On the Minimisation of Cost for Replica Migration

by Nikolaos Tziritas

A dissertation submitted to the
University of Thessaly

in partial fulfillment to the requirements of the degree of

MASTER OF SCIENCE

Accepted on the recommendation of:
Spyros Lalis, committee chair
Catherine Houstis, committee member
Panagiotis Mpozanis, committee member

1 October 2006

ACKNOWLEDGEMENTS

ABSTRACT

Although several replica placement algorithms have been proposed and studied in the literature, little research has been done so far on capturing and minimizing the cost for migrating from an existing replica placement to a new one. In this work we investigate the so called *Replica Transfer Scheduling Problem* (RTSP) which can be briefly described as follows: Given replica placements X^{old} and X^{new}, determine a sequence of object transfers and deletions to obtain X^{new} based on X^{old} with minimum transfer (network) cost. We study RTSP for the case where servers have limited storage capacity. It can be shown that the problem is NP-complete. Therefore, we propose several heuristics and compare their performance. The proposed heuristics fall in three categories: (i) algorithms derived from existing continuous replica placement heuristics, which take as input an existing placement and produce a new placement along with a schedule to implement it; (ii) algorithms that take as input two placements and compute a schedule to obtain the one from the other; (iii) algorithms for enhancing a given schedule. Results indicate that heuristics of type (ii) perform favorably compared to the algorithms of type (i), and that heuristics of type (iii) can optimize schedules generated by algorithms of type (i) and (ii). To our knowledge, this is the first time that RTSP has been studied as a separate problem.

TABLE OF CONTENTS

1 Introduction and background

The explosive growth of the Internet has turned centralized data servers into a performance bottleneck. Popular sites may receive millions of requests per day, which can easily lead to server and network overload; and consequently to increased service delay for the end user. Apart from conventional web content, this situation may also arise for multimedia (e.g. video) traffic or grid servers providing very large data sets to remote processing applications.

This problem can be addressed using two different approaches: by distributing client requests on many servers or other machines acting like servers; and by moving server contents closer to the clients. In the first case, per server load drops and client requests are processed faster. In the second case, requests and responses take less time to travel through the network, reducing the roundtrip delay for the client. Well known techniques, such as caching, replication and mirroring, are based on these principles. The next sections give a brief overview of the research done in this area.

1.1 Caching

Caching is an attempt to (temporarily) store the most commonly accessed data objects as close as possible to the clients that requested them. Ideally, a cache is kept on the same

machine as the client, thereby completely eliminating network access in case of a hit. Alternatively, the cache is placed on a machine that can be accessed with less overhead compared to the server that holds the (original) data. Caches are typically assumed to be of small size compared to the total amount of data stored on the server.

Caching is traditionally used in distributed client-server architectures, e.g. file systems like AFS [24], to enhance system performance in both LAN and WAN environments. It has also been widely studied in conjunction with the web, in which case cached items are web pages or parts thereof. A lot of research has been done on several aspects of caching, most notably: cache replacement policies [25], dynamic page caching [26, 27, 39], cache consistency [63, 50], pre-fetching [45, 60] and cache architectures [54, 69].

1.2 Replication

Replication is about actively deploying several machines for the purpose of processing client requests. It is commonly used to increase availability and fault tolerance as well as to boost system performance. Combined with a mechanism for distributing client requests on less busy servers, replication can be exploited to achieve load balancing and thus faster request processing. Moreover, by placing servers on different parts of the network, often referred to as mirroring, the client-server communication latency can be reduced[1].

The replication of a service on the Internet raises the problem of how client requests are forwarded to the available servers in a transparent way, giving the illusion of a single (powerful) service. The proposed approaches include client-side redirection [28, 35], router redirection [71], DNS redirection [43, 38, 30] and server-side redirection [49, 61, 59]. There is also the issue of server selection, based on various performance parameters,

---

[1] On the other hand, keeping a large number of widely distributed servers increases the cost of update propagation, which if done asynchronously may also introduce (temporary) inconsistencies.

and efficient dissemination of server status information; for related work see [72, 58, 40] and [33, 29], respectively.

It is worthwhile to note that caching can be thought as a special form of replication, for the case where servers hold only a part of the system data objects. This analogy leads to some interesting comparisons. For instance, cache replacement algorithms could be regarded as on-line, distributed, greedy algorithms for the creation of local data replicas under strict storage constraints [73]. Forwarding client requests that resulted in a cache miss to the server could also be viewed as a simple client-side redirection policy. In principle, every major aspect of a caching scheme has an equivalent mapping in a replicated system; without the opposite being true.

1.3 Replica placement

In order to deploy a replicated service one must first decide where to place the respective servers (or service / data replicas). There exist a wealth of system definitions and algorithms, each one attempting to capture and improve different performance aspects of this problem. Following, we classify research efforts in this area depending on their affiliation to well known theoretical problems.

k-Median problem: A graph is given with weights on the nodes representing the number of client requests and lengths on the edges representing network costs. Satisfying a request incurs network cost equal to the length of the shortest path between the client node and a server. The problem consists of placing k servers on the nodes so as to minimize the total network cost, provided that each node can hold at most one server. The k-Median problem has been shown to be NP-hard [74]. This formulation is used to tackle the problem of distributing a single replica over a fixed number of hosts. Most work assumes a replica to be a mirror server hosting all site contents, thus performing coarse grain replication. In

[36] the authors study the problem of placing M proxies at N nodes when the topology of the network is a tree, and propose an O(N^3M^2) dynamic programming based algorithm that finds the best solution. [44] provides a greedy heuristic that outperforms the method of [36] for the case of a general graph. [67] investigates the optimal placement of Internet distance measuring instrumentations under the IDMaps framework [32]. A more recent study [41] compares a 2-approximation algorithm for the k-Median with a greedy approach, a random algorithm and a heuristic which favors nodes of a higher outdegree for replica placement. The greedy heuristic achieves a smaller overall client-replica roundtrip delay, with the performance difference being more significant against the random placement.

Bin packing problem: Given N objects of various sizes, partition them to the minimum number of disjoint sets so that the cumulative size of each set does not exceed a given threshold. The problem is NP-hard. The bin packing formulation is commonly used to model load balancing problems. For instance, the problem of distributing documents in a cluster of web servers in order to balance their load is discussed in [53]. The paper proposes a binning algorithm for the initial distribution and network flow [66] formulations in the case of access patterns change or server failure. In [34] the authors propose a distributed protocol to load balance replicated servers along a tree hierarchy.

File Allocation problem (FAP): Given a network of M nodes with different storage capacities and N files exhibiting various read frequencies from each node, allocate the objects to the nodes so as to optimize a performance parameter (e.g., minimize total network traffic) while respecting the storage capacity of each node [65]. The problem is NP-complete [57]. In [70] the formulation is extended to account for multiple object copies and updates, and [52] provides an iterative approach that achieves good solution quality for the case where nodes have infinite capacity. A complete although old survey can be found in [48]. The file allocation problem originated from the need to allocate

programs and data to multiprocessor systems [42]. In its general formulation it can be viewed as a case of the uncapacitated facility location (UFL) problem [56], which was studied in the business management sector. FAP-like formulations have been used to describe similar problems arising in distributed databases [51, 62], multimedia databases [68] and video server systems [46, 37].

1.4 The cost of replica placement

The vast majority of work on the replica placement problem is concerned with the issue of calculating an optimal replica placement for a given client request pattern, but *ex vivo*, without assuming any current system state. More specifically, almost no work has been done so far taking into account the cost for implementing a new placement based on an existing one, which results due to the respective data (replica) transfers over the network.

This "delinquency" is mainly due to the fact that replication placement is viewed as a long term pre-fetching mechanism. As a consequence, the costs that will be incurred to achieve the desired replica placement are considered to be of secondary importance, assuming that they will be amortized over a long time period. However this is not always the case. Consider for example a distributed video server system where new and potentially popular movies arrive each day. It could be desirable to change the current replica placement relatively frequently, perhaps even on daily basis, in which case the implementation costs may be too important to ignore. We believe that this issue will become more important in the future given the increasing large-scale web hosting market and recent deployment of large content distribution networks [64, 55].

Some extended replica placement formulations tackle this problem by factoring the implementation strategy and respective cost into the replication decisions [61, 47]. This approach can indeed result in better solutions compared to simple formulations that do not

take into account this cost (as [47] demonstrates for the case of CDN networks). Still, it is interesting to study this issue as a separate problem. For example, one might wish to optimize the implementation of a certain replica placement regardless of how this was produced; perhaps not using a computer program, but based on manual administrative decisions. We refer to this as the *Replica Transfer Scheduling Problem* (RTSP), which is the research focus of our work.

1.5 Thesis overview

The rest of this thesis is organized as follows. In the Section 2, we describe our system model, and give respective formulations of the replica placement problem (RPP) and the replica transfer scheduling problem (RTSP). In Section 3, we give various heuristics for RTSP, in part refining existing algorithms that have been proposed for solving RPP. In Section 4, we analyze the performance of these heuristics based on simulations. In Section 5 we discuss related work. Finally, Section 6 concludes the thesis, identifies open issues and points towards possible future research directions.

2   System model and problem formulations

In the following we present our system model. We adopt a similar system model with the one used in [8]. For convenience, the notations introduced here and in the next section are summarized in the Appendix. Based on this model, we formulate the problem statements that are of relevance to our work, namely the Replica Placement Problem, the Continuous Replica Placement Problem and the Replica Transfer Scheduling Problem.

2.1 System model

Consider a generic distributed system consisting of M servers. Let S_i be the name and $s(S_i)$ the total storage capacity of ith server, where $1 < i < M$. Also, let there be N different data objects in the system. We denote the kth object by O_k and its size by $s(O_k)$, where $1 < k < N$.

The communication topology is a general graph, where servers communicate with each other directly via point-to-point links (if any), or indirectly via other servers. The per-byte link cost between servers S_i and S_j, denoted by l_{ij}, is equal to the aggregated cost of the corresponding "shortest" path. We assume that it remains fixed throughout system operation. We also assume that $l_{ij} > 0$ for $i \neq j$, $l_{ii} = 0$ and $l_{ij} = l_{ji}$.

Every object O_k has exactly one *primary replica* hosted on a distinguished *primary server* for that object, denoted by P_k. The primary replica of each object is chosen a priori and remains fixed. Additional copies of O_k may be hosted on other servers. Server S_i is called a *replicator* of O_k iff it holds a copy thereof. Obviously, P_k is a replicator of O_k. The replication state of the system is encoded in the form of an *MxN* matrix X, also referred to as the *replication matrix*, where element X_{ik} is 1 iff S_i is a replicator of O_k and 0 otherwise. Notably, a replica placement is *valid* iff it obeys the following two constraints:

$$\sum_{k=1}^{N} X_{ik} s(O_k) < s(S_i), \forall i \quad \text{(server storage constraint) (1)}$$

$$P_k = i \Rightarrow X_{ik} = 1, \forall k \quad \text{(primary replica constraint) (2)}$$

The client request processing model, shown in Figure 1, is as follows. Any server S_i may receive a client request for reading any object O_k. If S_i is a replicator of O_k the request is processed locally and the reply is sent back to the client, without incurring any overhead for the server network. Else, S_i forwards the client request to the "nearest" (in terms of communication cost) replicator of O_k, denoted by N_{ik}^X, and returns the reply it receives from N_{ik}^X back to the client. In this case the cost for the server network is proportional to the data (size of request plus size of reply) exchanged between S_i and N_{ik}^X and the respective link cost $l_{N_i^x}$. For reasons of symmetry, if S_i is a replicator of O_k we let $N_{ik}^X = i$ (this also allows us to simplify expressions that will be introduced in the sequel). It must be stressed that N_{ik}^X is a *function* of the replication matrix X; the notation N_{ik} is occasionally used in favor of readability.

2.2 The Replica Placement Problem (RPP and CRPP)

The aim of replica placement is to install copies of data objects on the servers in order to minimize the total communication cost due to client requests. Given our system model, the Replica Placement Problem (RPP) can be stated as follows: *For a given client request traffic profile, find a replication matrix X that minimizes the cost function (4) subject to the server storage capacity constraint (1) and the primary replica constraint (2).*

It is important to realize that this (classic) problem formulation, which is equivalent to the File Allocation Problem (FAP), adopts a *blank slate* approach, whereby the current state of the system is not taken into account. To address this limitation, the original problem statement can be extended to (i) include the *migration cost* (or *implementation cost*) that will be incurred to obtain the new replication placement based on the existing one, and (ii) capture the fact that this cost must not nullify the relative gain in client access cost that will be achieved by the new replica placement.

This can be expressed in the form of a benefit function:

$$B^{X^{old}, X^{new}} = C^{X^{old}} - C^{X^{new}} - I^{X^{old}, X^{new}} \quad (5)$$

where $C^{X^{old}} - C^{X^{new}}$ is the client access cost difference between the existing replica placement and the new one, and $I^{X^{old}, X^{new}}$ is the respective implementation cost. Obviously, it is desired to find a replica placement X^{new} that maximizes this function.

The problem can then be reformulated as follows: *For a given client request traffic profile and a current replication matrix X^{old}, find a new replication matrix X^{new} that maximizes the benefit function (5) subject to the server storage capacity constraint (1) and the primary replica constraint (2).* In the literature [14] this is referred to as the Continuous

Replica Placement Problem (CRPP) to underline the fact that the new replica placement is computed *in context* of an already existing one.

To compute a solution to CRPP one must compute the implementation cost $I^{X^{old},X^{new}}$. An upper bound can be determined by assuming the worst case, namely that every required copy of O_k on S_i is created by fetching the object from the respective primary server P_k. Conversely, the lower (most likely infeasible) bound corresponds to the best possible case where every required copy of O_k on S_i is created by using as a source server S_j with which S_i communicates at the lowest possible cost, i.e., for which $l_{ij} < l_{ij'}, \forall j' : j' \neq i$ holds.

2.3 The Replica Transfer Scheduling Problem (RTSP)

While these upper and lower bounds can be of practical significance to the problem of CRPP (e.g., they can be used as rough estimates in heuristics), it is also worthwhile to investigate how to compute this implementation cost more accurately. Ideally, one would like to determine the *minimum* cost for migrating from X^{old} to X^{new}, or (equivalently) to determine the cost of the *most efficient* implementation schedule. This gives rise to a separate and non-trivial problem, which we refer to as the Replica Transfer Scheduling Problem (RTSP).

To indicate the complexity of RTSP, we give a simple example, illustrated in Figure 2. Suppose there are four servers S_1, S_2, S_3 and S_4, with storage capacity 2, 2, 2 and 4, respectively, connected to S_2 via links of equal cost. Let there be four objects O_a, O_b, O_c and O_d of size 1. Let the current replication state be: S_1 has copies of O_a and O_b; S_2 has copies of O_b and O_c; S_3 has copies of O_a and O_c; and S_4 has copies of all objects. Let the desired replication state be: S_1 has copies of O_a and O_c; S_2 has copies of O_a

and O_d; S_3 has copies of O_a and O_b; and S_4 remains unchanged. In this case, the best way to migrate from the old replica placement to the new one is as follows: (1) delete O_b on S_1 and O_c on S_3; (2) transfer O_c from S_2 to S_1, and O_b from S_2 to S_3; (3) delete O_b and O_c on S_2; (4) transfer O_a and O_d from S_4 to S_2. If instead one starts by installing copies on S_2, i.e., by performing steps (3) and (4) first, then the penalty for installing the rest of the required copies on S_1 and S_3 (from S_4 via S_2) will lead to a bigger total cost. It turns out that the choice of the action(s) to be taken first is impossible to make without (exhaustively) trying out the alternatives.
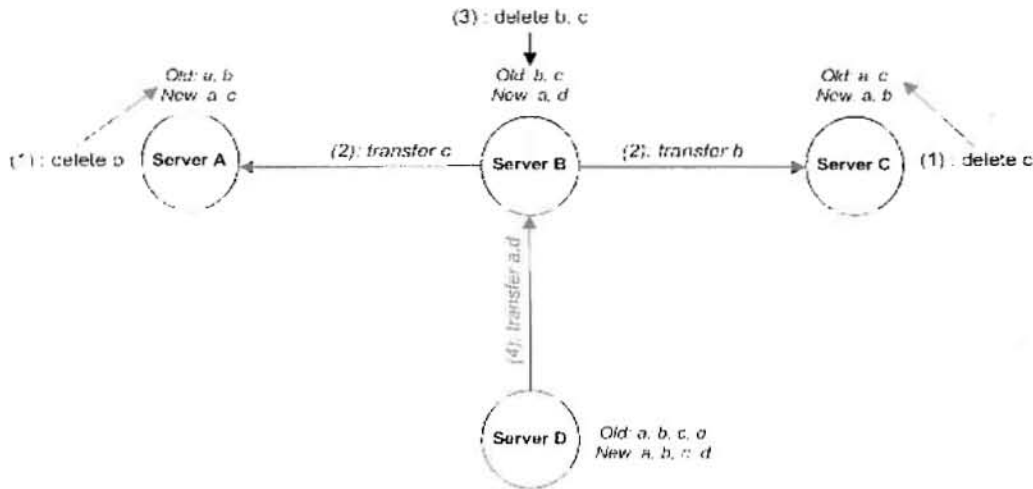
Figure 2. Example of migrating between two different replica placements

To express RTSP in a more formal manner, we introduce some additional notations. Let T_{ijk} denote the transfer of O_k from S_i to S_j. Let D_{ik} denote the deletion of O_k on S_i. Let $H = \{A_1, A_2, ..., A_t\}$ denote a schedule of t actions, where an action is an object transfer or deletion. Let X^{u+1} denote the replication matrix after the uth action (assuming an

initial replication matrix X^1). A transfer action $A_u = T_{ijk}$ is valid iff (i) $X_k^u = 1 \wedge X_{jk}^u = 0$ and (ii) $s(O_k) + \sum_{k'} X_{jk'}^u s(O_{k'}) < s(S_j)$, i.e., a transfer may take place only if the source server S_i is a replicator of object O_k, the destination server S_j is not a replicator of O_k, and S_j has enough space to store the new replica (server capacity constraint). A delete action $A_u = D_{ik}$ is valid iff (i) $X_k^u = 1$ and (ii) $i \neq P_k$, i.e., a copy can be deleted only if server S_i is a replicator but not the primary server of O_k (primary replica constraint). An action results in a transition of the current replica placement $X^u \rightarrow X^{u+1}$, as follows: $A_u = T_{ijk} \Rightarrow X_{jk}^{u+1} = 1$ and $A_u = D_{ik} \Rightarrow X_{ik}^{u+1}$. Schedule $H = \{A_1, A_2, ..., A_t\}$ is valid with respect to (X^1, X^{t+1}) iff it corresponds to a sequence of t valid actions, which transform X^1 into X^{t+1}. Finally, let C^{H_u} denote the cost of the uth action in schedule H as follows: $A_u = T_{ijk} \Rightarrow C^{H_u} = l_{ij} s(O_k)$ and $A_u = D_{jk} \Rightarrow C^{H_u} = 0$. Then the cost of a schedule $H = \{A_1, A_2, ..., A_t\}$ that is valid with respect to (X^1, X^{t+1}) is:

$$I_H^{X^0, X^{t+1}} = \sum_u C^{H_u} \quad (6)$$

We can now formulate the Replica Transfer Scheduling Problem (RTSP) as follows: *For given replica placements X^{old} and X^{new}, find a schedule $H = \{A_1, A_2, ..., A_t\}$ that is valid with respect to $(X^{old} = X^1, X^{new} = X^{t+1})$ and incurs the lowest possible cost (6).*

2.4 Discussion

RTSP is NP-complete. A proof is given in the Appendix. The "toughness" of RTSP is primarily due to the storage constraint. A brief explanation is given below.

Let us consider the general case where it is required to perform an object transfer towards a server that has not enough capacity to store it. To free space, it becomes necessary to delete one or more object copies from the server. Candidates for deletion are obviously the copies which are not required in the new replica placement. Nevertheless, by deleting a (superfluous) replica we eliminate a potential source, thereby possibly increasing the cost of future transfers for that object. Another related problem is to decide whether to proactively install superfluous copies for certain objects on certain servers that may act as better data transfer sources for subsequent (future) transfers. Of course, the degree to which this can be done is also limited by the storage capacity of the servers.

Given NP-completeness, RTSP cannot be (efficiently) tackled using brute force algorithms. This has motivated the design of several heuristics, which are presented in the next section.

3   Heuristics

We present three different types of heuristics for tackling the Replica Transfer Scheduling Problem. Heuristics of the first category are derived from algorithms that have been proposed to solve the Continuous Replica Placement Problem. The second category comprises algorithms that have been designed explicitly to solve the Replica Transfer Scheduling Problem. Finally, heuristics of the third category have the form of operators that can be applied to an existing schedule of replica transfers and deletions to enhance it.

3.1 CRPP-based heuristics

Several heuristics have been proposed to solve the Continuous Replica Placement Problem [14]. These algorithms take as input a client read traffic pattern $r_{ik}, \forall i, k$ and a replica placement X^{old}, and output a new replica placement X^{new} that is computed according to the CRPP formulation. In our work, we focus on the two most representative heuristics, and extend them to produce a valid schedule for migrating from X^{old} to X^{new}. Henceforth we will refer to these algorithms as CRPP-based heuristics.

3.1.1 Greedy Global (GG)

The Greedy Global algorithm starts with an initial replica placement $X = X^{old}$ and works in iterations to produce the final replica placement X^{new} along with the corresponding implementation schedule H. The pseudocode is given below:

```
X:= Xold; H := {}; b:= 0;
while positive_flip_exists() do
    repeat
        X' := X; H' := H;
        (i,k) := find_unmarked_positive_flip();
        mark_temp_flip(i,k);
        while !has_space_for(i,k) do
            k' := -1; b' := MIN_INTEGER;
            for k" := 1 to N
                if X'[i][k"] = 1 && Pk" != i then
                    X'[i][k"] := 0; cost := 0;
                    b" := CRPP_benefit_function(X,X',cost);
                    if b" > b' then k' := k"; b' := b"; end
                    X'[i][k"] := 1;
                end
            end
            if k' = -1 then break; // cannot free more space
            X'[i][k'] :=0;
            H' := H' + Dik';
        end
        if has_space_for(i,k) then
            j :- Nik;
            H' := H' + Tjik;
            X'[i][k] := 1; cost := s(Ok)lij;
            b' := CRPP_benefit_function(X,X',cost);
            if b' > b then bi := i; bk := k; b := b'; Xb := X'; Hb := H'; end
        end
    until !unmarked_positive_flip_exists();
    if b = 0 then break; // all options lead to a worse situation
    X := Xb; H := Hb; b :=0;
    unmark_temp_flips();
end
return (X,H);
```

In each iteration, all possible element flips $X_{ik} \rightarrow 1$, corresponding to transfers of O_k to S_i, are considered, and the one that maximizes the benefit function (5) subject to constraints (1) and (2) is chosen. Since the only difference between the "previous" and the "next" replica placement is the replication of O_k on S_t, the implementation cost of this transition is trivially equal to the cost for transferring the object from the current nearest replicator: $l_{iN_i} s(O_k)$. If the storage capacity of S_i does not suffice to store O_k, other replicas $O_{k'}$ on S_i are deleted in ascending order of their value according to the benefit function (5). The respective deletion actions are added to the schedule before the transfer action. The algorithm terminates when the storage capacity in each server is reached or any further replication creation results in a negative benefit.

3.1.2 Greedy Object Random (GOR)

The Greedy Object Random algorithm is similar to Greedy Global, but focuses on the replication of the same object at a time. The pseudocode follows:

```
X:= X^old; H := {};
while not_all_objects_considered() do
    k := random_pick_object();
    while positive_flip_exists_for(k) do
        // same as GG, but with k fixed
    end
end
return (X,H);
```

The algorithm starts by picking an object O_k at random, and performs the same routine as GG for this object. More specifically, in each iteration a single replica allocation is performed for the focus object, until no more beneficial replicas can be created for it. Then the next object is chosen at random, and the same process is repeated. The algorithm terminates when all objects have been considered.

3.2 RTSP heuristics

Algorithms of this category are designed exclusively for the purpose of tackling the Replica Transfer Scheduling Problem. They take as input two replica placements X^{old} and X^{new}, and produce a valid schedule H for migrating from X^{old} to X^{new}. All heuristics follow a common processing template, given in pseudocode below:

```
X := X^old; H := {};
while unmarked_outstanding_replica_exists(X, X^new) do
    (i,k) := pick_unmarked_outstanding_replica(X, X^new);
    mark_temp_flip(i,k);
    if has_space_for(i,k) then
        mark_perm_flip(i,k);
        j := N_ik;
        H := H + T_jik;
        X[i][k] := 1;
    end
end
unmark_temp_flips();
while unmarked_outstanding_replica_exists(X, X^new) do
    (i,k) := pick_unmarked_outstanding_replica(X, X^new);
    mark_perm_flip(i,k);
    while !has_space_for(i,k) do
        k' := pick_superflous_replica(i,X, X^new);
        H := H + D_ik';
        X[i][k'] := 0;
    end
    j := N_ik;
    H := H + T_jik;
    X[i][k] := 1;
end
while superfluous_replica_exists(X, X^new) do
    (i,k) := pick_superflous_replica(X, X^new);
    H := H + D_ik;
    X[i][k] := 0;
end
return (H);
```

The algorithms start from an initial replica placement $X = X^{old}$ and an empty schedule H, which is incrementally extended with transfer and deletion actions until the desired replica placement X^{new} is reached. In a first phase, *outstanding* replicas (that are required in X^{new} but are not available in X) which can be created on servers without violating the storage constraint are iteratively picked. For each such replica creation, a corresponding transfer action (using the currently nearest replicator as a source) is appended to the schedule. In a second phase, the same process is repeated but this time it is necessary to delete on the target servers one or more *superfluous* replicas (that are not required in X^{new} but available in X). Corresponding replica deletion actions are added to the schedule before the respective transfer action. The algorithms terminate when there are no more outstanding replicas. In a third (cosmetic) phase, deletion actions for any remaining superfluous replicas are scheduled; however this does not affect the implementation cost.

The heuristics presented in the sequel all operate based to this scheme. They differ only in the criteria used to pick the outstanding replicas to be installed (routine pick_unmarked_outstanding_replica in the pseudocode) and the superfluous replicas to be removed (routine pick_superfluous_replica in the pseudocode). These criteria are chosen so that the storage capacity (1) and primary replica constraints (2) are never violated, hence the produced schedules are valid. Notably, no attempt is made to proactively create additional superfluous replicas, which could potentially lead to a more efficient (in terms of cost) schedule.

3.2.1 All Random (AR)

The outstanding replicas to be created and the superfluous replicas to be deleted, if needed, are selected in random order, without any attempt to optimize the total communication cost. This algorithm serves as a low baseline for the next ones.

3.2.2 Least popular server first (LPSF)

Outstanding replicas are chosen with preference to the servers with the least *popular* superfluous replicas (in terms of them being nearest sources for other outstanding replica transfers). The motivation is that deleting these replicas, if needed, does not (directly) increase the cost of future transfers. Each superfluous replica O_k on S_i is assigned a popularity value P_{ik} equal to the number of outstanding replicas O_k on all other servers S_j for which S_i is the current nearest replicator: $P_{ik} = |\{S_j : N_{jk} = i\}|$. In each iteration, the server with the lowest aggregate popularity for all its superfluous replicas is chosen. If there are several outstanding replicas that need to be created on this server, one is picked randomly. If it is necessary to delete superfluous replicas on this server, these are chosen in increasing order of their popularity.

3.2.3 Least valuable server first (LVSF)

Outstanding replicas are chosen with preference to the servers with the least *valuable* superfluous replicas (in terms of the relative cost benefit for using them as nearest sources for outstanding replica transfers). The motivation is similar to LPSF but another metric is used in order to pick servers. Each superfluous replica O_k on S_i is associated with a benefit value B_{ik} equal to the difference cost for transferring outstanding replicas of O_k on all servers S_j for which S_i is the current nearest replicator N_{jk}, via S_i or the second-nearest replicator (referred to as $N2_{jk}$):

$$B_{ik} = s(O_k) \sum_{\forall j: N_{jk}=i} l_{jN2_{jk}} - l_{jN_k} \quad (7)$$

Similarly to LPSF, the server with the lowest aggregate benefit value of its superfluous replicas is chosen in each iteration, and the superfluous replicas are deleted in increasing

order of their benefit. As in LPSF, if there are several outstanding replicas for the chosen server, one is picked randomly.

3.2.4 Highest Opportunity Cost First (HOCF)

The next outstanding replica is chosen as to minimize the implementation *opportunity* cost. The motivation is to prioritize outstanding replicas that will become expensive to implement if the corresponding nearest transfer source is deleted. Each outstanding replica O_k on S_i is associated with an opportunity cost V_{ik} which is equal to 0 if the replica held by the nearest replicator N_{ik} is not superfluous (i.e. $X_{N_{ik}k}^{new} = 1$), else is equal to the cost difference between transferring O_k on S_i via the nearest and the second-nearest source: $V_{ik} = s(O_k)(l_{iN2_k} - l_{iN_k})$. In each iteration, the outstanding replica with the highest opportunity cost is chosen. If it is necessary to delete one or more superfluous replicas on the target server, these are chosen in increasing benefit values as in LVSF (7). Outstanding replicas with zero opportunity cost are chosen last, in random order.

3.2.5 Greedy Object Lowest Cost First (GOLCF)

This heuristic adopts a slightly different approach. In a similar fashion as GOR, in a top-level loop a new focus object O_k is randomly picked at a time, which is then iteratively replicated on all servers that require a copy.

In each iteration, the server S_i with the lowest communication cost to its currently nearest replicator N_{ik} is selected: $l_{iN_{ik}} < l_{i'N_{i'k}}, \forall i'$. If it is necessary to delete one or more superfluous replicas on S_i these are chosen in increasing benefit values as in LVSF (7). When there are no more outstanding replicas for O_k, the next focus object is picked and the same process is repeated. The algorithm terminates when all objects have been considered.

The pseudocode is given below:

```
X:= X^old; H := {};
while not_all_objects_considered() do
    k := random_pick_object();
    while unmarked_outstanding_replica_exists_for(k,X, X^new) do
        i := pick_server_for(k,X, X^new);
        while !has_space_for(i,k) do
            k' := pick_superflous_replica(i,X, X^new);
            H := H + D_ik;
            X[i][k'] := 0;
        end
        j := N_ik;
        H := H + T_jik;
        X[i][k] := 1;
    end
end
// delete remaining superfluous replicas, as in RTSP template
return (H);
```

The motivation of GOLCF 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. However, the random order in which objects are considered may lead to non-optimal overall results.

3.3 Schedule enhancement operators

Contrary to the previous CRPP and RTSP algorithms that produce a schedule from scratch, this category of heuristics have the form of operators that are applied on a given schedule to optimize it in terms of communication cost as per (7). They take as input replica placements X^{old} and X^{new} as well as a valid implementation schedule H, and iteratively produce a new valid schedule that is equivalent to H and incurs a lower cost.

3.3.1 Operator 1 (OP1): changing the order of transfer actions

The motivation of this heuristic is to *pull existing transfer actions* as early as possible towards the start of the schedule, provided that the newly installed replicas can then serve as more efficient sources for the subsequent transfers.

The concept is illustrated via a simple example, shown in Figure 3. Suppose there are three servers S_1, S_2 and S_3, connected as follows: $S_1 \leftrightarrow S_2$ with link cost 6, $S_2 \leftrightarrow S_3$ with link cost 1, and $S_1 \leftrightarrow S_3$ with link cost 7. Also, let the original schedule shown in Figure 3a be $\{..., T_{1a3}, ..., T_{1a2}, ...\}$. In this case one may (try to) reduce the cost of the schedule by performing the second transfer before the first one, i.e., change the schedule to $\{..., T_{1a2}, ..., T_{2a3}, ...\}$, shown in Figure 3b. The cost for the initial and modified schedule is $13s(O_a)$ and $7s(O_a)$, respectively. Notably, this change can be applied only if it has no side-effects or these can be addressed in a satisfactory way; this is discussed in more detail in the sequel.

(a) (b)
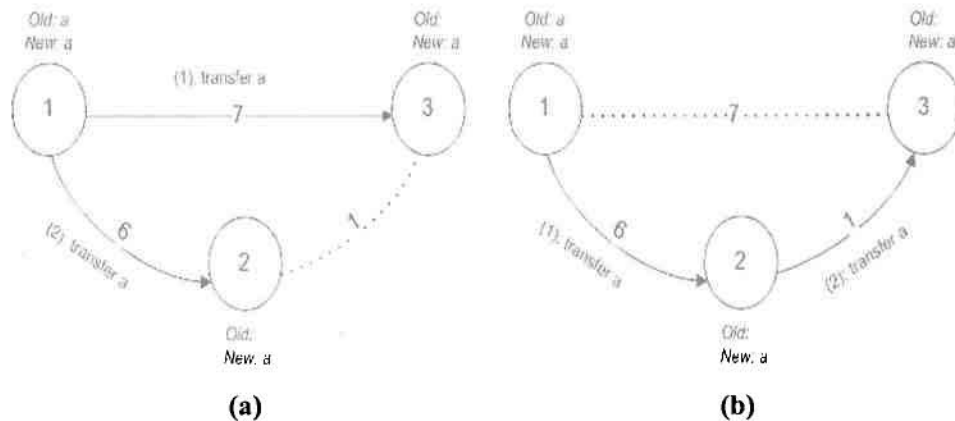
Figure 3. Reordering transfers via OP1

The algorithm operates as follows. The input schedule H is scanned from the left to the right until a transfer action $T_{j'ki'}$ is encountered. Scanning is continued until another transfer T_{jki} is found for the same object. It is then considered to move T_{jki} before $T_{j'ki'}$ to reduce the cost for all subsequent transfers for O_k found in the schedule.

Assume that H is of the form $\{..., T_{j'ki'}, G_1, T_{jki}, G_2\}$, where G_1 and G_2 are arbitrary sub-schedules. If the reordering is made, the resulting schedule will be $H' = \{..., T_{N_{ik}^{x^u}k'}, T_{iki'}, G_1', G_2'\}$, assuming that $T_{N_{ik}^{x^u}ki}$ is the uth action in the new schedule. The cost for implementing $T_{N_{ik}^{x^u}ki}$ is $s(O_k)l_{iN_{ik}^{x^u}}$. Also, for each transfer action $T_{j'ki''}$ in $\{T_{j'ki'}\} \cup G_1 \cup G_2$ the benefit of having a copy of O_k on S_i is 0 if $l_{ii''} > l_{j'i''}$ else equal to the cost difference between transferring O_k on $S_{i''}$ via the currently used source $S_{j'}$ and S_i: $s(O_k)(l_{j'i''} - l_{ii''})$. The algorithm considers modifying the schedule only if the total benefit outweighs the implementation cost, in which case the transfers in G_1 and G_2 that are affected from this change are updated to use S_i as their source (noted as G_1' and G_2'). However, additional *validation* checks and *repairs* are required to decide whether to consider such a modification. These are discussed in the following.

Let $D_{i,l1..lm}$ denote the sequence of deletions D_{il1}, D_{il2}, ..., D_{ilm}. In the general case, H is of the form $\{..., D_{i,kl..kn}, , T_{j'ki'}, G_1, D_{i\,l1..lm}, T_{jki}, G_2\}$ where $D_{i,kl..kn}$ and $D_{i,l1..lm}$ are replica deletions on $S_{i'}$ and S_i to enable transfers $T_{j'ki'}$ and T_{jki}, respectively. The suggested reordering results in schedule $H' = \{..., D_{i,l1..lm}, T_{N_{ik}^{x^u}ki}, D_{i\,k1..kn}, T_{iki'}, G_1', G_2'\}$ which is further evaluated / processed according to the following special cases:

(i) No crucial deletions: If no deletions $D_{i\,l1\,lm}$ precede T_{jki} and G_1 does not contain any deletions, schedule $H' = \{..., T_{N_{ik}^{v''}ki}, D'_{i\,k1..kn}, T_{iki'}, G_1', G_2'\}$ is valid and is adopted.

(ii) Void deletions: If G_1 is of the form $\{G_{1.1}, T_{j''li}, G_{1.2}\}$, where $l1 \le l < lm$, then the resulting schedule $H' = \{..., D_{i,l1..lm}, T_{N_{ii}^{...}ki}, D'_{i\,k1..kn}, T_{iki'}, G_{1,1}', T_{j''li}, G_{1,2}', G_2'\}$ is invalid. This is because $D_{i,l1..lm}$ contains a delete action for a replica O_l that not (yet) exist on S_i; the corresponding transfer action $T_{j''li}$ is (now) located further down the schedule. Consequently, the reordering is not adopted.

(iii) Outdated transfer sources. If G_1 is of the form $\{G_{1.1}, T_{ilj''}, G_{1.2}\}$, where $l1 \le l < lm$, then the resulting schedule $H' = \{..., D_{i,l1..lm}, T_{N_{ik}^{...}ki}, D'_{i\,k1..kn}, T_{iki'}, G_{1,1}', T_{ilj''}, G_{1,2}', G_2'\}$ is invalid. This is because action $T_{il''}$ assumes that S_i is a replicator of O_l, but a delete action for this replica, as part of $D_{i,l1..lm}$, is (now) located further up the schedule. Nevertheless, the validity of H' can be re-established by substituting $T_{ilj''}$ with $T_{N_{j''l}^{x''}lj''}$, yielding schedule $H' = \{..., D_{i,l1..lm}, T_{N_{ik}^{...}ki}, D'_{i,k1..kn}, T_{iki'}, G_{1,1}', T_{N_{j''l}^{v''}lj''}, G_{1,2}', G_2'\}$, assuming that $T_{ilj''}$ is the u'th action of the new schedule. Updating each such outdated transfer may introduce an additional penalty equal to $s(O_l)(l_{j''N_{j''l}^{x''}} - l_{j''i})$ which must be taken into account in order to decide whether this reordering actually leads to a cost cut.

(iv) Capacity constraint violation: If G_1 is of the form $\{G_{1.1}, D_{il}, G_{1.2}\}$, where $l \ne l1,..,lm$, then the resulting schedule $H' = \{..., D_{i\,l1..lm}, T_{N_{ik}^{...}ki}, D'_{i\,k1..kn}, T_{iki'}, G_{1,1}', D_{il}, G_{1,2}', G_2'\}$ is invalid, if the deletion of O_l on S_i was indeed required, in addition to deletions $D_{i,l1..lm}$,

to free space for hosting a replica of O_k via transfer action T_{jki} in schedule H. In this case, the validity of H' can be re-established by moving action D_{il} before $T_{N_{ik}^{v''}ki}$, yielding schedule $H' = \{..., D_{i,J1..lm}, D_{il}, T_{N_k^{x''}ki}. D_{i,k1..kn}, T_{iki'}, G'_{1.1}, G'_{1.2}, G'_2\}$. This also requires checking for outdated transfer sources in schedule G'_{11} as in (iii).

Each time the current schedule is modified, the algorithm starts scanning the new schedule from the beginning. The algorithm terminates when the end of the schedule is reached without having performed any modification.

3.3.2 Operator 2 (OP2): creating superfluous intermediate replicas

The motivation of this heuristic is to *introduce additional (superfluous) transfer actions* as early as possible towards the start of the schedule, provided that the newly installed replicas can then serve as more efficient sources for the subsequent transfers.

The concept is illustrated via a simple example, shown in Figure 4. Suppose there are four servers S_1, S_2, S_3 and S_4, connected as follows: $S_1 \leftrightarrow S_2$ with link cost 3, $S_1 \leftrightarrow S_3$ with link cost 4, $S_1 \leftrightarrow S_4$ with link cost 4, $S_2 \leftrightarrow S_3$ with link cost 1 and $S_2 \leftrightarrow S_4$ with link cost 1. Also, let the original schedule shown in Figure 4a be $\{..., T_{1a3},.., T_{1a4}, ...\}$. In this case one may (try to) reduce the cost of the schedule by introducing a new transfer before the one that already exists and a respective delete action afterwards, i.e., change the schedule to $\{..., T_{1a2}, T_{2a3}, ..., T_{2a4}, ..., D_{2a}, ...\}$, shown in Figure 4b. The cost for the initial and modified schedule is $8s(O_a)$ and $5s(O_a)$, respectively.
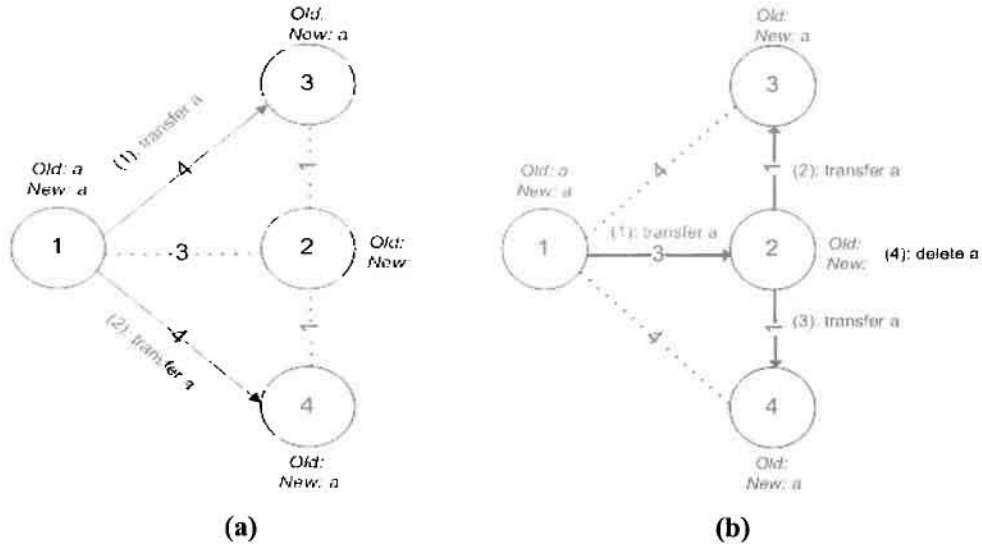
Figure 4. Introducing intermediate superfluous replicas via OP2

The algorithm operates in the spirit of OP1, as follows. The input schedule H is scanned from the left to the right until a transfer action $T_{j'ki'}$ is encountered. It is then considered to inject a new transfer action of O_k on some server S_i (immediately) before $T_{j'ki'}$ to reduce the cost for all subsequent transfers for O_k found in the schedule (including $T_{j'ki'}$).

Suppose H is of the form $\{..., D_{i'l1..lm}, T_{j'ki'}, G\}$. Performing this modification would result in schedule $H' = \{..., T_{N_{ik}^{x^u}ki}, D_{i',l1..lm}, T_{iki'}, G\}$, assuming that $T_{N_{ik}^{x^u}ki}$ is the uth action in the new schedule. The cost for implementing $T_{N_{ik}^{v^u}ki}$ is $s(O_k)l_{iN_{ik}^{v^u}}$. Also, for each transfer action $T_{j'ki''}$ in $\{T_{j'ki'}\} \cup G$ the benefit of having a copy of O_k on S_i is 0 if $l_{ii''} > l_{j'i''}$ else equal to the cost difference between transferring O_k on $S_{i''}$ via the currently used source $S_{j'}$ and S_i: $s(O_k)(l_{j'i''} - l_{ii''})$. The algorithm considers modifying the schedule

only if the total benefit outweighs the implementation cost, in which case the transfers in G that are affected from this change are updated to use S_i as their source (noted as G).

If S_i has no space to host a replica of O_k the algorithm considers deleting other superfluous local replicas $O_{k'}$ on S_i. The penalty for each such deletion is equal to $s(O_{k'})(l_{j''N_{j''k'}^u} - l_{ij''})$ for each transfer $T_{ik'j''}$ in G, assuming it is the uth action in the new schedule; to re-establish validity of H' the transfer $T_{ik'j''}$ must be replaced with $T_{N_{j''k'}^u k'j''}$. Superfluous replicas on S_i are considered in increasing penalty order, until enough free space is made for a copy of O_k or there are no more superfluous replicas left to consider.

A server S_i is considered as a candidate for hosting a superfluous replica of O_k only if the respective benefit outweighs the implementation cost and the aggregated deletion penalties, if any. If there are several candidate servers, the one that maximizes the cost reduction is chosen (if several candidates lead to the same result, one is picked at random). A corresponding transfer action is inserted in the schedule at the appropriate position, preceded by the required replica deletions, if any. If no candidate is found, the schedule remains unchanged. In any case, the algorithm proceeds with the next transfer action in the schedule. It terminates when the end of the schedule is reached, at which point any remaining superfluous replicas are deleted.

3.3.3 Combining OP1 and OP2

Operators OP1 and OP2 can be applied to the schedules produced by any of the previous CRPP or RTSP heuristics. It is also possible to apply them in a pipelined fashion. Notably, first applying OP2 and then OP1 is less effective than the reverse, because OP2 fills free server space with superfluous intermediate replicas which in turn makes it typically hard for OP1 to rearrange transfers without violating the capacity constraint (of course OP1

cannot interpret the semantics of the additional transfers introduced by OP2 to create the superfluous intermediate replicas). For this reason the combination OP2+OP1 is not shown nor discussed in the following evaluation section; in our experiments it has consistently led to worse results than the combination OP1+OP2.

4 Evaluation

This section discusses experimental results that were produced via simulation. The first series of experiments compares the RTSP heuristics of Section 3.2 combined with the schedule enhancement operators of Section 3.3. The second series of experiments investigates the improvement that can be achieved by applying the operators of Section 3.3 to the schedules produced by the CRPP heuristics of Section 3.1.

4.1 Setup

The server network was generated using BRITE [15], for 50 nodes each having a connectivity of 1. 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. Point-to-point communication costs were set equal to aggregated link cost along the shortest (less costly) paths. A set of 1000 objects was used, with sizes uniformly distributed between 1000 and 5000. The primary replicas were randomly assigned to the server nodes. The resulting topology is shown in the Appendix.
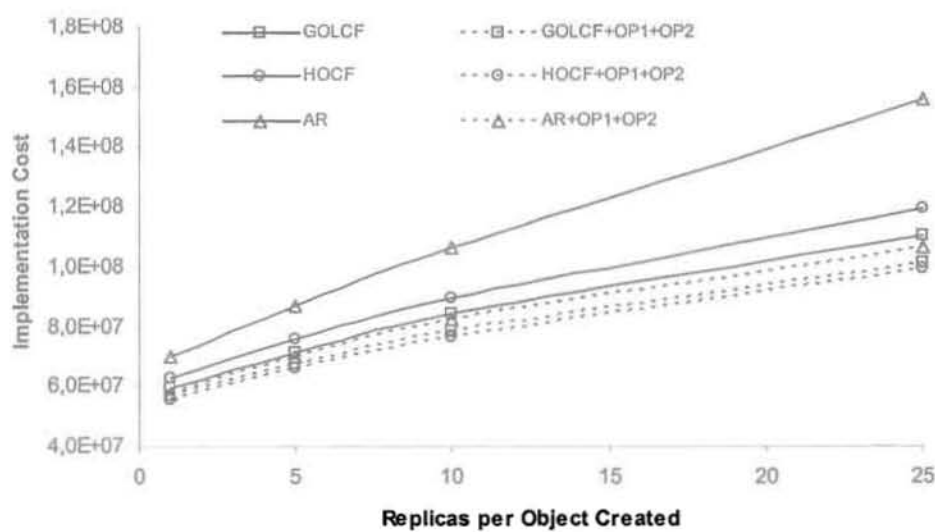
4.2 Comparison of RTSP heuristics

First, we compare the implementation cost of the schedules produced by RTSP heuristics for the case where X^{old} consists only of the primary replicas. We let X^{new} vary in terms of the number of replicas that need to be created for each object. The servers where the additional replicas must be placed in X^{new} are chosen randomly. The storage capacity of each server is set equal to the sum of the replicas it must host in X^{new}.

Figures 5 and 6 plot the cost of the schedules produced by AR, HOCF, GOLCF (solid lines) and their combinations with OP1, OP2 and OP1+OP2 (dashed lines). It is important to note that in this case no replica deletions take place. For this reason LPSF and LVSF operate in a "degraded" manner that is equivalent to AR; they are omitted to avoid cluttering the plots. For all algorithms, the cost (naturally) grows as the number of replicas in X^{new} increases, because more transfers are required to achieve this. GOLCF achieves the best performance, closely followed by HOCF which employs a more "defensive" replication policy. AR clearly produces the worse results, as expected, given its random choice design (it is used as a reference for the performance of the other algorithms).

Operators OP1 and OP2 enhance the schedules of all algorithms, the best result being produced when applying OP1+OP2. The effects of OP1 and OP2 differ depending on which schedule they are applied to. Most notably, OP1 does not change the schedule of GOLCF, and any improvement is due to OP2. This is because GOLCF optimally creates the needed replicas for each object when there are no deletions, leaving no room for OP1 to optimize object transfers. OP2 can further reduce the cost by injecting superfluous replicas on servers that have available space, which becomes more notable with increasing server capacity. Particularly noteworthy is the drastic improvement of AR+OP1+OP2 over AR, which performs close to the rest of the algorithms, even when combined with OP1+OP2.
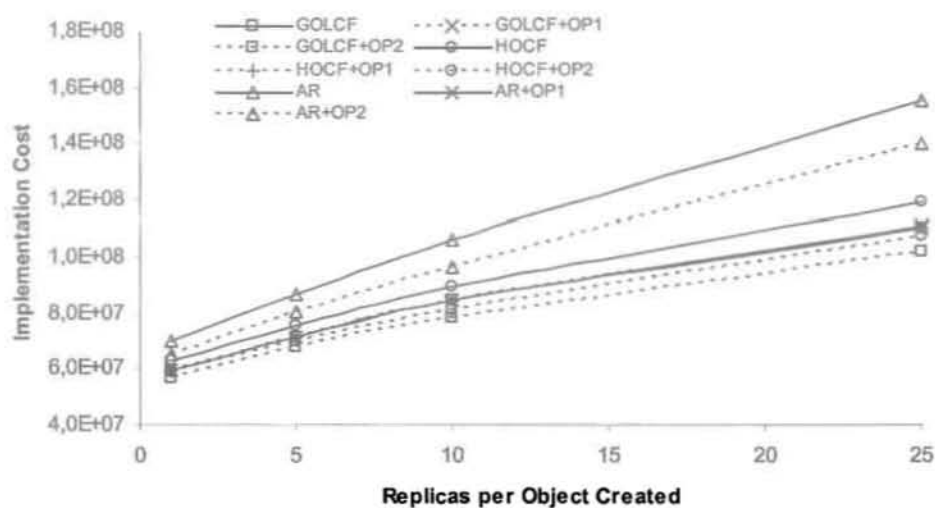
Figure 5. Schedule cost of RTSP heuristics and their combination with OP1 or OP2, while increasing replication requirements starting from primary copies only
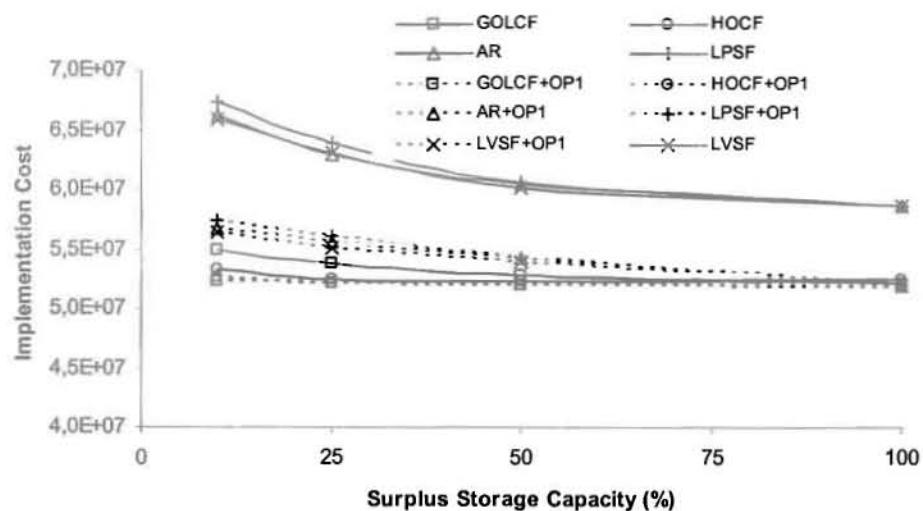

Figure 6. Schedule cost of RTSP heuristics and their combination with OP1+OP2, while increasing replication requirements starting from primary copies only

In a second experiment, we compare the RTSP algorithms for the case where both replica transfers and deletions must be performed to arrive at the desired replica placement. For this purpose X^{old} is defined so that each server holds 50 randomly picked object copies, and X^{new} is constructed by quasi-randomly flipping the bits of X^{old} so that each server stores 50 replicas with a relative overlap of 75% with respect to X^{old}. In other words, 12.5% of the replicas (on the average about 6 replicas per server) have to be deleted from their old hosts and created on other servers. The performance of the algorithms is measured as a function of the surplus in storage capacity with respect to the minimum space required by each server in X^{old} and X^{new}.

Figures 7 and 8 plot the costs for AR, HOCF, GOLCF, LPSF and LVSF, and their combination with operator OP1 and OP2, respectively. Figure 9 plots AR, HOCF, GOLF, LPSF and LVSF, and their combination with OP1+OP2. For all algorithms the cost drops as storage surplus increases. This is because the required replicas can be created without having to delete as many superfluous replicas, which in turn can serve as better sources for subsequent transfers, reducing the overall implementation cost. Costs stabilize once storage surplus reaches a certain level that is sufficient for optimizing the required transfers. This time HOCF slightly outperforms GOLCF, due to its more refined policy for deleting superfluous replicas. LVSF and LPSF produce rather poor schedules; the latter is even worse than AR. This is because they attempt to optimize the deletion order without making any effort to optimize replica creations (the first transfer that fits is chosen randomly). LVSF performs better than LPSF, indicating that it employs a better criterion for selecting victim superfluous replicas (the same as HOCF).

Again, all schedules can be improved by applying OP1+OP2, in particular those of AR, LPSF and LVSF; and less for HOCF and GOLCF. OP1 and OP2 play a complementary role, with the impact of OP2 becoming more notable relative to that of OP1 for increasing surplus capacity, since there is more space that can be used to create intermediate replicas.

Figure 7. Schedule cost for RTSP heuristics and their combination with OP1, while increasing storage surplus for replica placements with 75% overlap
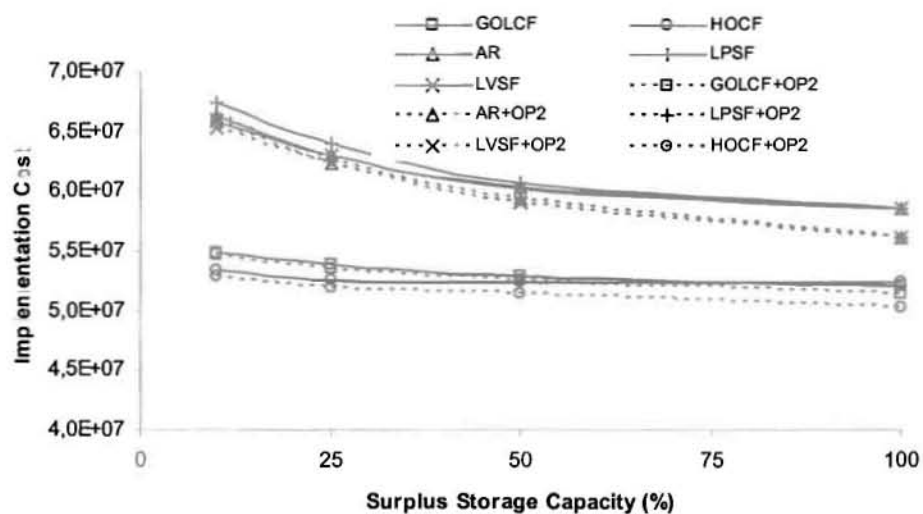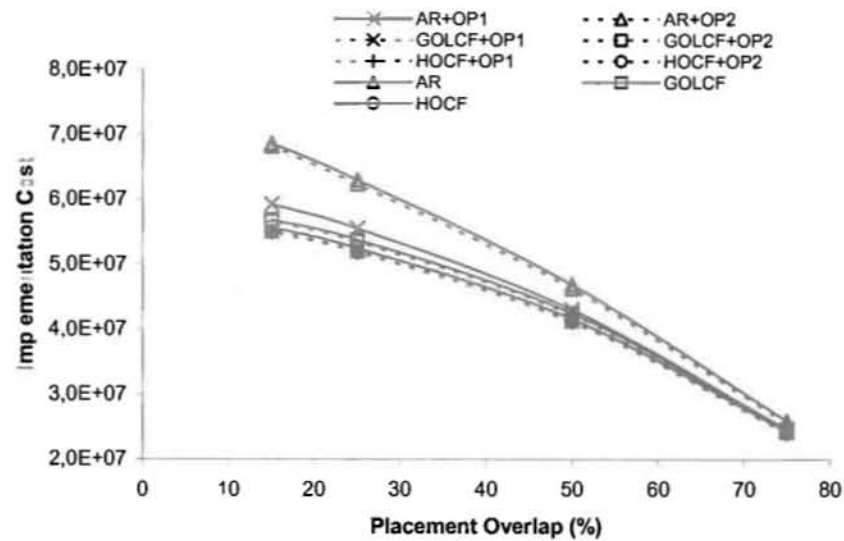

Figure 8. Schedule cost for RTSP heuristics and their combination with OP2, while increasing storage surplus for replica placements with 75% overlap
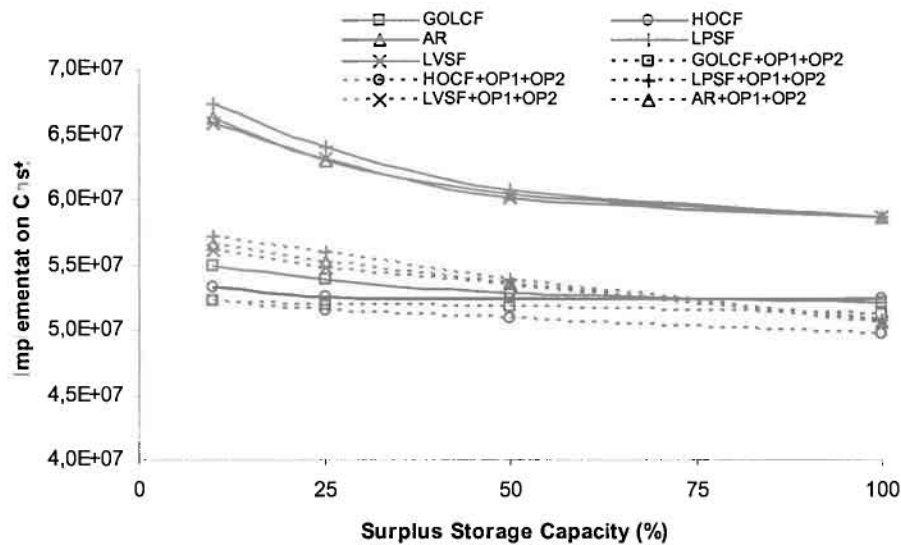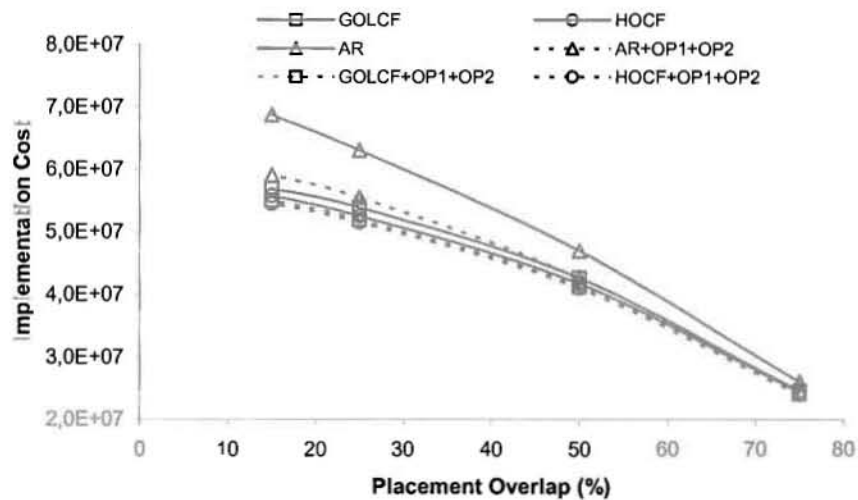
Figure 9. Schedule cost for RTSP heuristics and their combination with OP1+OP2, while increasing storage surplus for replica placements with 75% overlap

In a third experiment, we investigate the performance of our heuristics for the case where the storage capacity of each server amounts to 25% of the sum of all objects sizes, while varying the overlap between X^{old} and X^{new}. Given their poor performance in the previous experiment, LPSF and LVSF are not included. AR is still used as a reference.

Figure 10 plots the costs of the schedules produced by AR, HOCF and GOLCF also in combination with OP1 or OP2, and Figure 11 shows the respective costs also in combination with OP1+OP2. As expected, for all algorithms the cost drops sharply as overlap increases and the number of new replicas to be created decreases. HOCF performs marginally better than GOLCF, because it places more emphasis in the order in which superfluous replicas are deleted. As in the previous experiment, the enhancement achieved via OP1 and OP2 is limited, expect for AR.

Figure 10. Schedule cost for RTSP heuristics and their combination with OP1 or OP2, while increasing overlap between X^{old} and X^{new} at 25% surplus capacity

Figure 11. Schedule cost for RTSP heuristics and their combination with OP1+OP2, while increasing overlap between X^{old} and X^{new} at 25% surplus capacity

In summary, among the RTSP heuristics, HOCF and GOLCF achieve the best performance. HOCF has a slight edge over GOLCF when there are replica deletions. The schedules produced by all heuristics can be enhanced when combined with OP1+OP2.

4.3 Comparison of RTSP vs CRPP heuristics

Having two clear (and mostly equivalent) winners between the RTSP heuristics, we choose HOCF and GOLCF for a comparison with CRPP heuristics GG and GOR. The operators OP1 and OP2 are also evaluated in conjunction with these algorithms.

Recall that CRPP heuristics take as input a client traffic pattern $r_{ik}, \forall i, k$ to drive the computation of X^{new} as well as the schedule H for implementing it based on X^{old}. In our experiments we model client traffic as follows: (i) every request is for reading the entire object; (ii) requests are uniformly distributed on the servers; (iii) object popularity, i.e., client requests per object, follows a Zipf distribution with parameter 0.8 (this is indicative of several large-scale information systems) and a maximum popularity value of 1000 which corresponds to an average of 5 read cost units for each object per server.

RTSP algorithms take as input X^{old} and X^{new}, the latter being this time produced via a CRPP heuristic, and produce a different schedule. In addition, operators OP1 and OP2 are applied to all schedules produced. The comparison is made in terms of the cost reduction achieved relative to the cost of the original schedules of GG and GOR, respectively.

The first experiment investigates the case where X^{old} consists only of the primary replicas and the storage capacity of each server ranges from 3.5% to 75% of the sum of all object sizes. Figures 12 and 13 depict the results with respect to GG, and Figures 14 and 15 depict the results for GOR.

In terms of RTSP vs CRPP heuristics, GOLCF produces more efficient schedules compared to the ones generated by both GG and GOR. Note that the performance of GOLCF shows opposite trends relative to that of GG and GOR, i.e., increasing and decreasing gains, respectively, as server storage capacity grows. The impressive cost difference for GOR when server capacity is limited (less than 25%) is because GOR fully replicates one object at a time, so that replicas created in previous iterations will most likely be deleted in subsequent ones; resulting in wasted transfers. HOCF lags behind GOLCF, due to its more "defensive" replication policy which cannot bear fruits as long as few or no replicas need to be deleted to produce the new replica placement.

Combined with OP1+OP2, GOLCF continues to produce good results but with a smaller difference compared to GG, while being slightly outperformed by HOLCF+OP1+OP2. GOR+OP1+OP2 significantly lags behind GOLCF+OP1+OP2 for relatively small server capacities. This is because OP1 and OP2 cannot possibly "repair" the wasted transfers problem since they simply optimize existing transfers. Notably, OP1 significantly enhances the schedules of GG whereas OP2 has a small impact. This can be explained as follows. Since GG creates new replicas in order to reduce the client access cost, the first replica of for a given object will most likely be installed at a server that is far away from the existing sources (e.g., primary replica). Subsequent replicas for the same object, if deemed necessary, are likely to be placed on servers that lie in between. Clearly, instead of creating the remote replica first, a better schedule is to first create replicas on servers that are near the existing sources, and use them as more proximate sources for the next transfers on servers further away. This is what OP1 attempts to do, and is more likely to succeed as server storage capacity increases (storage constraints are being relaxed). OP2 has little impact because GG will aggressively fill servers with replicas, leaving little space for introducing superfluous replicas throughout the schedule. The same observations hold for GOR, which uses the same replica placement criteria as GG.
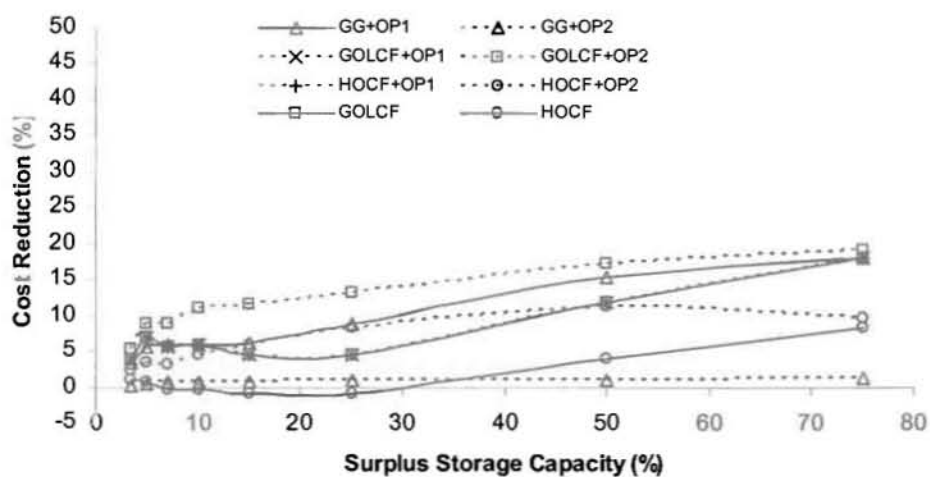
Figure 12. Schedule cost reduction wrt GG for RTSP heuristics and OP1 or OP2, when increasing storage capacity while starting from primary replicas only
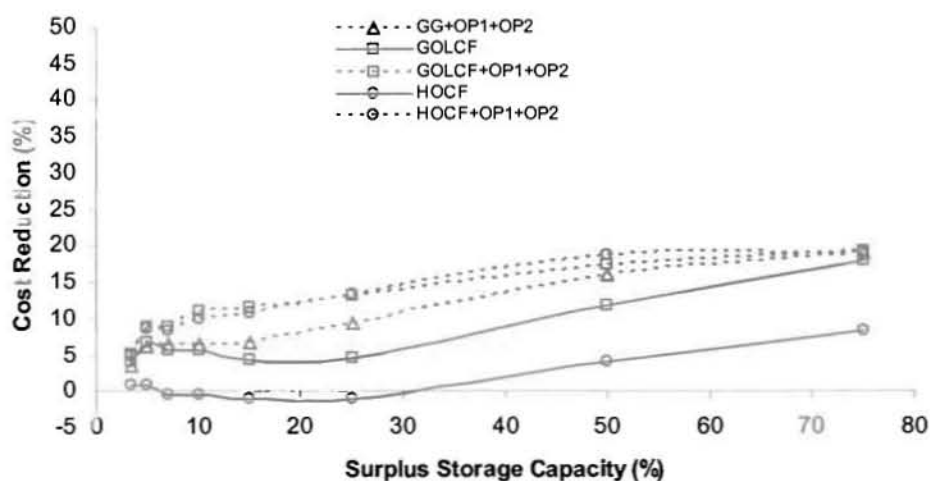
Figure 13. Schedule cost reduction wrt GG for RTSP heuristics and OP1+OP2, when increasing storage capacity while starting from primary replicas only
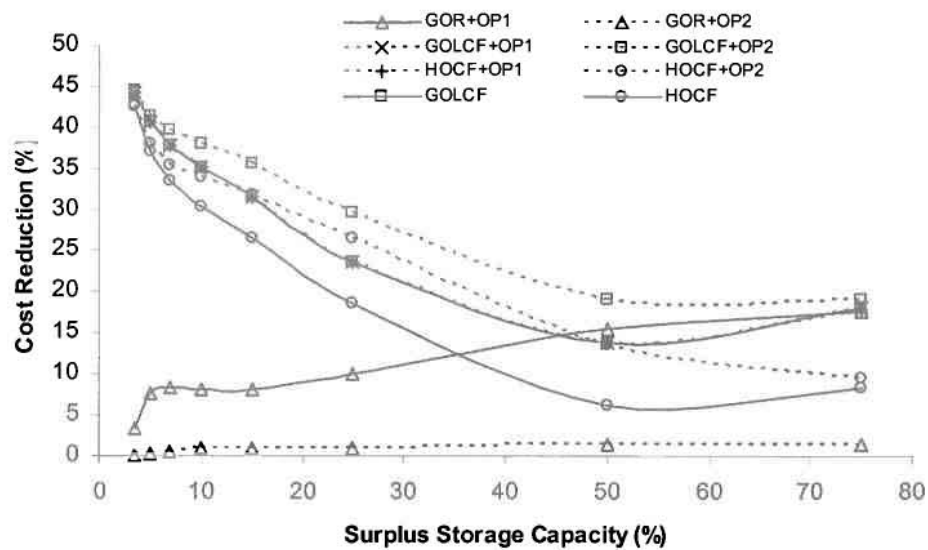
Figure 14. Schedule cost reduction wrt GOR for RTSP heuristics and OP1 or OP2, when increasing storage capacity while starting from primary replicas only
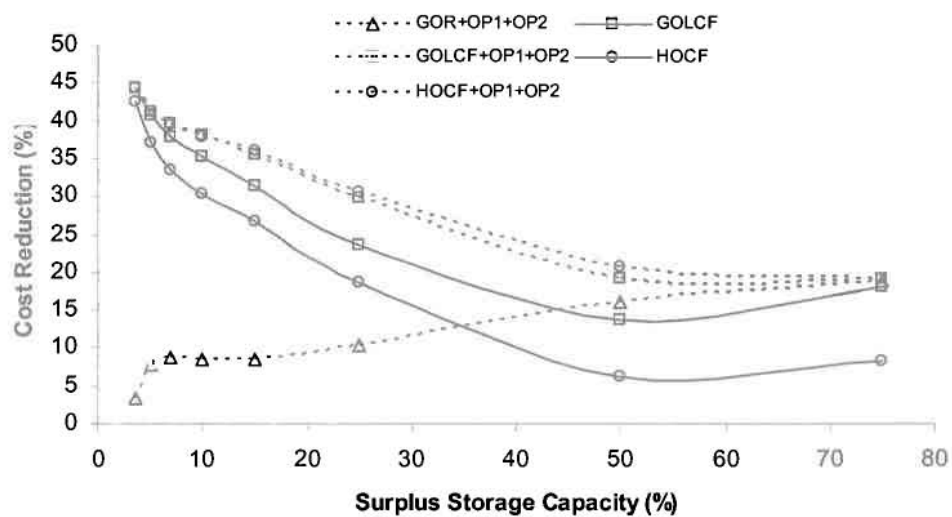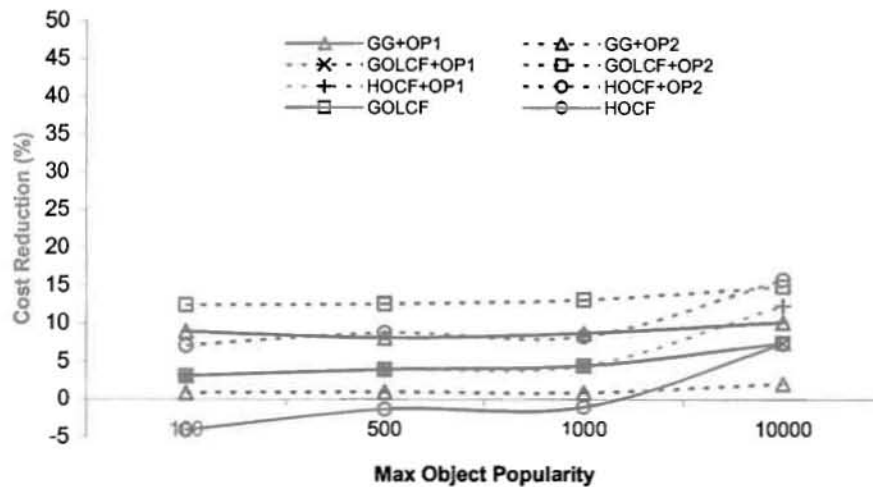

Figure 15. Schedule cost reduction wrt GOR for RTSP heuristics and OP1+OP2, when increasing storage capacity while starting from primary replicas only

In the next experiment, the storage capacity of each server is kept fixed at 25% of the sum of all object sizes, and the initial replica placement X^{old} features only the primary replicas. The algorithms are compared for different maximum popularity values of an object, taking values 100, 500, 1000 and 10000. Larger maximum popularity values result in a higher number of total client requests as well as in bigger differences in the number of replicas that will be created for each object. Notably, the total number of client requests also affects the relative weight of the read cost function compared to the implementation cost, which affects the results of RTSP algorithms. The cost reduction of the schedules produced using the various heuristics are shown in Figures 16 and 17 for GG, and Figures 18 and 19 for GOR, respectively.

It can be seen that GOLCF performs better relative to GG and GOR, increasingly so for larger object popularity values. This is because as the read volume increases more replicas will be created, hence the margin for improvement increases for GOLCF. The performance gap grows slowly for GG and significantly faster for GOR. This is due to the fact that as object popularity increases, the aggressive "per object" replication strategy of GOR becomes increasingly non-optimal because the probability of the most popular objects not being considered first (GOR picks objects randomly) increases. This in turn considerably increases the number of "wasted" transfers, as discussed previously. HOCF exhibits a similar behavior, but for small maximum object popularity values produces schedules that are more costly than the ones produced by GG and GOR.

Again, GOLCF+OP1+OP2 steadily outperforms GG+OP1+OP2 and GOR+OP1+OP2. The performance difference remains constant between GOLCF and GG but increases between GOLCF and GOR. This is because, as already discussed, neither OP1 nor OP2 can "repair" the wasted transfers introduced by GOR. HOCF+OP1+OP2 performs slightly better than GOLCF+OP1+OP2, with the tendency to outperform it as the maximum popularity increases.

Figure 16. Schedule cost reduction wrt GG for RTSP heuristics and OP1 or OP2, when increasing popularity at 25% server capacity and starting from primary replicas
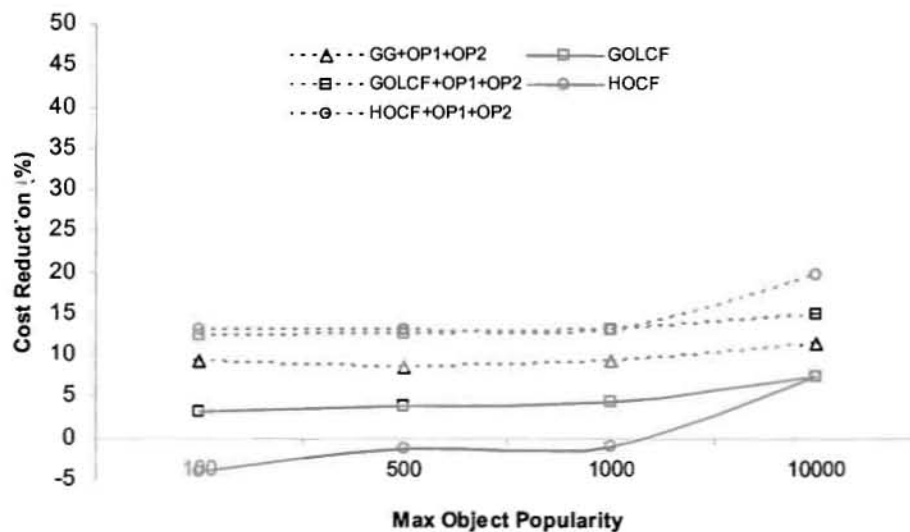
Figure 17. Schedule cost reduction wrt GG for RTSP heuristics and OP1 or OP2, when increasing popularity at 25% server capacity and starting from primary replicas
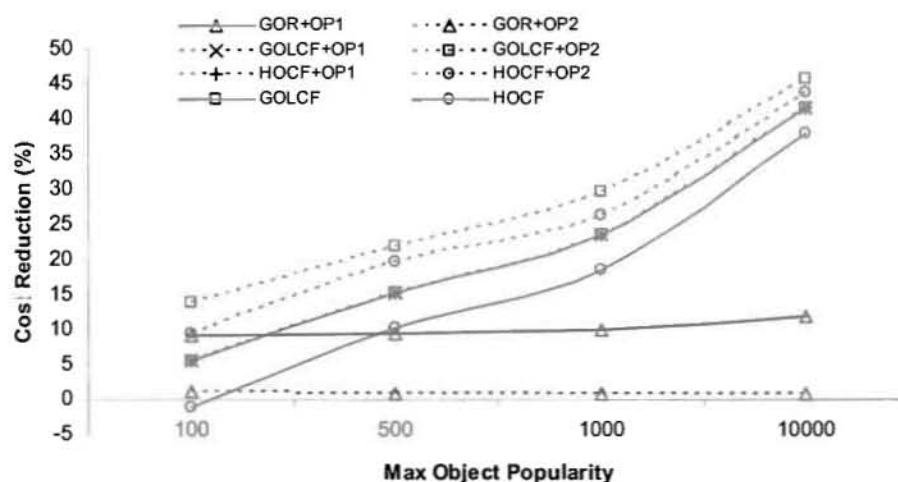
Figure 18. Schedule cost reduction wrt GOR for RTSP heuristics and OP1 or OP2, when increasing popularity at 25% server capacity and starting from primary replicas

Figure 19. Schedule cost reduction wrt GOR for RTSP heuristics and OP1 or OP2, when increasing popularity at 25% server capacity and starting from primary replicas
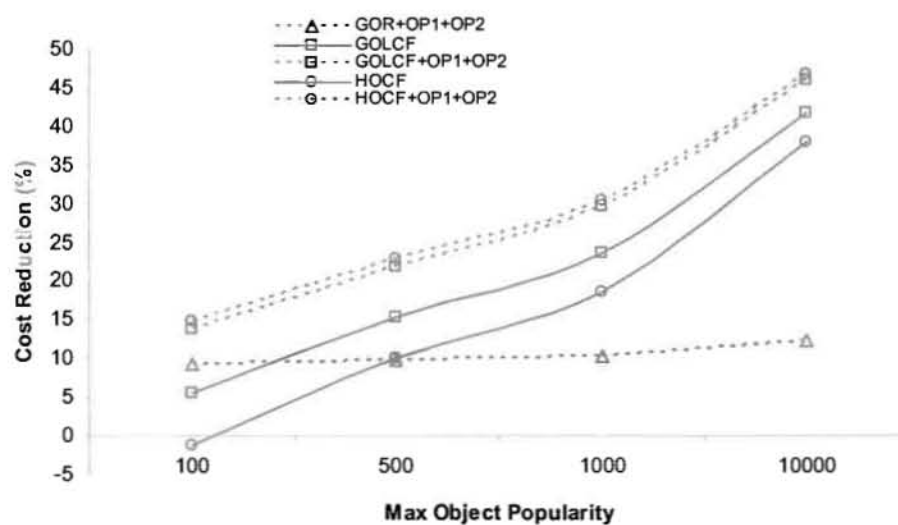
The last experiment measures the cost reduction of the schedules produced by the heuristics as a function of the available copies for each object in the initial replica placement X^{old}. Notably, this indirectly controls the number of replicas that will be created and deleted in the new replica placement. More specifically, the more replicas are initially available, the greater the number of replicas corresponding to less popular objects which will be deleted, and (at the same time) the smaller the number of new replicas that will be created to lower the read cost for the most popular objects. The storage capacity of each server is 25% of the sum of all object sizes, and the maximum object popularity is 1000. The results are shown in Figures 20 and 21 for GG, and Figures 22 and 23 for GOR, respectively.

For replica placements derived from GG, GOLCF generates marginally better schedules than GG while HOCF does slightly worse than GG. This small difference confirms that GG produces relatively good schedules when server storage capacity is relatively limited. Notably, the tendency is for HOCF to reach and outperform GG for large numbers of initial sources, indicating that its replica deletion criterion works well. The reverse is true for GOLCF, indicating that its "per object" replication policy becomes less efficient when there are already several initial replicas in the system. Both GOLCF and HOCF significantly outperform GOR in all cases, with a tendency in favor of HOCF as the number of initial replicas increases.

Finally, the benefit of applying OP1 and OP2 is notable for the case of GOR and remains constant. On the contrary, this has a small impact for the case of GG, which diminishes for large numbers of initial replicas. This is due to the fact that an increasing number of available replicas are kept in place and that new replicas can be created more efficiently using existing ones as sources. Hence, OP1 and OP2 are given little room for further improvement. Nevertheless, even in the case where each object has initially 10 replicas, i.e., is replicated on $1/5^{th}$ of the servers, OP1 and OP2 still result in better schedules.
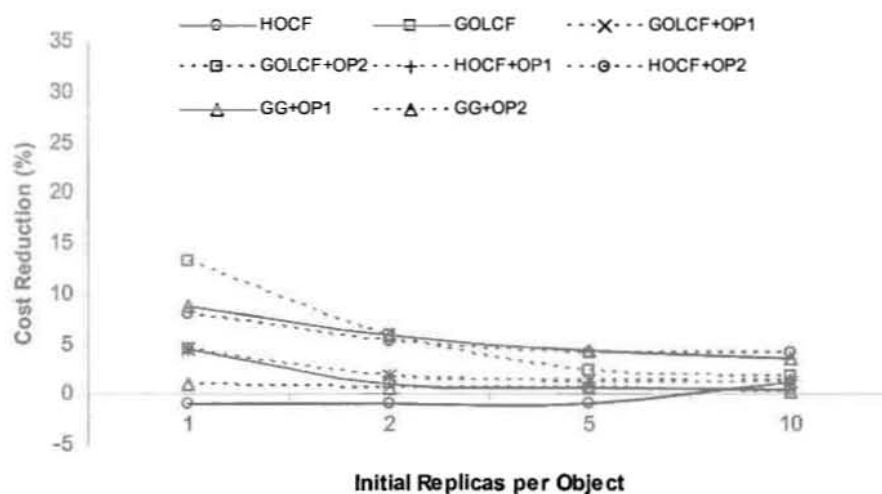
Figure 20. Schedule cost reduction wrt GG for RTSP heuristics and OP1 or OP2, when increasing initially available replicas at 25% server capacity and max. pop. 1000
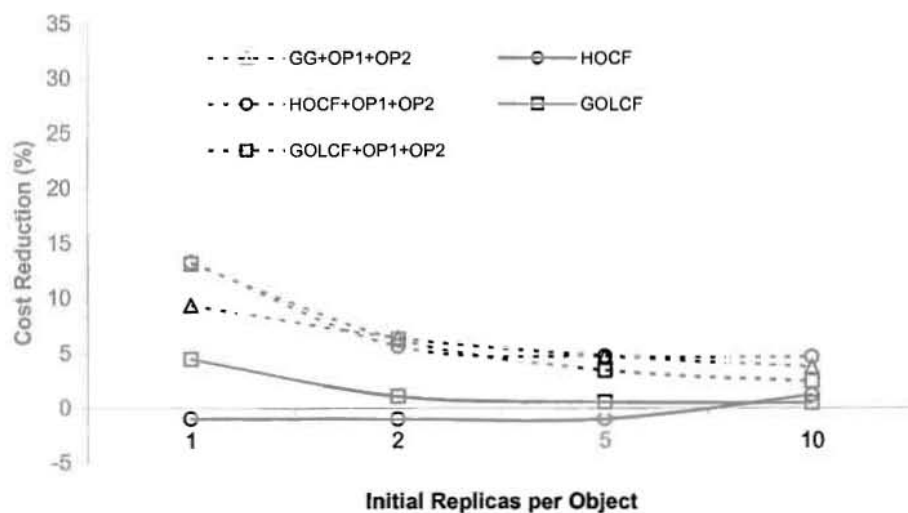
Figure 21. Schedule cost reduction wrt GG for RTSP heuristics and OP1+OP2, when increasing initially available replicas at 25% server capacity and max. pop. 1000

Figure 22. Schedule cost reduction wrt GOR for RTSP heuristics and OP1 or OP2, when increasing initially available replicas at 25% server capacity and max. pop. 1000
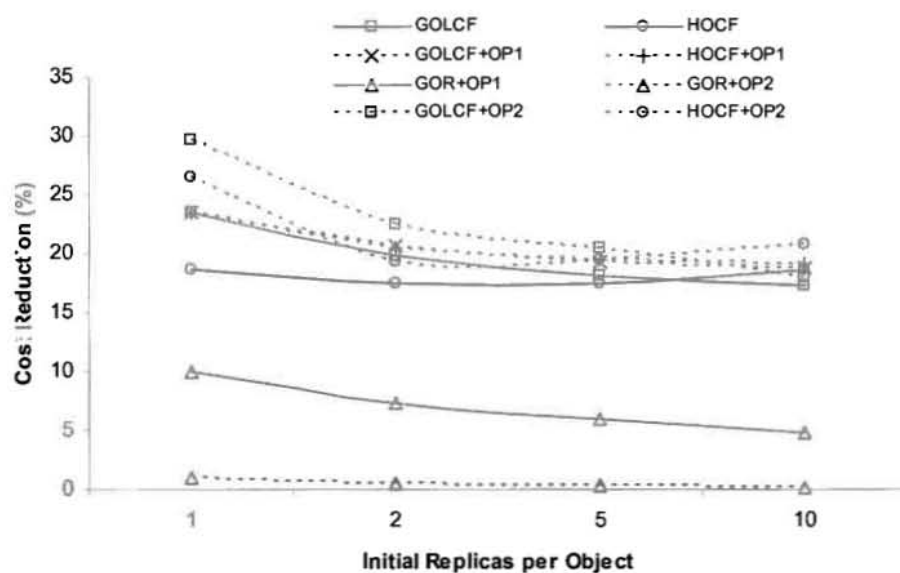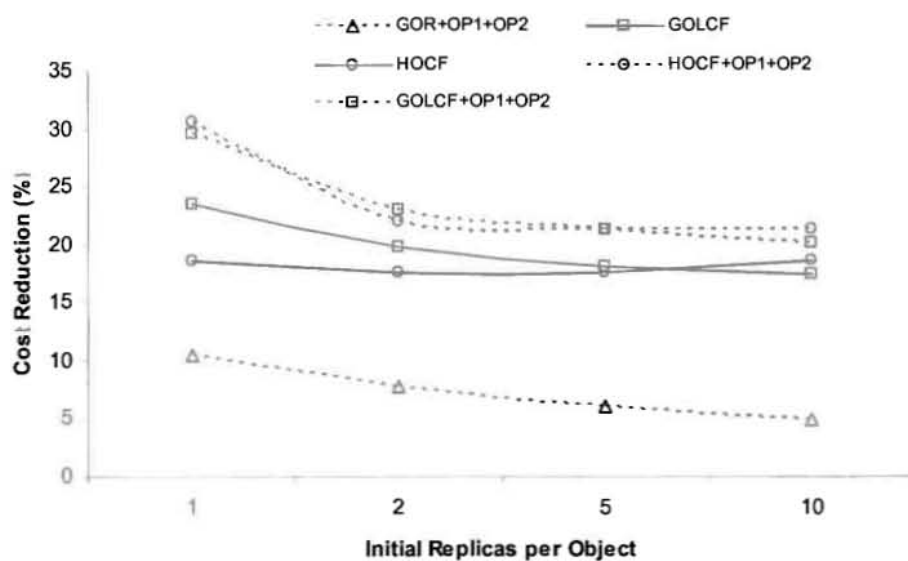

Figure 23. Schedule cost reduction wrt GOR for RTSP heuristics and OP1+OP2, when increasing initially available replicas at 25% server capacity and max. pop. 1000

4.4 Summary

From the RTSP heuristics presented, HOCF and GOLCF produce the most efficient schedules for migrating between two replica placements. In most cases, GOLCF also produces a better schedule for implementing replica placements computed via RTSP heuristics GG and GOR.

The application of operators OP1 and OP2 on any schedule, whether this was produced via RTSP or CRPP algorithms, leads to notable improvements. In fact, schedules of AR can be enhanced to a considerable degree, almost to the level of schedules that are produced by far more sophisticated algorithms. In the majority of our experiments, OP1 turned out to be more effective than OP2.

Overall, GOLCF+OP1+OP2 produced the best results with satisfactory consistency.

5   Related work

The Replica Placement Problem has been researched quite extensively, and a variety of problems definitions have been proposed in this context. In [7] client-replica distance is considered as the optimization target, whereas the primary goal of [23] is load balancing. Read access cost is the focus in [8, 12], while [13] considers client traffic that includes both read and update requests. Other issues taken into account in conjunction with RPP formulations are server storage capacity [8, 13], processing capacity [18] and bandwidth [7] to name a few. In this dissertation we have adopted a model similar to [8]. Although our RTSP definition can be extended to include additional parameters, in this work we have focused in what we believe to be the essence of the problem.

The literature on scheduling related problem is also very rich. It spans many disciplines such as parallel computing in the context of scheduling tasks to multi-processors [5, 11] or operations research in the context of vehicle sequencing/routing [4, 16]. Solutions to these problems include deterministic algorithms, e.g., branch and bound [11], as well as randomized algorithms, e.g., genetic algorithms [16]. Unfortunately, the existing scheduling heuristics cannot be applied to the Replica Transfer Scheduling Problem without extensive modifications. This is because RTSP, though similar in some aspects, differs significantly from the problems that have been already investigated. Consider for example the classic multi-processor scheduling problem. Heuristics take as input a task

graph and output task mappings to processors as to minimize the total execution time. This is in contrast to RTSP where there is no predetermined graph since (among other things) intermediate sources might be created and deleted. Also, our formulation of RTSP focuses on minimizing the communication cost dues to object transfers without taking into account the notion of time.

We are unaware of any previous research efforts that try to minimize the cost for migrating between two different replica placements. The closest work can be found in [14], which investigates CRPP heuristics (including GG) in comparison to conventional RPP heuristics. On the contrary, in this thesis we have addressed RTSP as a separate and self-standing optimization problem.

6   Conclusions and outlook

In this thesis we have formulated the Replica Transfer Scheduling Problem (RTSP) and have given several heuristics that can be used to produce solutions. We have evaluated their performance via simulations, also with reference to indicative greedy Continuous Replica Placement (CRPP) algorithms. Our results show that the proposed RTSP algorithms can significantly improve the schedules of CRPP algorithms. To the best of our knowledge, this is the first time RTSP is investigated as separate problem.

Although our thesis has provided first insight into RTSP, there is still more work that can be done to advance our knowledge in this area. The presented heuristics can be studied in more detail, through a wider range of scenarios to get a better feeling of their behavior. For example, OP2 could be investigated on different topologies where the proactive creation of superfluous replicas is expected to have more impact. In addition, new RTSP heuristics can be proposed, by varying the criteria used to pick outstanding and superfluous replicas. It may also be worthwhile researching variants of the current problem, for instance to study RTSP for the case where the primary replicas can change location but still requiring that in the new placement there is at least one replica for each object (this problem may not even have a solution due to the server storage constraint). Last but not least, it would be interesting to adjust our RTSP formulation in order to take into account the actual time it takes for a schedule to complete, with the goal to minimize

this value. In a sense, this can be thought of as the "dual" of the problem we have studied in this thesis. While this is strongly related to known problems in the area of multi-processor scheduling, we believe that a different approach may be needed to tackle this problem.

References

[1]   P. Apers, "Data allocation in distributed database systems," *ACM Transactions on Database Systems*, vol. 13, no. 3, pp. 263–304, 1988.

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

[3]   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.

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

[5]   D. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling - a status report," in Proc. *11th Int. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2005)*, Cambridge, MA, USA, 2005.

[6]   I. Foster, "The Grid: Blueprint for a New Computing Infrastructure," Morgan Kaufmann, 2nd Ed., 2004.

[7]   S. Jamin, C. Jin, A. R. Kurc, D. Raz, and Y. Shavitt, "Constrained mirror placement on the Internet," in Proc. *IEEE INFOCOM*, April 2001, pp. 31–40.

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

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

[10] M. Karlsson, C. Karamanolis, and M. Mahalingam, "A framework for evaluating replica placement algorithms," HP Labs, Technical Report HPL-2002-219, July 2002.

[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 and I. Ahmad, "Static and adaptive distributed data replication using genetic algorithms," in *J. Parallel and Distr. Comp. (JPDC)*, Vol. 64(11), pp. 1270-1285, 2004.

[14] 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.

[15] 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

[16] J. Potvin and S. Bengio, "The vehicle routing problem with time windows part II: genetic search," in *Journal on Conputing*, Vol. 8(2), pp. 165-172, 1996.

[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, I. Rabinovich, R. Rajaraman, and A. Aggarwal, "A dynamic object replication and migration protocol for an Internet hosting service," in Proc. *ICDCS'99*, May 1999, pp. 101–113.

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

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

[21] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam, "Taming aggressive

replication in the Pangaea wide-area file system," in Proc. *Fifth Symposium on Operating System Design and Implementation (OSDI'02)*, pp. 15-30, 2002.

[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.

[24] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal and F. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, Vol. 29, No.3, Mar. 1986, pp. 184-201.

[25] S. Williams, M. Abrams, C. Standridge, G. Abdulla and E. Fox, "Removal Policies in Network Caches for World-Wide Web Documents," *in Proc. of the SIGCOMM'96 Conf.*, 1996.

[26] G. Banga, F. Douglis and M. Rabinovich, "Optimistic deltas for WWW latency reduction," *in Proc. of the USENIX Technical Conf.*, pp. 289-303, Anaheim, 1997.

[27] J. Challenger, A. Iyengar and P. Dantzig, "A Scalable System for Consistently Caching Dynamic Web Data," *in Proc. of the IEEE INFOCOM'99 Conf.*, March 1999.

[28] M. Baentsch, L. Baum, G. Molter, S. Rothkugel and P. Sturm, "Enhancing the web infrastructure - from caching to replication," *IEEE Internet Computing*, pp. 18-27, Mar-Apr. 1997.

[29] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar, "A novel server selection technique for improving the response time of a replicated service," *in Proc. of the IEEE INFOCOM '98 Conf.*, March 1998.

[30] CISCO, "Distributed Director," White paper at: http://www.cisco.com/warp/public/cc/cisco/mkt/scale/distr/tech/dd_wp.htm.

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

[32] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. Gryniewicz, and Y. Jin, "An Architecture for a Global Internet Host Distance Estimation Service," *in Proc. of the IEEE INFOCOM '99 Conf.*, March 1999.

[33] V. Cardellini, M. Colajanni and P. Yu, "Redirection Algorithms for Load Sharing in Distributed Web-server Systems," *in Proc. of the 19th IEEE Int. Conf. on Distributed Computing Systems ICDCS'99*, Austin, Texas, May 1999.

[34] A. Heddaya and S. Mirdad, "WebWave: Globally Load Balanced Fully Distributed Caching of Hot Published Documents," *in Proc. of the 17th Int. Conf. On Distributed Computing Systems (ICDCS'97)*, May 1997.

[35] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson and D. Culler, "Using smart clients to build scalable services," *in Proc. of the 1997 USENIX Annual Technical Conference,* Jan. 6-10, 1997, Anaheim, CA, pp. 105-117, USENIX, January 1997.

[36] B. Li, M. Golin, G. Italiano and X. Deng, "On the optimal placement of web proxies in the Internet," *in Proc. of the IEEE INFOCOM'00 Conf.*, March 2000.

[37] T.D.C. Little and D. Venkatesh, "Popularity-Based Assignment of Movies to Storage Devices in a Video-on-Demand System," *ACM/Springer Multimedia Systems*, 1994.

[38] S. Bhattacharjee, M. Ammar, E. Zegura, V. Shah and Z. Fei, "Application-layer anycasting," *in Proc. of the INFOCOM'97 Conf.*, 1997.

[39] A. Iyengar and J. Challenger, "Improving Web Server Performance by Caching Dynamic Data," *in Proc. of the USENIX Symp. on Internet Technologies and Systems*, pp. 49-60. Monterey, CA, Dec., 1997.

[40] M. Crovella and R. Carter, "Dynamic Server Selection in the Internet," *in Proc. of the 3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS'95)*, June 1995.

[41] S. Jamin, C. Jin, T. Kurc, D. Raz and Y. Shavitt, "Constrained Mirror Placement on the Internet," *in Proc. of the IEEE INFOCOM 2001 Conf.*, Alaska, USA.

[42] R. Casey, "Allocation of Copies of a File in an Information Network," *in Proc. of the Spring Joint Computer Conf.*, IFIPS, 1972, pp. 617-625.

[43] M. Beck and T. Moore, "The Internet-2 distributed storage infrastructure project: An architecture for internet content channels," *in Proc. of the 3rd Int. WWW Caching Workshop*, Manchester, UK, June 1998.

[44] L. Qiu, V. Padmanabhan and G. Voelker, "On the Placement of Web Server Replicas," *in Proc. of the IEEE INFOCOM'01 Conf.*, Anchorage, AK, USA April 2001.

[45] V. Padmanabhan and J. Mogul, "Using Predictive Prefetching to Improve World Wide Web Latency," *in Proc. of the ACM SIGCOMM'96 Conf.*, pp. 22-36, July 1996.

[46] C. Bisdikian and B. Patel, "Cost-Based Program Allocation For Distributed Multimedia-on-demand Systems," *IEEE Multimedia*, 3(3), pp. 62-72, 1996.

[47] P. Barford, A. Bestavros, A. Bradley and M. Crovella, "Changes in Web client access patterns: characteristics and caching implications," *WWW Journal*, 2(1): 3-16, 1999.

[48] L.W. Dowdy and D.V. Foster, "Comparative Models of the File Assignment problem," *ACM Computing Surveys*, Vol.14(2), June 1982.

[49] D. Andresen, T. Yang, V. Hohmedahl and O. Ibarra, "SWEB: Toward a scalable World Wide Web server on multicomputers," *in Proc. of the 10th Int. Symp. on Parallel Processing*, Honolulu, pp. 850-856, April 1996.

[50] V. Cate, "Alex - a global file system," *in Proc. of the 1992 USENIX File System Workshop*, pp. 1-12, May 1992.

[51] P.M.G. Apers, "Data Allocation in Distributed Database Systems," *ACM Trans. On Database Systems*, 13(3), Sep. 1988, pp. 263-304.

[52] S. Mahmoud and J. Riordon, "Optimal allocation of resources in distributed information networks," *ACM Trans. Database Systems*, 1(1), March 1976, pp. 66-78.

[53] B. Narendran, S. Rangarajan and S. Yajnik, "Data Distribution Algorithms for Load Balanced Fault-Tolerant Web Access," *in Proc. of the 16th Symposium on Reliable Distributed Systems (SRDS '97)*, Oct. 22-24, 1997 Durham, NC.

[54] A. Rousskov and D. Wessels, "Cache Digest," *Computer Networks and ISDN Systems*, Vol. 30, No 22-23, Nov. 1998.

[55] Digital Island, available at: http://www.digitalisland.com.

[56] P. Mirchandani and R. Francis, "Discrete Location Theory," *John Wiley and Sons*,

1990.

[57] K.P. Eswaran, "Placement of Records in a File and File Allocation in a Computer Network." *Information Processing*, 1974, pp. 304-307.

[58] V. Cardellini, M. Colajanni and P. Yu, "High performance Web-server systems," *in Proc. of the 13th Int. Symp. On Computer and Information Sciences (ISCIS'98)*, Ankara, Turkey, pp. 288-293, Oct. 1998.

[59] A. Bestavros, M. Crovella, J. Liu and D. Martin, "Distributed Packet Rewriting and its application to scalable server architectures," *in Proc. of the 6th Int. Conference on Network Protocols (ICNP'98)*, Austin, Texas, Oct. 1998.

[60] L. Fan, P. Cao, W. Lin and Q. Jacobson, "Web prefetching between low-bandwidth clients and proxies: potential and performance," *in Proc. of the SIGMETRICS'99 Conf.*, 1999.

[61] L. Aversa and A. Bestavros, "Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting," *in Proc. of the IEEE Int. Performance, Computing, and Communication Conference (IPCCC'2000)*, Phoenix, AZ, February 2000.

[62] L. Bellatreche, K. Karlapalem and L. Qing, "An Iterative Approach for Rules and Data Allocation in Distributed Deductive Database Systems," *in Proc. of the 7th International Conference on Information and Knowledge Management (ACM CIKM'98)*, pp. 356-363, November 1998, Washington D.C.

[63] P. Cao and C. Liu, "Maintaining strong cache consistency in the World Wide Web," *in Proc. of the 17th IEEE Int. Conf. on Distributed Computing Systems (ICDCS'97)*, May 1997.

[64] Akamai Technologies, Freeflow content delivery system, at: http://www.akamai.com.

[65] R. Casey, "Allocation of Copies of a File in an Information Network," *in Proc. of the Spring Joint Computer Conf.*, IFIPS, 1972, pp. 617-625.

[66] R. Ahuja, T. Magnanti and J. Orlin, "Network Flows: Theory, Algorithms, and Applications," *Prentice Hall*, 1993.

[67] S. Jamin, C. Jin, Y. Jin, D. Riaz, Y. Shavitt and L. Zhang, "On the Placement of Internet Instrumentation," *in Proc. of the IEEE INFOCOM'00 Conf.*, March 2000.

[68] Y.K. Kwok, K. Karlapalem, I. Ahmad and N.M. Pun, "Design and Evaluation of Data Allocation Algorithms for Distributed Database Systems," *IEEE Journal on Sel. areas in Commun.(Special Issue on Distributed Multimedia Systems)*, Vol. 14, No. 7, pp. 1332-1348, Sept. 1996.

[69] S. Gadde, M. Rabinovich and J. Chase, "Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches," *in Proc. of the 6th Workshop on Hot Topics in Operating Systems*, pp. 93-98, May 1997.

[70] W. Chu, "Optimal File Allocation in a Multiple Computer System," *IEEE Trans. Computers*, vol. C-18, no. 10, 1969.

[71] IBM Interactive NetworkDispatcher, available at: http://www.ics.raleigh.ibm.com/ netdispatch.

[72] Y. Amir, A. Peterson, and D. Shaw, "Seamlessly selecting the best copy from Internetwide replicated web servers," *in Proc. of the 12th Int. Symposium on Distributed Computing*, Andros, Greece, Sept. 1998.

[73] L.W. Dowdy and D.V. Foster, "Comparative Models of the File Assignment problem," *ACM Computing Surveys*, Vol.14(2), June 1982.

[74] P. Mirchandani and R. Francis, "Discrete Location Theory," *John Wiley and Sons*, 1990.

Appendix-A

The most important notations used in this thesis are summarized in the table below.

Table 1

| Symbol	Meaning
M	total number of servers
N	total number of objects
S_i	the ith server
O_k	the kth object
$s(S_i)$	storage capacity of S_i (in data units)
$s(O_k)$	size of O_k (in data units)
l_{ij}	communication cost (per data unit) between S_i and S_j
X, X^{old}, X^{new}	The (old, new) replication matrix / placement
P_k	primary server of O_k
N_{ik}^X	replicator of O_k nearest to S_i in replica placement X
r_{ik}	read volume arriving at S_i for O_k
T_{ikj}	transfer of O_k from S_i to S_j
D_{ik}	deletion of O_k on S_i
H	schedule of transfer/delete actions
C^{H_u}	cost of the uth action of schedule H
$I_H^{X,X'}$	cost of a valid schedule H that leads from X to X'

Appendix-B

One may prove[2] 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 [30]: Given n objects, with positive benefit values $b_1, b_2, ..., b_n$ and non zero sizes $s_1, s_2, ..., s_n$, is there a subset W of these objects, such as $\sum_{i \in W} s_i < S$ and $\sum_{i \in W} b_i > K$ (S, K are positive integers)?

Assuming an instance of the $(0,1)$ Knapsack-decision, we construct an equivalent RTSP instance as follows. Consider a network of $n+3$ servers $S_1, ..., S_{n+3}$ and $n+1$ 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 $s(O_{n+1}) = \sum_{1 \leq i \leq n} s(O_i)$. For each Knapsack object O_i we define S_i to be the primary server of it. Initially, $S_1, ..., S_n$ store only the respective primary replicas in X^{old}. S_{n+1} has a storage capacity of $s(S_{n+1}) = S + \sum_{1 < i < n} s(O_i)$ (where S is the Knapsack size) and stores only O_{n+1} in X^{old}. S_{n+2} has a storage capacity of $s(S_{n+2}) = \sum_{1 < i < n} s(O_i)$ and stores all Knapsack objects in X^{old}. Finally, S_{n+3} has a storage capacity of O_{n+1} and stores only the primary copy of O_{n+1} in X^{old}. The following links exist: (i) a link between S_{n+1} and S_{n+2} with link cost 1, (ii) links between the primary servers of the Knapsack objects and S_{n+1}, each of link cost $l_{in+1} = b_i'$, $1 < i < n$, where

$$b_i' = \frac{b_i \prod_{1 \leq i \leq n} s(O_i)}{s(O_i)},$$ (iii) a link between S_{n+3} and S_{n+2} of cost $\sum_{1 \leq i \leq n} (b_i' + 1)$. Figure A1 illustrates this setup.

---

[2] The NP-completeness proof was provided by Dr. Thanasis Loukopoulos.
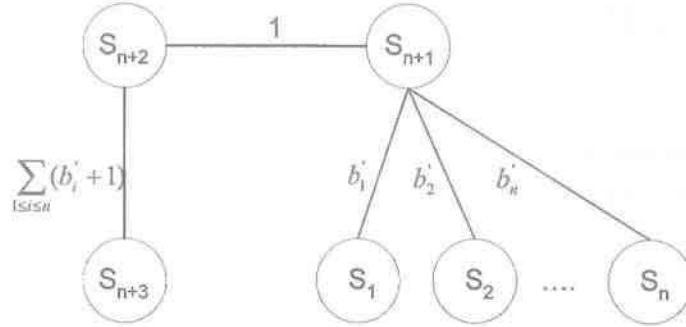
Figure A1: Network structure for reducing knapsack to RTSP

X^{new} is set to be the same as X^{old} with the exception that S_{n+1} and S_{n+2} must exchange their hosted replicas. Consider an optimal order of actions $H\text{-}OPT$, implementing X^{new}. We observe that the transfer of O_{n+1} from its primary server can not belong to $H\text{-}OPT$ 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 this be H') that starts with the transfer of O_{n+1} from S_{n+1} and continues with the transfers of all Knapsack objects from their respective primary servers for a total cost of $\sum_{1 < i < n} s(O_i) + \sum_{1 < i < n} b_i' s(O_i)$. Thus, $H\text{-}OPT$ must contain the transfer of O_{n+1} from S_{n+1}.

We also observe that $H' \ne H\text{-}OPT$, since S_{n+1} starts with S unused storage space, which can be used to transfer at least one Knapsack object at a cost lower than performing the transfers from the respective primary servers ($s(O_i)$ compared to $b_i s(O_i)$). Therefore, $H\text{-}OPT$ 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 their primary servers 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: $I - OPT^{X^{old}, X^{new}} = \sum_{i \in W' \wedge i \neq n+1} s(O_i) + \sum_{1 \leq i \leq n} s(O_i) + \sum_{i \notin W' \wedge i \neq n+1} b_i' s(O_i)$.

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: $\sum_{i \in W' \wedge i \neq n+1} s(O_i) + \sum_{1 \leq i \leq n} s(O_i) + \sum_{i \notin W' \wedge i \neq n+1} b_i' s(O_i)$ (A1).

After substitutions (A1) gives: $\min(\sum_{i \in W' \wedge i \neq n+1} s(O_i) + \prod_{1 \leq i \leq n} s(O_i) \sum_{i \notin W' \wedge i \neq n+1} b_i)$, since $\sum_{1 \leq i \leq n} s(O_i)$ is

constant. But notice, that the following holds: $\prod_{1 \leq i \leq n} s(O_i) b_i > s(O_i) \forall 1 \leq i < n$ (A2)

Thus, (A1) reduces to $\min(\prod_{1 \leq i \leq n} s(O_i) \sum_{i \notin W' \wedge i \neq n+1} b_i)$, which gives $\min(\sum_{i \notin W' \wedge i \neq n+1} b_i)$ since $\prod_{1 \leq i \leq n} s(O_i)$

is constant. 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 corresponds to 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*:

$I^{X^{old}, X^{new}} < \sum_{\forall i} s_i + (\sum_{\forall i} b_i - K) \prod_{\forall i} s_i + S$ (by (A1) and (A2)). If *H* exists so does a solution to

the (0,1) Knapsack-decision instance.

Appendix-C

Experimental server network topology