instead, H2 will take advantage of any server that has enough free space in order to restore $T_{i'kd}$. After $T_{i'kd}$ is restored the superfluous replica created is deleted.

Assuming H is of the form {G_1, $D_{i''k}$, G_2, $T_{i'kd}$, G_3} and that S_i has enough free space to store O_k, the resulting schedule H' will be {G_1, $T_{iki''}$, $D_{i''k}$, G_2, $T_{i'ki}$, D_{ik}, G_3}. In case no server has enough free space to store O_k, the algorithm attempts to create space by performing deletions of superfluous replicas, provided that at least one replica will still exist for each object. If freeing enough space is impossible, the original schedule H is restored and $T_{i'kd}$ is left as a dummy transfer, otherwise the schedule is updated with the superfluous transfer and the necessary deletions. The algorithm then proceeds with checking the next dummy transfer of the schedule.

4.2. Minimizing implementation cost

The algorithms of this category aim at minimizing the implementation cost (rather than trying to eliminate dummy transfers). In doing so, it is possible that dummy transfers are replaced with valid ones, however this happens as a side-effect. Here too, we distinguish the algorithms depending on whether they operate on an existing schedule (OP1) or build one from scratch (AR, GOLCF).

All Random (AR). The outstanding replicas to be created and the superfluous replicas to be deleted, if needed, are chosen randomly.

Greedy Object Lowest Cost First (GOLCF) [14]. Each superfluous replica O_k on S_i is associated with a benefit value B_{ik} equal to the cost difference for transferring outstanding replicas of O_k on all S_j for which S_i is the nearest replicator, via S_i or the second-nearest replicator:

$$B_{ik} = s(O_k) \sum_{\forall j: N(j,k,X)=i} l_{jN2(j,k,X)} - l_{jN(j,k,X)} \quad (4).$$

The algorithm picks an object O_k at random and iteratively transfers it to all severs that require a replica thereof. In each iteration, the server S_i with the lowest communication cost from the currently nearest source of O_k (i.e. $l_{iN(i,k,X)}$) is selected. If it is necessary to delete one or more superfluous replicas on S_i, these are chosen in order of increasing benefit values as per (4). When there is no outstanding replica for O_k, the next object is picked. The algorithm terminates when all objects have been considered. The motivation is that by focusing on the "full" replication of one object at a time, it is possible to optimize the order of the corresponding transfers.

Initial schedule exists: Changing action order (OP1) [14]. Schedule H is scanned from left to right until a transfer action $T_{i'kj'}$ is encountered. Scanning is continued until another transfer T_{ikj} for the same object O_k is found. Assume H is of the form {…, $T_{i'kj'}$, G_1, T_{ikj}, G_2}, where G_1 and G_2 are sub-schedules containing transfer and deletion actions. The algorithm then considers moving T_{ikj} before $T_{i'kj'}$ in order to reduce the implementation cost for all subsequent transfers found in the schedule. For each transfer involving O_k in {$T_{i'kj'}$, G_1, G_2}, the respective benefit of moving T_{ikj} is equal to 0 if $l_{i''i} > l_{i''j''}$, else it is equal to the cost difference between transferring O_k on $S_{i''}$ via the currently used source $S_{j''}$ and S_i, i.e. $s(O_k)(l_{i''j''} - l_{i''i})$. The cost for implementing the transfer of O_k on S_i at the uth position of the schedule is $s(O_k) l_{iN(i,k,X^u)}$. The algorithm considers modifying the schedule only if the total benefit outweighs the implementation cost, in which case the transfers of O_k that benefit from this change are also updated to use S_i as their source.

However, additional validity checks are required to decide whether to consider such a modification. Let $D_{il_1..l_m}$ denote the sequence of deletions D_{il_1}, D_{il_2}, …, D_{il_m}. Then, in the general case, H is of the form {…, $D_{i'k_1..k_n}$, $T_{i'kj'}$, G_1, $D_{il_1..l_m}$, T_{ikj}, G_2}, where $D_{i'k_1..k_n}$ and $D_{il_1..l_m}$ are the deletions performed on $S_{i'}$ and S_i to enable transfers $T_{i'kj'}$ and T_{ikj}, respectively. The suggested reordering (before updating the sources of subsequent transfers of O_k) results in schedule H'={…, $D_{il_1..l_m}$, $T_{ikN(i,k,X^u)}$, $D_{i'k_1..k_n}$, $T_{i'kj'}$, G_1, G_2}, i.e. superfluous deletions for server S_i are brought before T_{ikj}, which is further evaluated according to the following special cases:

(i) *No crucial deletions*: If no deletions $D_{il1..lm}$ precede T_{ikj} and G_1 does not contain any deletions itself, schedule $H' = \{\ldots,\ T_{ikN(i,k,X^u)},\ D_{i'k_1..k_n},\ T_{i'kj'},\ G_1,\ G_2\ \}$ is valid and is adopted.

(ii) *Void transfers and deletions*: If G_1 is of the form $\{G_{1.1},\ T_{ikj''},\ G_{1.2},\ D_{ik},\ G_{1.3}\}$, schedule $H' = \{\ldots,\ D_{il_1...l_m},\ T_{ikN(i,k,X^u)},\ D_{i'k_1..k_n},\ T_{i'kj'},\ G_{1.1},\ T_{ikj''},\ G_{1.2},\ D_{ik},\ G_{1.3},\ G_2\ \}$ is invalid because $T_{ikj''}$ creates a second replica of O_k on S_i as it now follows $T_{ikN(i,k,X^u)}$. Similarly, if G_1 is of the form $\{G_{1.1},\ T_{il'j''},\ G_{1.2}\}$, where $l_1 \le l' \le l_m$, schedule $H' = \{\ldots,\ D_{il_1...l_m},\ T_{ikN(i,k,X^u)},\ D_{i'k_1..k_n},\ T_{i'kj'},\ G_{1.1},\ T_{il'j''},\ G_{1.2},\ G_2\ \}$ is invalid because it contains a deletion $D_{il'}$ for replica $O_{l'}$ on S_i that does not exist; it will be created at a later stage via $T_{il'j''}$. In both cases, H' is dropped.

(iii) *Outdated transfer sources*: If G_1 is of the form $\{G_{1.1},\ T_{j''l'i},\ G_{1.2}\}$, where $l_1 \le l' \le l_m$, schedule $H' = \{\ldots,\ D_{il_1...l_m},\ T_{ikN(i,k,X^u)},\ D_{i'k_1..k_n},\ T_{i'kj'},\ G_{1.1},\ T_{j''l'i},\ G_{1.2},\ G_2\ \}$ is invalid since $T_{j''l'i}$ assumes that S_i is a replicator of $O_{l'}$, but this replica is deleted via $D_{il'}$ earlier on. Nevertheless, H' can be made valid by substituting S_i with $S_{N(j'',l',X^u)}$, assuming that $T_{j''l'i}$ is the u^th action in H'. Updating each such outdated transfer may however introduce an additional penalty equal to $s(O_{l'})(l_{j''N(j'',l',X^u)} - l_{j''i})$, which must be taken into account.

(iv) *Capacity constraint violation*: If G_1 is of the form $\{G_{1.1},\ D_{il'},\ G_{1.2}\}$, where $l' \ne l_1,...,l_m$, schedule $H' = \{\ldots,\ D_{il_1..l_m},\ T_{ikN(i,k,X^u)},\ D_{i'k_1..k_n},\ T_{i'kj'},\ G_{1.1},\ D_{il'},\ G_{1.2},\ G_2\ \}$ is invalid if the deletion of $O_{l'}$ on S_i was required, in addition to deletions $D_{il1..lm}$, to create space for O_k and to enable transfer T_{ikj} in schedule H. In this case, H' can be made valid by moving $D_{il'}$ before $T_{ikN(i,k,X^u)}$, resulting in schedule $\{\ldots,\ D_{il_1..l_m,l'},\ T_{ikN(i,k,X^u)},\ D_{i'k_1..k_n},\ T_{i'ki},\ G_{1.1},\ G_{1.2},\ G_2\ \}$. This in turn requires checking for outdated transfer sources in sub-schedule $G_{1.1}$ as per (iii).

The schedule is changed in case (i), or in cases (iii) and (iv) provided that the benefit outweighs the respective implementation cost as well as the *total* penalty for adjusting *all* outdated transfer sources. Each time the schedule is changed, it is scanned from start. The algorithm terminates if the entire schedule is scanned without being able to introduce any changes.

5. Experiments

Here we present results from the experimental evaluation. Sec. 5.1 describes the simulation parameters while Sec. 5.2 illustrates the results. Due to space restrictions we are forced to omit a large part of our evaluation, leaving it for an extended version.

5.1. Experimental setup

The server network was generated using the BRITE tool [15], for 50 server nodes each having a connectivity of 1, resulting in a tree-graph. Node connections followed the Barabasi-Albert model, which has been used to describe power-law router graphs [2]. Links were assigned a fixed cost, uniformly distributed between 1 and 10. Server-to-server communication costs were set equal to the aggregated link cost along the shortest paths. A set of 1,000 objects was used and the constant factor a that controls the cost of dummy transfers was set to 1. In all the experiments we measure the number of dummy transfers left in the schedule and the implementation cost.

5.2. Results

In the first experiment we set the size of all objects to 5,000 data units and varied the number of replicas for each object. Fig. 4 and 5 plot the results as the number of replicas for each object varies from 1 to 5. All servers store the same number of objects in X^{old} which remains unaltered in X^{new}. Server capacities were set to be equal, and sufficient to just satisfy X^{old} and X^{new} without leaving any additional free space. The allocation of objects to the servers is performed randomly in X^{old}. X^{new} is the result of the servers interchanging their objects in a way that X^{old} and X^{new} have no common replicas (overlap 0%).

Fig. 4 depicts the number of dummy transfers for the cases where H1 and H2 are applied over AR and GOLCF. We can observe that as more replicas per object exist in X^{old}, the number of dummy transfers

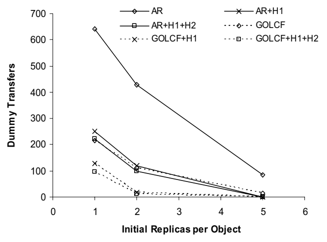


Figure 4. Number of dummy transfers as the replicas per object increase (equal object sizes).

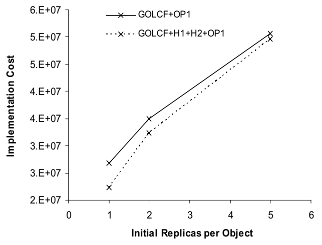


Figure 5. Implementation cost as the replicas per object increase (equal object sizes).

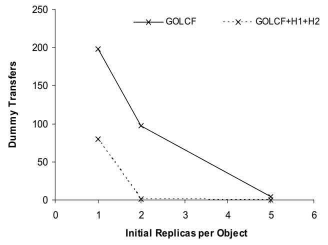


Figure 6. Number of dummy transfers as the replicas per object increase (uniform object sizes).

drops. This is due to the fact that with more replicas available, the probability that either AR or GOLCF deletes the last replica of an object decreases. As anticipated, GOLCF is a better choice compared to AR which builds a schedule in a completely random fashion. Especially noteworthy is the improvement achieved by applying H1 and H2 (almost nullifying dummy transfers in the two replicas per object case). As a result H1 and H2 also manage to reduce the implementation cost of the GOLCF+OP1 schedule (Fig. 5). The combinations of H1+H2 with RDF and GSDF resulted in similar trends and are not shown.

The second experiment is similar to the first one with the exception that object size is varied uniformly between 1,000 and 5,000. Fig. 6 and 7 show the dummy transfers and the implementation cost respectively. In Fig. 6 we only plot GOLCF variants. Here too, H1+H2 appear to have the largest contribution in minimizing dummy transfers which results in large implementation cost savings as shown in Fig. 7.

In the last experiment the setup was similar to the first experiment, i.e. equally sized objects, no common replicas between X^{old} and X^{new}, replicas equally distributed to servers and servers having the minimum capacity sufficient to store the objects specified by X^{old} and X^{new}. However, we wanted to test the behavior of the algorithms when free storage space is available in the system. Therefore, we fixed the number of replicas per object to 2 and introduced at a random server additional capacity to store one more object. Fig. 8 and 9 show the results as the number of the servers having extra capacity rises. We can observe that the remaining dummy transfers after applying H1+H2 drop as the capacity increases. This implies that the H1+H2 combination explores the extra space more efficiently compared to standalone GOLCF (the corresponding plot is almost flat). As a result, the implementation cost of GOLCF+H1+H2+OP1 is smaller compared to standalone GOLCF+OP1 (Fig. 9).

Summarizing the experiments, GOLCF+OP1 provides a good starting point for building a schedule, while the application of H1 and H2 drastically improves it by reducing the number of dummy transfers at the initial schedule.

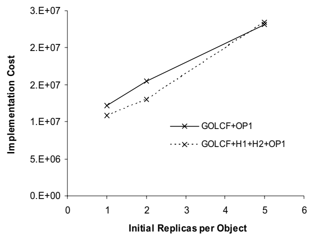


Figure 7. Implementation cost as the replicas per object increase (uniform object sizes).

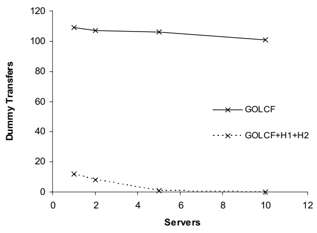


Figure 8. Number of dummy transfers as more servers acquire extra capacity.

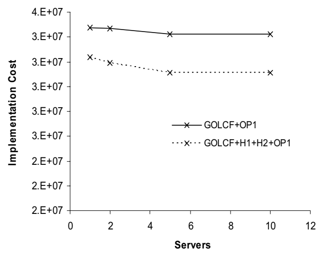


Figure 9. Implementation cost as more servers acquire extra capacity.

6. Conclusions

In this paper we investigated the replica transfer scheduling problem taking into account feasibility and cost optimization issues. We introduced various heuristic operators and evaluated their performance also with reference to a previously developed algorithm (GOLCF+OP1). Results demonstrate that especially H1 and H2 contribute largely on optimizing the transfer schedule, making GOLCF+H1+H2+OP1 a clear winner over other alternatives. Our ongoing work includes an extended version with a larger experimental evaluation.

Acknowledgements

This work was partially supported by the European Social Fund and National Resources - (EPEAEK-II) ARCHIMIDIS II.

References

[1] T. Anderson, Y. Breitbart, H. Korth and A. Wool, "Replication, Consistency and Practicality: Are These Mutually Exclusive?," *in Proc. ACM SIGMOD'98*, Seattle, June 1998.

[2] A.L. Barabasi and R. Albert, "Emergence of Scaling in Random Networks", in *Science*, Vol 286, pp. 509-512, Oct. 1999.

[3] C. Basnet, L. Foulds and J. Wilson, "An exact algorithm for a milk tanker scheduling and sequencing problem," in *Annals of Operations Research*, Vol. 86, pp. 559-568, 1999.

[4] O. Beaumont, A. Legrand and Y. Robert, "Optimal Algorithms for Scheduling Divisible Workloads on Heterogeneous Systems," in Proc. *17th Int. Parallel and Distributed Processing Symp. (IPDPS 2003)*, Nice, France, 2003.

[5] C. Bisdikian and B. Patel, "Cost-based program allocation for distributed multimedia-on-demand systems," *IEEE Multimedia*, vol. 3, no. 3, pp. 62–72, 1996.

[6] T. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys and B. Yao, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," in *Journal of Parallel and Distributed Computing (JPDC)*, Vol. 61(6), pp. 810 – 837, June 2001.

[7] N. Christofides, "Vehicle Routing," in *The Traveling Salesman Problem*, Lawler, Lenstra, Rinooy Kan and Shmoys, eds., John Wiley, pp. 431-448, 1985.

[8] F. Desprez and A. Vernois, "Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid," Research Report RR2005-01, INRIA, France, Jan. 2005.

[9] J. Kangasharju, J. Roberts, and K. Ross, "Object replication strategies in content distribution networks," *Computer Communications*, vol. 25, no. 4, pp. 367–383, 2002.

[10] M. Karlsson and C. Karamanolis, "Choosing replica Placement Heuristics for Wide-Area Systems," in Proc. *ICDCS'04*, pp. 350-359.

[11] Yu-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," in *ACM Computing. Surveys*, Vol. 31(4), pp. 406-471, 1999.

[12] N. Laoutaris, G. Smaragdakis, A. Bestavros and I. Stavrakakis, "Mistreatrment in Distributed Caching Groups: Causes and Implications," in Proc. *IEEE INFOCOM* 2006, Barcelona, Spain.

[13] T. Loukopoulos, P. Lampsas, and I. Ahmad, "Continuous replica placement schemes in distributed systems", in Proc. *19th ACM International Conference on Supercomputing (ACM ICS)*, Boston, MA, June 2005.

[14] T. Loukopoulos, N. Tziritas, P. Lampsas and S. Lalis, "Investigating the Replica Transfer Scheduling Problem," to appear in Proc. *18th Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'06).*

[15] S. Martello and P. Toth, "Knapsack Problems: Algorithms and Computer Implementations," *John Wiley & Sons-Interscience Series in Discrete Mathematics and Optimization*, 1990.

[16] A. Medina, A. Lakhina, I. Matta, and J. Byers, BRITE: Boston University Representative Internet Topology Generator, http://cs-pub.bu.edu/brite/index.htm, March 2001.

[17] L. Qiu, V. Padmanabhan, and G. Voelker, "On the placement of web server replicas," in Proc. *IEEE INFOCOM*, April 2001, pp. 1587–1596.

[18] M. Rabinovich and O. Spatschek, "Web Caching and Replication," Addison-Wesley, 2002.

[19] T. Phan, K. Ranganathan and R.Sion, "Evolving Toward the Perfect Schedule: Co-Scheduling Job Assignments and Data Replication in Wide-Area Systems Using a Genetic Algorithm," in Proc. *11th Workshop on Job Scheduling Strategies for Parallel Processing* (*JSSPP 2005*), June 19, 2005.

[20] P. Radoslav, R. Govindan, and D. Estrin, "Topology informed Internet replica placement," *Computer Communications*, vol. 25, no. 4, pp. 384–392, 2002.

[21] O. Sinnen, L. Sousa and F. Sandnes, "Toward a Realistic Task Scheduling Model," in *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, Vol. 17(3), pp. 263-275, March 2006.

[22] X. Tang and J. Xu, "On Replica Placement for QoS-Aware Content Distribution," in Proc. *IEEE INFOCOM*, March 2004, Hong Kong.

[23] L. Zhuo, C. Wang, and F. Lau, "Load balancing in distributed web server systems with partial document replication," in Proc. *ICPP'02*, August 2002, pp. 305–312.A.B. Smith, C.D. Jones, and E.F. Roberts, "Article Title", *Journal*, Publisher, Location, Date, pp. 1-10.