Dynamic Replica Placement in Content Delivery Networks

Francesco Lo Presti
Dipartimento di Informatica
Università de l'Aquila
lopresti@di.univaq.it

Chiara Petrioli and Claudio Vicari
Dipartimento di Informatica
Università di Roma "La Sapienza"
{petrioli,vicari}@di.uniroma1.it

Abstract

The Content Delivery Networks (CDN) paradigm is based on the idea to transparently move third-party content closer to the users. More specifically, content is replicated on CDN servers which are located close to the final users, and user requests are redirected to the "best" replica (e.g. the closest) in a transparent way, so that users perceive a better content access service.

In this paper we address the problem of dynamic replica placement. Being dynamic, our solutions adaptively select the number of replicas for each content and the replicas positions to account for traffic requests dynamics. The schemes we propose are designed to minimize the overall cost paid by the CDN provider (for replicas placement, removal, and maintenance) without degrading the quality of the users perceived access service.

The contributions of the paper are twofold. First we introduce a centralized and distributed scheme for replica placement in a dynamic traffic scenario. Then, by means of a simulation based performance evaluation, we assess the effectiveness of the proposed schemes, and compare their performance with static solutions which have been proven to perform well in the literature. Simulation results show that both the two proposed algorithms achieve very good performance, resulting in a significant improvement over the static solutions. Despite relying on local information only, the distributed scheme has comparable performance to the centralized one. Both the two schemes result in low average distance between the users and their serving replicas, in low average number of replicas, in infrequent replicas add and tear down, and in high probability of being able to serve a request.

1 Introduction

Content Delivery Networks (CDNs) are one of the answers to the challenges posed by the remarkable commercial success of the Internet. Replicating third-party content on servers closer to the final users, and redirecting transparently the user requests to the "best" replica (e.g., the closest replica in terms of distance, latency, etc.) CDN providers are able to offer improved content access service.

Solutions for CDN thus require to address a number of technical problems, which include: 1) Deciding the kind of content that should be hosted (if any) at a given CDN server (replica placement), 2) selecting the best replica for a given user, and 3) designing mechanisms for transparent redirection of the users requests to the best replicas.

This paper concerns the definition and evaluation of algorithms for replica placement which minimize the CDN provider costs (in terms of average number of replicas, number of replicas added and removed over time), while at the same time provide a good access service to the users. The latter is expressed in terms of service availability, i.e. assurance that the users will always be able to find a replica which can serve them with an acceptable latency, and in terms of low average distance between the served user requests and the serving best replica). Our solutions are dynamic in the sense that replicas are added and removed from CDN servers according to the dynamically changing user request traffic.

Of the solutions proposed in the literature for replica placement, most concern the static case. Basically, given the topology of the network, the set of CDN servers as well as the request traffic pattern, replicas are placed so that some objective function is optimized while meeting constraints on the system resources (server storage, server sustainable load, etc.). Typical problem formulations aim at either maximizing the user perceived quality given an upper bound on the number of replicas, or minimizing the cost of the CDN infrastructure while meeting constraints on the user perceived quality (e.g., latency) [1].

For the static case, simple efficient greedy solutions have been proposed in [2], [3] and [4]. In [2] Qiu et al. formulate the static replica placement problem as a minimum K median problem, in which K replicas have to be selected so that the sum of the distances between the users and their best replica is minimized. A simple greedy heuristic is shown to

have performance within 50% of the optimal strategy. An evaluation of such greedy scheme to assess the impact of K on the user satisfaction (closeness to the replica), for different traffic patterns, is presented in [5]. Qiu et al. have also proposed "hot spot," a solution for placing replicas on nodes that along with their neighbors generate the greatest load [2]. In [6] "hot zone" is presented as an evolution of the "hot spot" algorithm. The idea is to first identify network regions made of nodes whose latency to each other is low. Regions are then ranked according to the content request load that they generate and replicas are placed in the regions high in the ordering. In [3] and [4] Jamin et al. and Radoslavov et al. propose fan-out based heuristics in which replicas are placed at the nodes with the highest fan-out irrespective of the actual cost function. The rationale is that such nodes are likely to be in strategic places, closest (on average) to all other nodes, and therefore suitable for replica location. In [4] a performance evaluation based on real-world router-level topologies shows that the fan-out based heuristic has trends close to the greedy heuristic in terms of the average client latency.

A major limit of all these solutions is that they neglect to consider the natural dynamics in the user requests traffic. When network or user traffic changes occur which would make a different replica placement less costly or more satisfying for the users, the only possible solution is to re-apply the placement algorithm from scratch. This approach has a couple of problems. First of all, it may react slowly to the system changes, so that the new placement of the replicas is not the best one for the current user request traffic. Moreover, the replica placement happens every time from scratch, i.e., without considering where replicas are currently placed. This could possibly lead to non-negligible reconfiguration costs.

A few papers (e.g., [7] and [8]) have addressed the problem of dynamic replica placement. However, the proposed schemes are embedded in specific architectures for performing requests redirection and computing the best replicas. No framework is provided for identifying the optimal strategy, and for quantifying the solutions performance with respect to the optimum. In RaDar [7] a threshold based heuristic is proposed to replicate, migrate and delete replicas in response to system dynamics. The overall proposed solution combines dynamic replica allocation with servers load-aware redirection to the best replica to achieve low average users latency while empirically balancing the load among the CDN servers. No limits on the servers storage and on the maximum users latency are explicitly enforced. In [8] two schemes designed for the Tapestry architecture [9] are presented. The idea is that upon a content request the neighborhood of the user access point in the overlay network is searched. If there is a server hosting a replica of the requested content within a maximum distance from the user, and such server is not overloaded, the request will be redirected to this server (or to the closest server if multiple servers meet such constraints). Otherwise a new replica is added to meet the user request. Two variants are introduced depending on the neighborhood of the overlay network which is searched for replicas, and on the scheme used to select the best location for the new replica. Although the ideas presented in the paper appear promising they are tightly coupled with the Tapestry architecture, and the approach does not explicitly account for neither the costs of reconfiguration nor for possible servers storage limits. Finally, no information is provided in [8] on the rule to remove replicas, making it hard to compare with our approach. Recently, the authors have presented a framework for dynamic replica placement in [10]. By assuming the user requests dynamics to obey to a Markovian model the problem of optimal dynamic replica placement has been described as a semi-Markov decision process accounting for the traffic, the user level of satisfaction as well as the costs paid to add, maintain or remove a replica from CDN servers. Although this model allows us to achieve optimality and provides insights to the dynamic replica placement problem, it is not scalable, making it unusable in practice for controlling the operation of an actual CDN network. In this work we leverage these limits proposing and evaluating simple, distributed, scalable heuristics for dynamic replica placement. First, based on the outcomes of the resolution of the model proposed in [10] we derived criteria on when and where to add/remove replicas. Such criteria were the basis for the design of a centralized heuristic. Then a fully distributed scheme is introduced. The distributed solution is based on the idea that, for the CDN providers to see the infrastructure costs paid off, and at the same time for the user requests to be promptly served, replicas utilization should be within a desirable interval $[t_{min}, k)$. Overloaded replicas (serving k user requests or more) should clone themselves, as, otherwise, the user access performance would degrade severely, and the load should be shared among the replica and its clone. Under-loaded replicas (serving less than t_{min} user requests) are a cost for the CDN provider. Our scheme tries to discourage redirection of the requests both toward overloaded and under-loaded replicas. When a replica is totally unused (which happens in case of under-loaded replicas when their assigned user requests can be redirected toward alternative replicas) then the replica is discarded. Decisions are taken at the CDN servers based only on local knowledge of the CDN network status (traffic redirected to the server, server local neighborhood and the replicas at those neighbors)

A comparative performance evaluation between the centralized heuristic, the distributed scheme sketched above and (static) solutions previously introduced in the literature allows us to quantify the advantages that can be ob-

tained by a dynamic replica placement scheme and to assess the effectiveness of the proposed solutions in limiting costs and providing excellent users perceived quality. In particular, the distributed heuristic shows surprisingly good results. Despite the local exchange of information and the limited knowledge of the users traffic and server status, its performance is close to those of the centralized scheme. The centralized heuristic proposed leads to the same or slightly better performance than the greedy static schemes used as benchmarks: the performance are basically the same in terms of average number of replicas, replica utilization, av. distance to the best replica, percentage of unsatisfied requests. Both the dynamic centralized and distributed schemes however perform (orders of magnitude) better in terms of reconfiguration costs.

The paper is organized as follows. In Section 2 the problem dealt with in this paper is introduced. In Section 3 we describe the heuristics (both centralized and distributed) proposed for replica placement. Also the static schemes selected for sake of benchmarking will be described. In Section 4 the simulation framework is described, and the results of a comparative performance evaluation among the different schemes are reported. Finally conclusions end the paper in Section 5.

2 Problem Formulation

Users access the CDN network through one among $|V_A|$ access points. They request access to one among C possible contents. The number of user accesses is expressed in units of aggregate requests from a given access site for a specific content. No more than $V(A)_{max}$ units of aggregate requests can be generated by an access node (to model the limited access link bandwidth). Replicas of the C contents can be stored in one or more among a number $|V_R| \geq 0$ of CDN servers sites. Each site $v_r \in V_R$ can host up to $V(R)_{max}$ replicas (storage limit), each serving up to k aggregate user requests (load threshold).

The whole CDN network is represented as a graph whose nodes are $V_R \cup V_A$ and whose edges are the physical paths interconnecting the nodes. Each edge (i, j) has associated a weight $d(i, j)$ that is computed as the sum of the costs needed to traverse the physical links of the shortest path interconnecting i and j. This cost can model latency, distance, bandwidth of the traversed link, etc.

A request for content c originated at site i is redirected to its best replica j by the transparent redirection scheme. This request is said to be satisfied if and only if $d(i, j) \leq d_{max}$, d_{max} being a given threshold.

The proposal of a distributed redirection scheme is beyond the purpose of this paper. However, we make the common assumption that such scheme exists and that it balances the load of user requests among the replicas. The redirection scheme is also aware of the level of request load of the different replicas via periodic measurements on the replica sites. A replica is said to be overloaded if its request load reaches k, and underutilized when it is below a threshold t_{min}. The parameter k is chosen to indicate the sustainable load of requests for a specific replica, beyond which the user access service suffers severe degradation. The situation when replicas are underutilized is costly for the CDN provider as the replica maintenance costs are not not paid off by the limited use of the replica. We assume that the redirection mechanism considered for our replica placement solution attempts to avoid overloaded and underutilized replicas. The detailed implementation of the transparent redirection mechanism used in the performance evaluation of our solutions is described in Section 3.3.

Dynamic replica placement is formulated as follows. Let the configuration of requests and replicas at a given time be expressed by means of a state vector $\mathbf{x}(t)$ of size $C(|V_A| + |V_R|)$: $\mathbf{x}(t) = (\mathbf{a}(t), \mathbf{r}(t)) = (a_1^1(t), a_2^1(t), \ldots, a_{|V_A|}^1(t), a_1^2(t), \ldots, a_{|V_A|}^2(t), \ldots, a_{|V_A|}^C(t), r_1^1(t), r_2^1(t), \ldots, r_{|V_R|}^1(t), r_1^2(t), \ldots, r_{|V_R|}^2(t), \ldots, r_{|V_R|}^C(t))$ in which the variable $a_i^c(t)$ represents the number of request units for a content $c \in \{1, \ldots, C\}$ at node $i \in V_A$ at time t, and $r_j^c(t)$ is the number of replicas of content $c \in \{1, \ldots, C\}$ placed at site $j \in V_R$ at time t.

We associate to a state $\mathbf{x}(t)$ a CDN cost given by two components: The cost for replica maintenance, given by $C_M \sum_{j \in V_R, c=1,\ldots,C} r_j^c(t)$, where C_M is the cost associated to the maintenance of one replica at time t. Two other costs C^+ and C^- are paid by the CDN provider when dynamically adjusting the number of replicas. These costs are associated to the decision to add or remove a replica, respectively.

Aim of the dynamic replica placement schemes is that of minimizing the long-run costs associated to replica adds, removals and maintenance while meeting the constraints on the replica maximum load, the maximum number of replica per site, and while satisfying all users (dynamic) traffic demands (if feasible).

3 Algorithms Description

3.1 Static Replica Placement

We compared our solutions to static schemes which have been proposed in the literature and which have been proven to lead to good performance. Among the schemes introduced for static scenarios, we have selected the greedy heuristic introduced by Qiu et al. in [5] as it combines very simple rules with excellent performance. The model proposed in [5] does not account for practical constraints included in our model, such as 1) the maximum distance d_{max}

between user requests and their serving replicas, 2) the limit on the storage available at a site (number of replicas which can be allocated), and 3) the limit on the maximum number of user requests, which can be effectively served by a replica. We have therefore slightly modified the Qiu et al. greedy scheme to account for these features of our model without changing its philosophy. The resulting greedy algorithm is described in the following. In the description we first review the original Qiu et al. scheme and then we explain how it has been modified to map to our problem formulation.

Qiu's scheme is based on the idea of progressively adding replicas (up to a maximum number k) trying to maximize the access service quality perceived by the final users. Such quality is inversely proportional to the weight of the path joining the access site and the serving replica. Replicas for a content c are allocated greedily, one after another, as follows. The hosting site v^1 for the first replica is selected so that $v^1 = \min_{j \in V(R)}\{\sum_{i=1}^{|V_A|} a_i^c(t) d(i,j)\}$. In other words, v^1 minimizes the sum of the distances between the users access sites and the replica site. The site v^i at which the ith added replica is hosted is selected so that the set $v^1, v^2, \ldots, v^i$ minimizes the distance between the users requests and their serving replicas.

In our model, user requests can only be satisfied by replicas at distance $\leq d_{max}$. It is therefore not important to minimize the sum of the distances between the users and their serving replicas (as the quality of the user access is defined in terms of d_{max}). What matters here is the minimization of the number of replicas needed to satisfy all the user requests while at the same time meeting the load and storage constraints. The greedy approach in our new problem formulation results in selecting each time as new replica site the one that still has available storage and that can best increase the number of user requests satisfied.

The described greedy criterion is depicted in Algorithm 2.

The greedy procedure takes as input a snapshot of the user requests $\mathbf{a}(t)$, the set of possible replica sites V_R, the maximum number of replicas per site $V(R)_{max}$, and the maximum load per replica k. The output produced by the procedure is the vector $\mathbf{r}(t)$ that indicates the number of replicas to be allocated, their location, and the content of each replica.

At the start of the procedure operation the set $\mathbf{r}(t)$ is empty (no replica has been allocated. Line 1). A new replica is added by selecting the pair (replica site, replica content), (j, c), that satisfies the highest number of new requests (Lines from 4 to 16). (Possible ties are broken by using the sum of the distances between the users and the replica.)

In particular, all possible replica sites $j \in V_R$ which still have storage for replicas are examined. The number

Algorithm 2 Static Replica Placement

1: $\mathbf{r}(t) = \mathbf{0}$
2: **while** $MIN\_MATCHING(\mathbf{a}(t), \mathbf{r}(t))! = |\sum_i \sum_c a_i^c(t)|$ **do**
3: $max\_serv = 0; rep\_site = 0;$ $rep\_cont = 0; rep\_distance = MAXINT;$
4: **for all** $j \in V_R$ **do**
5: **if** $\sum_c r_j^c(t) < V(R)_{max}$ **then**
6: **for** $c = 1$ to C **do**
7: $serv = |MIN\_MATCHING(\mathbf{a}(t), \mathbf{r}(t) + \mathbf{e}_j^c)|$
8: **if** $(serv > max\_serv) or (serv = max\_serv) and$ $\sum_c \sum_i a_i^c(j,t) \cdot d(i,j) < rep\_distance$ **then**
9: $max\_serv = serv;$
10: $rep\_site = j;$
11: $rep\_cont = c;$
12: $rep\_distance = \sum_c \sum_i a_i^c(j,t) \cdot d(i,j);$
13: **end if**
14: **end for**
15: **end if**
16: **end for**
17: $\mathbf{r}(t) = \mathbf{r}(t) + \mathbf{e}_{rep\_site}^{rep\_cont}$
18: **end while**

of user requests which can be satisfied by the replica vector $\mathbf{r}(t)+\mathbf{e}_j^c$ is computed and stored in the variable $serv$, where $\mathbf{e}_j^c = \langle 0, \ldots, 0, 1, 0, \ldots, 0 \rangle$ is the vector with a 1 in the $(c-1)|V_R| + j$th position. $serv$ contains the overall number of user requests that could be satisfied by adding one replica of content c at site j.

The value of $serv$ is obtained by running C minimum matching procedures (one for each content). The matching for content c is performed over a bipartite graph whose nodes are partitioned into the set of user requests for c and the set of replicas of c allocated in $\mathbf{r}(t)+\mathbf{e}_j^c$. Since each replica can serve up to k requests, k nodes are needed to represent each replica in the second set. A finite-cost edge joins a user request and one of the k instances of a replica if and only if the distance between the user request access site and the replica hosting site is $\leq d_{max}$. When the distance is $> d_{max}$, an edge is added with infinite cost.

Here we call "cardinality of the minimum matching" the number of edges with finite cost in the solution output by the minimum matching procedure. The cardinality of the minimum matching for content c represents the maximum number of user requests for content c that can be satisfied by the current replicas allocation $\mathbf{r}(t)+\mathbf{e}_j^c$. The procedure call $MIN\_MATCHING(\mathbf{a}(t), \mathbf{r}(t)+\mathbf{e}_j^c)$ produces as output the sum of the cardinalities of the C minimum matchings, namely, the overall number of

user requests which can be satisfied at this time. The (site) j and the (content) c that maximize the output of $MIN\_MATCHING(\mathbf{a}(t), \mathbf{r}(t)+\mathbf{e}_j^c)$ are selected as the site and the content of the next replica.

In case of request traffic dynamically changing, the described procedure has to be run periodically. We consider two alternatives, depending on when the algorithm is run. The first alternative concerns running the algorithm whenever a request traffic change occurs ("instantaneous greedy" solution, referred to as Greedy_inst below). Otherwise, the algorithm is run periodically, every T seconds. In this case, if replicas are allocated according to user requests at the time when the algorithm has been run, when user request traffic changes some requests could go unsatisfied. In order to limit the occurrence of this problem, we have estimated the user requests dynamics for the upcoming time interval (of length T) by using RLS (Recursive Least Square) prediction [11]. This allows us to estimate, based on current and past traffic, the future user requests traffic process from site i to content c, and set a user requests traffic $a_i^c(t)$ as the maximum expected value over the interval T.

3.2 Centralized Solution

If we model the user requests to obey to a Markov process, we can formulate the optimal strategy as Markov Decision Process [10]. This approach, though, poses serious problems in practice since determining the optimal strategy is not feasible in practice. Building on the findings in [10], here we propose a centralized heuristic for the dynamic replica placement problem. The algorithm determines which replica placement/removal decision $\mathbf{d}$ to take (if any) at each state change. The goal of the heuristics is to mimic the MDP optimal policy behavior observed in [10] by minimizing the number of replicas used to serve all existing requests leaving at the same time enough spare capacity to accommodate for requests increases.

The algorithm we propose, shown in Figure 1, works as follows. (For the sake of readability in the description below with $\mathbf{a}$ and $\mathbf{r}$ we mean $\mathbf{a}(t)$ and $\mathbf{r}(t)$.) At each step, it first determines whether the current replica configuration $\mathbf{r}$ can accommodate any possible increase in user requests $\mathbf{a}$. This is accomplished via the function enough_replica_on_increase() (see Figure 2). In case that any possible increase in user requests can be accommodated by $\mathbf{r}$ then the algorithm considers whether it is possible to remove a replica (remove_replica()); otherwise it tries to find a site where to add a replica (add_replica()). (Observe that to mimic the behavior of a Markovian Decision Process, actions are decided in a given state but only taken in correspondence of the next transition.)

The function enough_replica_on_increase($\mathbf{a}$, $\mathbf{r}$)) returns TRUE if any possible increase in $\mathbf{a}$ can be served by the current replica configuration $\mathbf{r}$ and FALSE otherwise. To this end, it uses the function enough_replica($\mathbf{a}$, $\mathbf{r}$)) which determines whether a given users access requests $\mathbf{a}$ can be served by the set of replicas $\mathbf{r}$. (The function enough_replica() itself is computed by solving a minimum matching problem between users requests and the available replicas from the solution of which we can determine whether all request in $\mathbf{a}$ can be served by $\mathbf{r}$.)

```
1. d =do nothing;
2. while ( TRUE ) {
3.     wait for a change in a; take action d;
4.     if ( enough_replica_on_increase( (a,r ) )
5.         d=remove_replica();
6.     else
7.         d=add_replica();
8. }
```

Figure 1. Replica Placement Algorithm.

```
1. boolean enough_replica_on_increase( state (a,r ) ) {
2.     for any c = 1, ..., C and i ∈ V_A
3.         if ( !enough_replica( (a + e_i^c, r) ) return FALSE;
4.     return TRUE;
5. }
```

Figure 2. enough_replica_on_increase().

add_replica() is called to determine content and location for a new replica. To this end it first identifies which requests increase would require additional replicas. This is accomplished in line 2 by determining the set I of the pairs (i, c) such that an increase of requests for content c at node i cannot be served by $\mathbf{r}$ (line 2). It then computes the sets $J(j, c) \subseteq I$ of users requests increment that could be served by an additional replica of content c in site j (line 3). If not all $J(j, c)$ are empty , the content and location of the additional replica is then chosen by finding the site j^* and content c^* which maximizes $|J(j, c)|$ (line 7). This to maximize the probability of being able to satisfy a request increase.

The procedure remove_replica() is called to determine whether to remove a replica. To this end, first it identifies the set U of replicas which should not be removed as they would be needed to serve an increase in users requests (line 2). Then, it determines, among the remaining replicas, the set J of the candidates for possible removal, i.e., all those replicas which are not used to serve current requests (line 3). Among these, it chooses to remove a replica from a node j which serves the smallest set of access nodes (line 5). Choosing the replica which serves the smallest population should minimize the likelihood to remove a replica which is going to be added soon again.

1. action remove_replica ( ) {
2. find $U = \{(j,c)\ j \in V_R, c = 1,\ldots,C \mid \exists i \in V_A$ enough_replica $((\mathbf{a}+\mathbf{e}_i^c, \mathbf{r}))$ AND !enough_replica $((\mathbf{a}+\mathbf{e}_i^c, \mathbf{r}-\mathbf{e}_j^c)),$ $(\mathbf{a}+\mathbf{e}_i^c, \mathbf{r}-\mathbf{e}_j^c) \in \Lambda\}$;
3. find $J = \{(j,c)\ j \in V_R, c = 1,\ldots,C\ (j,c) \notin U \mid$ enough_replica$((\mathbf{a}, \mathbf{r}-\mathbf{e}_j^c)),\ (\mathbf{a}, \mathbf{r}-\mathbf{e}_j^c) \in \Lambda\}$;
4. if ($|J| > 0$)
5. $(j^*, c^*) = \text{argmin}_{(j,c)\in J}|S_{j^*}|$, $S_j = \{i,\ i \in V_A \mid distance(i,j) \leq d_{Max}\}$;
6. $\mathbf{d}$=remove replica c^* in site j^*;
7. else
8. $\mathbf{d}$=do nothing;
9. return $\mathbf{d}$;
10. }

Figure 4. Algorithm for Deciding which Replica to Remove.

1. action add_replica ( ) {
2. find $I = \{(i,c)\ i \in V_A, c = 1,\ldots,C \mid$!enough_replica$((\mathbf{a}+\mathbf{e}_i^c, \mathbf{r}))\}$
3. for any $j \in V_R, c = 1,\ldots,C$
4. find $J(j,c) = \{(i,c) \in I \mid$ enough_replica$(\mathbf{a}+\mathbf{e}_i^c, \mathbf{r}+\mathbf{e}_j^c),$ $(\mathbf{a}+\mathbf{e}_i^c, \mathbf{r}+\mathbf{e}_j^c) \in \Lambda\}$
5. if $(\max_{(j,c)} |J(j,c)| > 0)$
6. $(j^*, c^*) = \text{argmax}\ |J(j,c)|$;
7. $\mathbf{d}$=place replica content c^* in site j^*;
8. else
9. $\mathbf{d}$=do nothing;
10. return $\mathbf{d}$;
11. }

Figure 3. Algorithm for Deciding where to Place a New Replica.

3.3 Distributed Heuristic

In this section we describe a *distributed* scheme to allocate/deallocate replicas so that the user requests are satisfied while minimizing the CDN costs in a dynamic scenario. Being dynamic this scheme always account for the current replica placement, adding replicas or changing replica location only when needed. Each site $j \in V_R$ decides on whether some of the replicas it stores should be cloned or removed. This decision is based on local information such as the number and content of the replicas stored at j, the load of such replicas, and the user requests served by the replicas hosted at the site. We also assume that each site j stores information about its local neighborhood in the CDN network topology. More specifically, site j knows the set $\alpha(j)$ of the nodes in V_A which are distant at most d_{max} from it, and the set $\rho(j)$ of the nodes in V_R that are distant at most d_{max} from any of the nodes in $\alpha(j)$. The first set includes all those access sites which can generate requests that j can satisfy. The set $\rho(j)$ is the set of serving sites that can cover for j. Of these sites, j maintains information about the replicas they host, and their content. Based on this sole information, site j is able to decide when to either clone or delete a replica, and in case of cloning, where the clone should be hosted. In particular, if (and only if) one of the replicas hosted at j is overloaded (i.e., serves k requests), j decides to clone it. Cloning a replica implies the selection of a host for the clone. To this aim site j selects the site $j' \in \rho(j)$ that satisfies the following requirements: It still has storage for hosting new replicas, and it is able to satisfy the largest amount of user requests for the content of the replica among the requests currently redirected to j (ties are broken selecting the hosting site j' closest to the user requests). Site j then contact site j' asking it to host the clone. Upon confirming availability, j physically sends the clone to j', and j' informs all the access sites in $\alpha(j')$ and all the serving sites in $\rho(j')$ of the new replica that it is hosting.

If (and only if) one of site j's replicas can be removed without affecting the capability of satisfying all the requests served by the replicas at j, then j removes that replica. The decision of removing a replica is based on a weighted average of the user requests currently redirected to the replica and past requests. This provides some smoothing in the decision process and helps avoiding the ping pong effect for which a replica is added and soon removed. Upon deciding to remove a replica, node j informs all the nodes in $\alpha(j)$ and in $\rho(j)$.

As mentioned, the redirection scheme we use performs load balancing among the different replicas and prevents overloaded or underutilized replicas to be selected as "best" replicas unless needed. This load balancing redirection scheme allows both to divert requests from underutilized replicas (thus removing such replicas whenever possible) and it also motivates the need for cloning the replica whenever the maximum threshold k is reached. Not only the extra replica will be able to serve some of the user requests reducing load and providing a better service to the final users, but also, the case of overloaded replicas at j implies that none of the other replicas hosted at sites distant $\leq d_{max}$

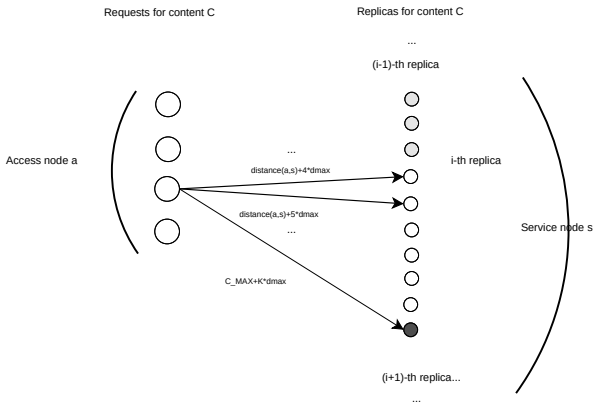

Figure 5. A model for the redirection scheme

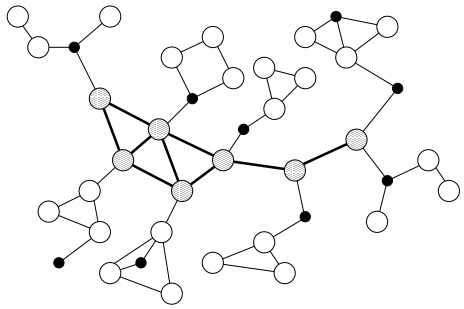

Figure 6. Network Topology used in the Example.

from the user requests could cover for j without reaching the threshold k themselves. In such scenario it is unavoidable to add an extra replica.

It is worth spending a few lines for introducing the way we implemented the redirection scheme used for all our heuristics. In order to realize the necessary load balancing of the redirection scheme, we have represented the set of current user requests and the set of allocated replicas by a bipartite graph. See Figure 5.

As described earlier, the nodes on the left represent the current users requests. The nodes on the right are the allocated replicas. (Instances of a replica corresponding to overloaded or under-used states are colored in black and gray in the figure.) Load balancing and the restricted use of underutilized and overloaded replicas is achieved by properly setting the weights of the edges of the bipartite graph. In particular, as shown in the figure, load balancing is achieved by weighting the edges as a function of a replica's load: The higher the load, the higher the weight. Edges to the kth instance of a replica (whose use would overload that replica) are associated to much higher weights (via the addition of a high constant C_{MAX}). While this discourages the selection of that edge, it still makes that link available in those cases where a user request would otherwise remain unsatisfied. Similarly, in the case of underutilized replicas, edges leading to those replicas are associated much higher weights in the attempt of discouraging their use and to check whether requests could be satisfied without them.

The last case to be considered is the following. It might happen that a user requests access to a content c from an access site i which does not have any replica of content c within d_{max} (this is often the case when the replica allocation process starts). In this case the request is directed to the origin server that clones a copy of its content to a replica site j that is distant at most d_{max} from i. Among the possible sites, the origin server selects the site j that can satisfy the requests originated by the largest number of sites of V_A. The selected site is clearly highly likely to be able to satisfy the largest number of requests in the near future.

4 Simulation Results

In this section we evaluate and compare the performance of the proposed algorithms by means of simulation. Due to the limited space here we concentrate on the simple topology in Figure 6 with 24 access nodes and 7 service nodes (sites). The thin lines denote slower links (with a weight of 2), the thick ones faster links (with a weight of 1). Users issue requests for C different types of contents. The aggregate requests at site $i \in V_A$ for content c are modeled as independent Markovian birth-death process with birth rate equal to 0.001 and death rate equal to 0.0001. We set $k = 15$, $V(A)_{\texttt{MAX}} = 30$, $V(R)_{\texttt{MAX}} = 10$. For the distributed algorithm, we set $t_{min} = 3$.

To compare the different algorithms we consider different metrics. We measure the user perceived quality by computing the average distance between the node issuing a request and the replica serving it. The CDN costs are defined in terms of the average number of replicas and of the number of replica added and removed. Finally, we measure the percentage of unsatisfied requests. We select the instantaneous greedy static algorithm as benchmark for comparing the performance of the other algorithms.

Results are illustrated in Figures 7-12 for $C = 1, 2$ and 5 and $d_{max} = \infty$ and $d_{max} = 6$. Results are shown along with the 99% confidence intervals. Additions and removals (only shown for the first set of simulations for space limitations) are computed over an interval of length 100000.

The greedy algorithms generally result in a slightly lower number of replicas and user-replica distances than the dynamic algorithms. As expected, both algorithms have instead very high reconfigurations costs as their decisions are oblivious of prior states. The cost is smaller for the greedy RLS because it recomputes the replica configuration only periodically. This behavior was consistently observed

throughout our experiments.

The dynamic algorithms are characterized by much lower reconfiguration cost, with the minimum attained by the distributed algorithm. The centralized algorithm uses less replicas (almost the same number required by the greedy algorithm), while the distributed algorithm enjoys better average distance but uses more replicas than the other algorithms (about 20% more in these simulations). This behavior can be explained by looking at the algorithms behavior. The centralized algorithm makes placement (and removal) decisions taking into account potential future users request increases and locating the replica where it can serve the largest population. By doing so, the replicas are typically steadily placed in barycentric positions. The average distance is thus greater than that allowed by the greedy algorithms which just shuffle back and forth replicas close to the user according to the instantaneous traffic pattern. The distributed algorithm, on the other hand, uses only local information and lacks coordination among nodes. Each node makes replica addition and removal decisions based on the local load. In this setting the smoothing mechanisms avoid oscillations which would result, for instance, from adding replicas, just removed because of a temporary load reduction. As a result, the distributed algorithm is characterized by a very small number of replicas additions and removals. This yields stable, slow-varying replica configurations, but requires a higher number of replicas. We verified that there are many such stable configurations and the algorithm converges to one of them depending on the dynamics during the initial transient. This also explains the larger spread of the average distance between requests and serving replicas in the distributed algorithm.

Comparing the different figures, we observe that, with a smaller d_{max}, the average number of used replicas increases while the average distance decreases. This reflects the need to place more replicas to meet the stricter constraint on the maximum user-replica distance. On the other hand, when the number of different contents hosted in the CDN increases, more replicas are needed to be able to serve a higher variety of requests.

In all the performed experiments all the instantaneous heuristics never resulted in unsatisfied user requests. The adoption of the RLS prediction turns out to be quite effective in limiting the occurrence of unsatisfied requests to negligible values.

5 Conclusions

We have proposed a distributed and a centralized heuristics which dynamically add or delete replicas from the network depending on the traffic dynamics. The performance of the proposed schemes has been compared with that of (static) greedy schemes which have been proven to perform well in the literature. The results have shown that the proposed dynamic algorithms perform very well in terms of both user perceived quality and CDN infrastructure and reconfiguration costs.

Acknowledgements

We would like to thank Dr. Stefano Basagni for the many useful discussions which improved the quality of the paper.

References

[1] M. Karlsson, C. Karamanolis, and M. Mahalingam, "A unified framework for evaluating replica placement algorithms," *Technical Report HPL-2002, Hewlett Packard Laboratories*, 2002.

[2] L. Qiu, V. N. Padmanabhan, and G. M. Voelker, "On the placement of web server replicas," in *Proceedings if IEEE INFOCOM 2001*, (Anchorage, AK), pp. 1587–1596, April 22–26 2001.

[3] S. Jamin, C. Jin, A. R. Kurc, D. Raz, and Y. Shavitt, "Constrained mirror placement on the internet," in *Proceedings if IEEE INFOCOM 2001*, (Anchorage, AK), pp. 31–40, April 22–26 2001.

[4] P. Radoslavov, R. Govindan, and D. Estrin, "Topology-informed internet replica placement," *Proceedings of WCW'01: Web Caching and Content Distribution Workshop*, June 20–22 2001.

[5] Y. Li and M. T. Liu, "Optimization of performance gain in content distrubution networks with server replicas," in *Proceedings of the 2003 Symposium on Applications and the Internet, SAINT 2003*, (Orlando, FL), pp. 182–189, January 27–31 2003.

[6] M. Szymaniak, G. Pierre, and M. van Steen, "Latency-driven replica placement." Submitted for publication, May 2004. http://www.globule.org/publi/LDRP_draft.html.

[7] M. Rabinovich and A. Aggarwal, "RaDaR: a scalable architecture for a global Web hosting service," *Elsevier Computer Networks*, vol. 31, no. 11–16, pp. 1545–1561, 1999.

[8] Y. Chen, R. Katz, and J. Kubiatowicz, "Dynamic replica placement for scalable content delivery," in *International Workshop on Peer-to-Peer Systems, IPTPS 2002*, (Cambridge, MA), March 7–8 2002.

[9] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communication*, vol. 22, January 2004.

[10] N. Bartolini, F. Lo Presti, and C. Petrioli, "Optimal dynamic replica placement in Content Delivery Networks," in *Proceedings of the* 11*th IEEE International Conference on Networks, ICON 2003*, (Sydney, Australia), pp. 125–130, September 28–October 1 2003.

[11] G. Box, G. Jenkins, and G. Reinkel, *Time Series Analysis: Forecasting and Control*. Prentice Hall, third ed., 1994.

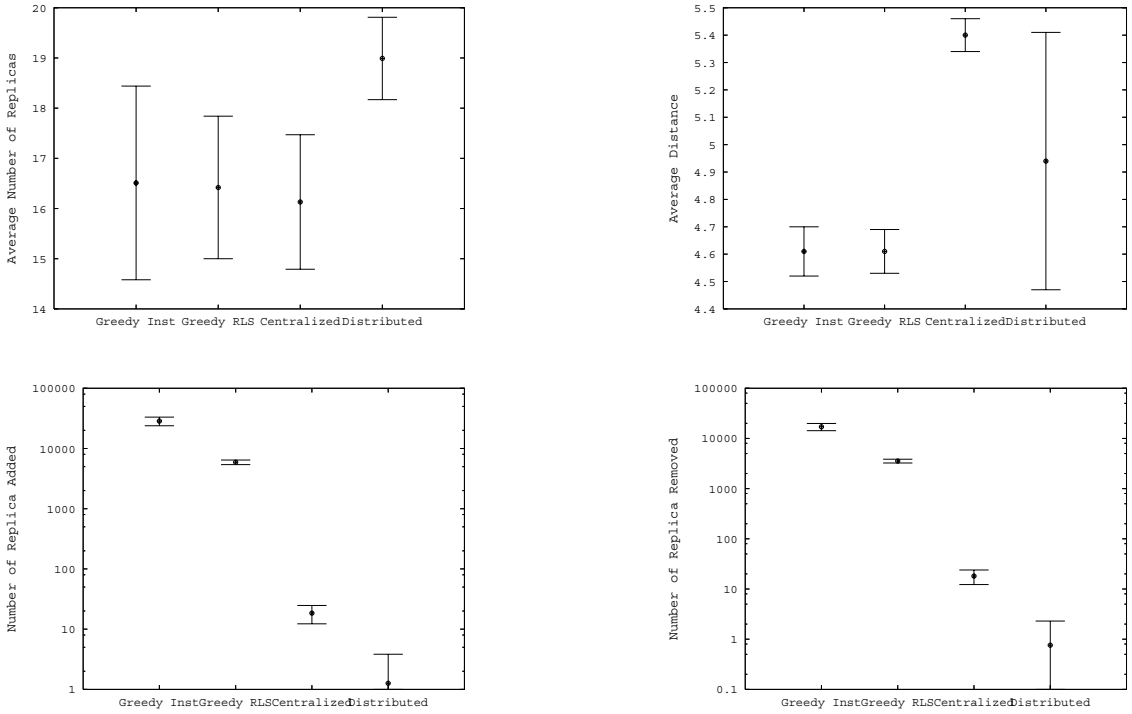

Figure 7. Simulation Results one content and $d_{\max} = \infty$.

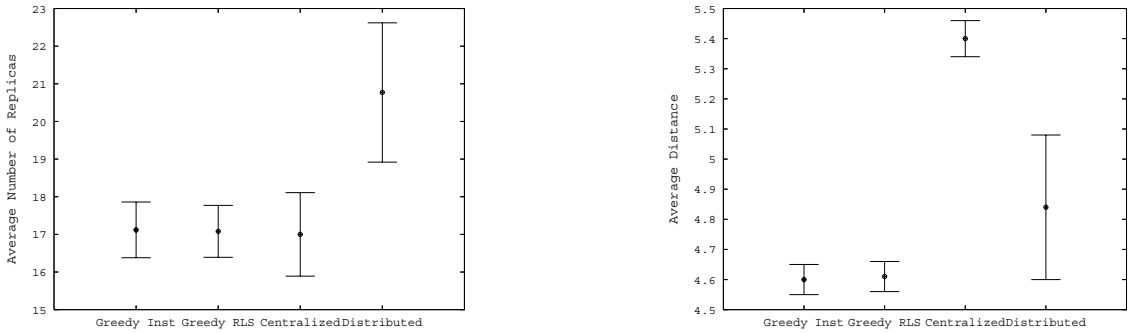

Figure 8. Simulation Results two contents and $d_{\max} = \infty$.

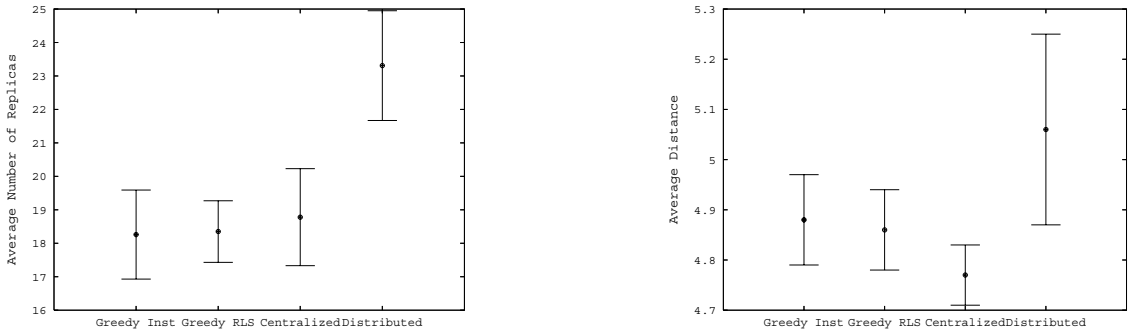

Figure 9. Simulation Results five contents and $d_{\max} = \infty$.

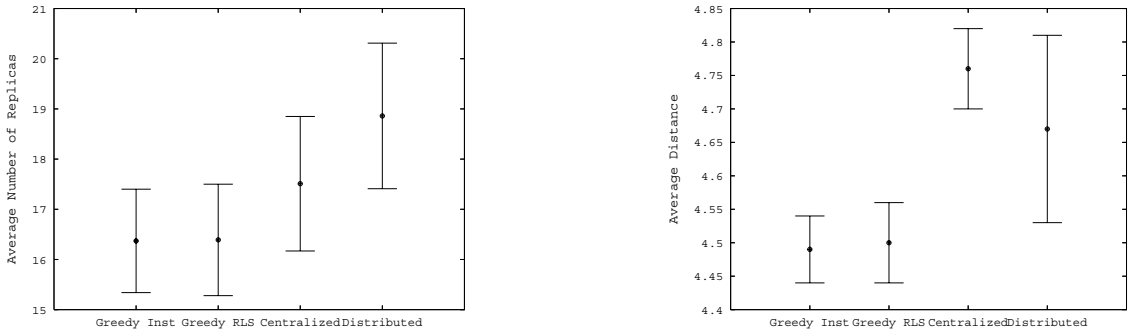

Figure 10. Simulation Results one content and $d_{\max} = 6$.

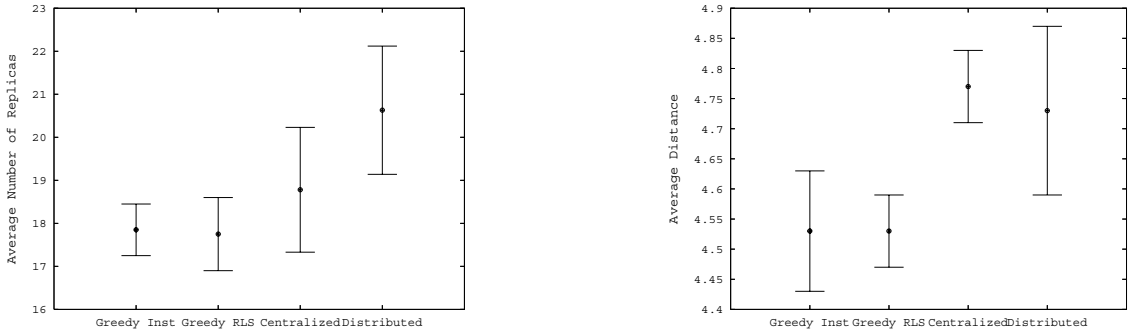

Figure 11. Simulation Results two contents and $d_{\max} = 6$.

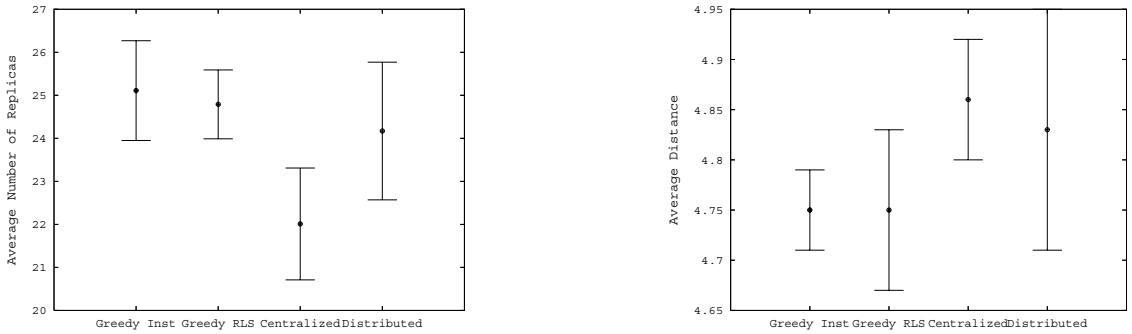

Figure 12. Simulation Results five contents and $d_{\max} = 6$.