Intra-cloud Lightning: Building CDNs in the Cloud

Fangfei Chen*, Katherine Guo†, John Lin†, Thomas La Porta*

*Department of Computer Science and Engineering, The Pennsylvania State University
† Bell Laboratories, Alcatel-Lucent

Abstract—Content distribution networks (CDNs) using storage clouds have recently started to emerge. Compared to traditional CDNs, storage cloud-based CDNs have the advantage of cost effectively offering hosting services to Web content providers without owning infrastructure. However, existing work on replica placement in CDNs does not readily apply in the cloud.

In this paper, we investigated the joint problem of building distribution paths and placing Web server replicas in cloud CDNs to minimize the cost incurred on the CDN providers while satisfying QoS requirements for user requests. We formulate the cost optimization problem with accurate cost models and QoS requirements and show that the monthly cost can be as low as 2.62 US Dollars for a small Web site. We develop a suite of offline, online-static and online-dynamic heuristic algorithms that take user location and request rates. We then evaluate the heuristics via Web trace-based simulation, and show that our heuristics behave very close to optimal under various network conditions.

I. INTRODUCTION

Traditional Content Distribution Networks (CDNs) such as Akamai and Mirror Image have deployed tens of thousands of data centers and edge servers to deliver content across the globe. It has become financially prohibitive for smaller providers to compete on a large scale following the traditional model by building new data centers.

The recent emergence of storage cloud providers such as Amazon S3, Nirvanix and Rackspace has opened up new opportunities to provision cost-effective CDNs. Storage cloud providers operate data centers that can offer Internet-based content storage and delivery capabilities with the assurance of service uptime and end user perceived service quality. Service quality is typically in the form of bandwidth and response time guarantees [1].

Two opportunities arise from the emergence of storage cloud services. First, one can build a CDN serving others without the high cost of owning or operating geographically dispersed data centers. MetaCDN [2] is an example falling into this category. Second, small Web sites can build their own global CDN by simply becoming customers of multiple storage cloud providers operating in different continents and locales.

We refer to this new breed of CDN that is based on storage clouds as a “cloud CDN”, as opposed to a “traditional CDN”. Think replica sites as charged centers within a cloud, then choosing a subset of the sites and connecting them with distribution paths are like the intra-cloud lightning phenomenon.

Storage cloud providers charge their customers by their storage and bandwidth usage following the utility computing model [3]. Storage cost is measured per GB per unit time and bandwidth cost is measured per GB transferred. Bandwidth cost further consists of two components: upload cost for incoming data (e.g., provisioning of Web content) and download cost for outgoing data (e.g., serving user requests).

As the customer, a cloud CDN may take advantage of the competitive prices offered by different cloud providers and reduce its own expense. Combined with the on-demand scaling feature of the cloud, cloud CDNs can easily adjust their storage and bandwidth usage based on demand and possibly trade decreased service quality for reduction in costs. A cloud CDN also multiplexes resources among multiple customers/Web sites like a traditional CDN. In other words, cloud CDNs can provide similar functionality as traditional CDNs, but without actually owning any infrastructure.

However, as the only cost for cloud CDNs is the bandwidth and storage cost, they are very sensitive to the usage variations. To efficiently operate a cloud CDN, intelligent replica placement and user redirection strategies are required. The “Replica Placement” problem in the traditional CDN setting has been widely studied in the literature. However, existing results cannot be readily applied to the cloud CDN setting for the following reasons:

1) Many works in traditional CDNs assume that the network topology is given, such as a tree rooted at the origin server. However, in cloud environment, we have the freedom to build any topology among all the potential replica sites. This topology may be different from the underlying network topology. Thus, replica placement in cloud CDN is a joint problem of building distribution paths and replication.

2) The cost of provisioning a replica site \(v\) from another site \(u\) has been considered as a distance \(d(u, v)\) between \(u\) and \(v\). For traditional CDN, usually the edges are undirected, meaning \(d(u, v) = d(v, u)\). However, storage clouds charge different prices for uploading and downloading, which requires the edge to be directed. This implies that only choosing a set of replica sites is not good enough; we need to specify the replication directions.

The content of the Web site is maintained on an origin server. The cloud CDN rents a set of storage cloud sites where replicas are placed. As today’s Web sites consist of segments (text, image or video) with different popularities and geographic affinities, which may require different SLAs, we consider each segment individually and solve the problem of replicating a segment of a web site to cloud sites.

Request redirection is an integral part of replica placement. Any replica placement algorithm requires the specification of the corresponding request redirection strategy. Once replicas
A considerable amount of research has been done for replica placement in CDNs. The cost model has evolved to include one or more of the three types of costs: retrieval (or download), storage and update (or upload) cost.

In terms of minimizing content retrieval cost only, Li et al. [5] and Krishnan et al. [6] showed that replica placement in general network topologies is NP-complete and provided optimal solutions for tree topologies. Qiu et al. [7] evaluated a number of heuristics and found a greedy algorithm offering the best performance. Radoslav et al. [8] and Jamin et al. [9] proposed a fan-out based heuristic in which replicas are placed at nodes with the highest fan-out. In these proposals, however, once a set of replica sites is chosen, the distribution paths are more or less implied.

In addition to retrieval cost, Xu et al. [10] and Jia et al. [11] further added update cost, whereas Cidon et al. [12] added storage cost into consideration. Furthermore, Kalpakis et al. [13] comprehensively considered all three costs (retrieval, update and storage) and offered solutions for a tree topology only. However, none of the work studied the case in which provisioning cost between replica sites is relevant to the replication direction.

Enhancing results from the above studies, a rich body of work has added QoS into consideration by requiring all user requests to reach replica servers within a certain network distance. Tang et al. [14] and Wang et al. [15] proposed algorithms to optimize total storage and update cost. They used the assumption that requests can be issued from any node, and ignored retrieval cost. Rodolakis et al. [16] added server capacity limitation to the formulation while optimizing storage and retrieval cost. Contrary to existing solutions, our schemes are in place, user requests can be redirected from the origin server to the appropriate replicas using any of the approaches in a traditional CDN setting, such as URL-rewriting, DNS-based request redirection, or transparent interception of user requests [4].

We show an example scenario of replica placement in a cloud-based CDN in Figure 1. In an overlay network shown in Figure 1(a), potential replica distribution paths are drawn as bold lines among an origin server $C_0$ and the potential replica sites $C_1$-$C_4$. We assume each site has a routing path to each user $U_k$\(^1\). However, only a subset of these paths satisfy QoS requirements for user requests, which are drawn as dashed lines. An example solution of the replica placement problem is shown in Figure 1(b), where $C_3$ is chosen to serve requests from $U_1$ and $C_4$ is chosen to serve requests from $U_2$ and $U_3$. $C_1$ is chosen to server requests from $U_4$ as well as relay the replica from $C_0$ to $C_3$ and $C_4$.

The total cost for this solution includes storage, upload and download cost at $C_0$, $C_1$, $C_3$ and $C_4$. For a replica site that serves user requests (e.g., $C_3$ or $C_4$), its upload cost is incurred by incoming traffic to provision Web content at the node, and its download cost is incurred by outgoing traffic to serve user requests\(^2\). We treat the origin server $C_0$ as a regular node in the storage cloud. Its upload cost is incurred by provisioning of Web content, and its download cost is incurred by traffic from $C_0$ to $C_1$. For a replica site that relays replicas to other sites, for example $C_1$, its download cost is also incurred by traffic to provision other replicas. Storage cost is incurred by storing the entire replica, including the cost at origin server $C_0$ and all provisioned replica sites, $C_1$, $C_3$ and $C_4$.

In this paper, we propose a suite of algorithms for replica placement for a cloud CDN. A cloud CDN is built particularly for a Web site (or a Web site segment, in the case that each segment has a different SLA), whose owner signs a SLA with the cloud CDN. In the SLA, certain QoS requirements are specified, such as $X\%$ of end user requests will incur response time of less than a certain value $Y$ from a certain region. The goal of the cloud CDN is to minimize its cost for hosting the Web site while honoring the SLA. Via trace-based study, we show that cloud CDN significantly reduces the monthly cost of providing CDN service to small Web sites to as low as 2.62 US Dollar compared to the 99 US Dollar minimum charge by a traditional CDN.

We first present an Integer Programming (IP) formulation of this problem and prove that it is NP-hard. We then propose two sets of heuristics to place replicas: (1) offline and static algorithms based on past user request patterns and (2) online-static and online-dynamic algorithms triggered by each request. We evaluate the performance of our heuristics via Web trace-based simulation. To the best of our knowledge, this is the first study into the replica placement and distribution path construction problem in cloud CDN settings.

The rest of this paper is organized as follows. Section II presents related work. Section III formally defines the problem. Section IV presents two heuristic algorithms for the offline settings while Section V proposes three heuristics for the online setting. All algorithms are then evaluated in Section VI. Finally, Section VII concludes the paper.

II. RELATED WORK

A considerable amount of research has been done for replica placement in CDNs. The cost model has evolved to include one or more of the three types of costs: retrieval (or download), storage and update (or upload) cost.

1\(^{\text{It is well known that the AS routing path is not necessarily the shortest path.}}\)

2\(^{\text{We ignore bandwidth cost associated with Web requests since its volume is negligible compared to that of the corresponding replies. As video traffic begins to dominate Web traffic, request traffic volume is becoming even less compared to reply traffic volume.}}\)
also consider the online case in which we allow occasional violations of QoS allowed by the SLA.

All the solutions described above are static in nature in that they either assume requests originate uniformly from all nodes or they simply use past request patterns to make replica placement decisions. Bartolini et al. [17] took a different approach by modeling replica placement as a Markovian decision process, and proposed a centralized heuristic. Presti et al. [18] further developed a distributed heuristic. Vicari et al. [19] optimized replica placement and traffic redirection in order to balance request load on replicas. Loukopoulos et al. [20] solved the problem of transferring one set of replica placement into another. In addition to CDN, the replica placement problem has also been studied in Overlay [15], Grid [21] and P2P settings [22].

MetaCDN by Broberg et al. [2] is a low cost CDN using storage cloud resources. The system provides mechanisms to place content in different storage cloud provider networks and redirect user requests to appropriate replicas. However no replica placement and request redirection algorithm is given. Our algorithms can be readily used by MetaCDN to make the system comprehensive.

### III. Problem Formulation

In this section, we first define different problem settings and then formulate the offline problem as an Integer Program. In the end, we define two basic operations used in our heuristics.

#### A. Problem settings

We study the replica placement problem in both offline and online settings. In the offline setting, all input parameters are known, including user request patterns. Algorithms for this setting do not react to individual user requests, but rather aggregate all requests from one user into one request.

In the online setting, we assume all input parameters to replica placement are known except future requests. This implies that we can only consider requests in real time and our decision should be made based on each request.

An algorithm can be either static or dynamic. We define a static algorithm as one that computes the set of replica sites based on current knowledge: past requests for online settings and future requests for offline settings. Then regardless of user requests, we maintain the set of provisioned replica sites. We define a dynamic algorithm as one that may open new sites based on current knowledge: past requests for online settings. Then regardless of user requests, we maintain the set of provisioned replica sites. We then formulate the offline problem as an Integer Program. In the offline setting, all input parameters are known, including user request patterns. Algorithms for this setting do not react to individual user requests, but rather aggregate all requests from one user into one request.

In the online setting, we assume all input parameters to replica placement are known except future requests. This implies that we can only consider requests in real time and our decision should be made based on each request.

An algorithm can be either static or dynamic. We define a static algorithm as one that computes the set of replica sites based on current knowledge: past requests for online settings and future requests for offline settings. Then regardless of user requests, we maintain the set of provisioned replica sites. We define a dynamic algorithm as one that may open new sites based on current knowledge: past requests for online settings. Then regardless of user requests, we maintain the set of provisioned replica sites. We then formulate the offline problem as an Integer Program. In the offline setting, all input parameters are known, including user request patterns. Algorithms for this setting do not react to individual user requests, but rather aggregate all requests from one user into one request.

In the online setting, we assume all input parameters to replica placement are known except future requests. This implies that we can only consider requests in real time and our decision should be made based on each request.

An algorithm can be either static or dynamic. We define a static algorithm as one that computes the set of replica sites based on current knowledge: past requests for online settings and future requests for offline settings. Then regardless of user requests, we maintain the set of provisioned replica sites. We define a dynamic algorithm as one that may open new sites based on current knowledge: past requests for online settings. Then regardless of user requests, we maintain the set of provisioned replica sites. We then formulate the offline problem as an Integer Program. In the offline setting, all input parameters are known, including user request patterns. Algorithms for this setting do not react to individual user requests, but rather aggregate all requests from one user into one request.

<table>
<thead>
<tr>
<th>Problem Setting</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline</td>
<td>Aggregate requests, no new sites</td>
</tr>
<tr>
<td>Online</td>
<td>Individual request in real time, no new sites</td>
</tr>
<tr>
<td></td>
<td>Dynamic</td>
</tr>
<tr>
<td></td>
<td>Individual request in real time, allow new sites</td>
</tr>
</tbody>
</table>

#### B. IP formulation

Now we formulate the offline problem of replicating a Web site replica/segment hosted by an origin server $C_0$ at multiple storage sites to serve all end user requests. In reality, a Cloud provider owns and operates multiple storage sites or data centers; however, each site belongs to only one provider. Cloud sites from different providers may be co-located, but they may offer different prices as well. We assume there are $m$ end users $U = \{U_1, U_2, ..., U_m\}$ indexed by $k$, and $n$ storage cloud sites $C = \{C_1, C_2, ..., C_n\}$ indexed by $i$ or $j$.

We assume the size of the replica is $W$ bytes. As mentioned in Section I, a storage cloud site will charge for storage, incoming and outgoing traffic. For storing the replica, the site $C_j$ will charge unit storage cost of $S_j$ per GB; for uploading the replica onto $C_j$, $C_j$ will charge a unit price of $P_j$ per GB for incoming traffic; for distributing the replica to other sites or delivering content to end users, $C_j$ will charge unit price of $D_j$ per GB for outgoing traffic. Let $V_{uv}$ denote the replication cost from $u$ to $v$. Depending on the nature of the node $v$, $V_{uv}$ has two different meanings. In the first case, $v$ is a cloud site, $V_{uv}$ is the site opening cost also known as the cost of provisioning a site $v = C_j$ by downloading the replica from any site $u = C_i$ that has the replica. The origin server will update the replica from time to time. On average, we assume that a fraction $F \cdot W$ of the content needs to be updated per unit time, where $F$ indicates the update frequency. Then $V_{uv} = V_{ij} = (S_j + P_j F + D_j F) W$.

We generally assume $V_{ij} > 0$. However, the first site having the replica is the origin server. If we assume download price on the origin server is free, that is, $D_0 = 0$, every site would download the replica from the origin server; otherwise, in more general cases where download price per GB is not negligible at the origin server, the replica distribution paths will be a tree structure rooted at $C_0$.

In the second case, $v$ in $V_{uv}$ indicates a user, say $v = U_k$, $V_{uv}$ is the access cost of user $U_k$ who is assigned to site $u = C_j$. We assume that user requests are given in the offline problem. An end user $U_k$ has a request of size $w_k$ bytes and then $V_{uv} = V_{jk} = w_k D_j$. Note that $w_k$ may be less than $W$, because of fractional requests of the whole replica, (e.g. browsing only a portion of the replica/Web site segment) or use of a Web cache. It may be greater than or equal to $W$, because of repeated requests without caching.

We use $L_{jk}$ to denote the routing distance from site $C_j$ to end user $U_k$. Note that this distance captures communication quality between two nodes, and it can be in the form of either hop count or delay. In order to satisfy QoS requirements for end user requests, we need to limit this distance within a required QoS distance $Q$. We do not put bandwidth capacity as a hard constraint on replica sites. Instead, we restrict $L_{jk}$ to be bounded by a smaller value of QoS distance $Q$. This is based on the assumption that web traffic congestion has
Definition

V
User
L
Term
Cloud site
Origin server
Storage cost of
QoS distance
W
Request size of
Upload cost of
QoS

The notations used in this paper are summarized in Table II.

IV. OFFLINE ALGORITHMS

In this section, we present two heuristic algorithms Greedy Site and Greedy User for the offline setting.

A. Greedy Site (GS)

The Greedy Site (GS) algorithm is adopted from an approximation algorithm for the Set Covering Problem [26]. We name it Greedy Site (Algorithm 1) because we iteratively decide to open a closed site which has the maximum utility and assign all its potential users to it. The site utility is defined as the total request volume (in bytes requested) divided by the total cost incurred by serving these requests. The potential users of a site are the users who are within the QoS distance of this site but not yet assigned. We open this site, then find the next best site to open until all users are assigned to a site.

B. Greedy User (GU)

We name the second algorithm Greedy User (GU, Algorithm 2) because users rather than sites are evaluated one after another. In the algorithm, we assign users to sites with the lowest cost and open new sites if necessary. We consider users with the least number of potential sites first (line 2 in Algorithm 2). This is because the more sites we can assign a user to, the more likely we can assign the user to a site already

C. Basic operations

Before we look into the heuristics for various settings, we need to define two basic operations: Replica Provision and User Redirection.

As traffic load approaches network and server capacity in the cloud provider network, delay is bound to increase. In future work, we plan to add link capacity and site capacity constraints into the formulation.

limited impact on storage clouds; one of the key benefits of cloud storage providers like Amazon is their ability to add capacity on demand quickly when replica sites become congested. When such congestion happens, $L_{jk}$ increases but is still bounded by $Q$.

The objective of the replica placement problem is to find a replication strategy minimizing the total cost by provisioning a subset of the cloud sites such that each end user is assigned to at least one site and the QoS requirements for end user requests are satisfied. Formally, let $G = (N, E)$ be a directed graph, where $N = \{C_0\} \cup C \cup U$. Edge $(C_i, C_j) \in E$ indicates a feasible provisioning or replication path from $C_i$ to $C_j$. Edge $(C_j, U_k) \in E$ indicates a potential assignment of user $U_k$ to $C_j$; in other words, for any $(C_j, U_k) \in E$, $L_{jk} \leq Q$. Each edge $(u, v) \in E$ is associated with a cost $V_{uv}$. The goal is to find a subtree that contains a directed path between $C_0$ and every $U_k$. The problem is formally defined as:

$$\min \sum_{(u, v) \in E} y_{uv}V_{uv}$$

subject to

$$\sum_{w \in N} x^k_{uw} - \sum_{v \in N} x^k_{vu} = \begin{cases} 1, & u = C_0 \\ -1, & u = U_k \\ 0, & \text{otherwise} \end{cases} \forall U_k$$

$$x^k_{uv} \leq y_{uv} \forall (u, v) \in E$$

where $y_{uv}$ is a binary variable indicating whether a replication path $(u, v)$ is chosen. And variable $x^k_{uv}$ is the number of replication flows between $C_0$ and $U_k$ via $(u, v)$.

Constraints (2a) guarantees that there is a directed path from $C_0$ to every $U_k$. Constraints (2b) indicates that a flow is allowed only if its replication path is chosen in the solution.

This problem is essentially an instance of the Directed Steiner Tree Problem. We present an NP-hard proof of this problem in a technical report [23] and show that the problem cannot be approximated with factor better than $O(\log(m))$, where $m$ is the number of users. Charika et al. [24] proposed an $O(m^{\epsilon})$-approximation algorithm for the general directed Steiner Tree problem, for any $\epsilon > 0$. For a large $m$, this factor can be very high. Instead of seeking good approximations, our interest is in efficient heuristics that perform well in practice.

We implement the IP formulation (1) and solve it with CPLEX [25]. The typical running time for the solver ranges from minutes to hours which is too long to be practical. This also motivates the need for heuristics. However, for a reasonably sized problem instance, we still solve it optimally to obtain a lower bound to evaluate our algorithms.

The notations used in this paper are summarized in Table II.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_k$</td>
<td>User $k$</td>
<td>$w_k$</td>
<td>Request size of $U_k$</td>
</tr>
<tr>
<td>$C_i$</td>
<td>Cloud site $i$</td>
<td>$W$</td>
<td>Replica size</td>
</tr>
<tr>
<td>$C_j$</td>
<td>Origin server</td>
<td>$V_{ij}$</td>
<td>Replication cost $i \rightarrow j$</td>
</tr>
<tr>
<td>$S_{ij}$</td>
<td>Storage cost of $C_j$</td>
<td>$L_{ij}$</td>
<td>Distance from $C_j$ to $U_k$</td>
</tr>
<tr>
<td>$D_{ij}$</td>
<td>Download cost of $C_j$</td>
<td>$Q$</td>
<td>QoS distance</td>
</tr>
<tr>
<td>$F_{ij}$</td>
<td>Upload cost of $C_j$</td>
<td>$E$</td>
<td>Update Frequency</td>
</tr>
</tbody>
</table>

The Replica Provision operation for a cloud site $C_j$ includes downloading a replica from a proper site $C_i$ and uploading and storing the replica on $C_j$. We also refer to this operation as open site. The first step is to locate the source site $C_i$. A greedy strategy is to download from one of the currently opened sites that has the lowest download price per GB. Let $O_j$ denote the opening cost of $C_j$. Then

$$O_j = \begin{cases} 0 & \text{if } C_j \text{ is OPEN} \\ \min_{i \in \{C_i \text{ is OPEN}\}} V_{ij} & \text{otherwise} \end{cases}$$

Note that the order in which replica sites are opened affects the cost for downloading replica to a site, because the download source can only be selected from the currently opened sites.

The second operation is User Redirection or User Assignment. In the offline setting, all requests of a user are directed to the best replica site. In the online setting, each user request is redirected separately. In practice, if a site is overloaded, attacked or fails, we remove the site and either rerun the offline algorithms or continue with iterations of the online algorithms. As a result, requests from the same user are redirected to multiple sites over time. Note that it is common practice in traditional CDNs today to redirect user requests to different sites upon congestion or failure of the current site the user is directed to.
In the online setting, one can use results from past request patterns to make replica placement decisions using offline algorithms, and only redirect user requests to the already opened sites. As shown in Section VI, Greedy Site (GS) performs better than Greedy User (GU), therefore we select Greedy Site (GS) to use in our online algorithms.

Based on different combinations of Greedy Request (GR) and Greedy Site (GS), we have the following three online schemes: Greedy Request Only (GRO), Greedy Request with Preallocation (GRP), and Greedy Site Only (GSO).

Algorithm 1 Greedy Site (GS)

1: \( E \) is the set of users who have not been assigned.
2: \( E_j \) is the current set of users who can be assigned to \( C_j \).
3: while \( E \neq \emptyset \) do
4: \( W_j \leftarrow \sum w_k, U_k \in E_j \).
5: \( j^* \leftarrow \arg\max_{j \in \{j | C_j \text{ closed}\}} \frac{W_j}{D_j} \).
6: Assign all users in \( E_j \) to \( C_{j^*} \).
7: Open \( C_{j^*} \).
8: \( E \leftarrow E - E_j \).

V. ONLINE ALGORITHMS

In this section, we present three online heuristics. We first present the basic algorithm Greedy Request (GR, Algorithm 3) where each request is processed upon arrival. For the first request of a user, we select the lowest cost potential site for the user based on the total cost for serving this request, opening the site if necessary. Any further requests from the same user will be redirected to the assigned site for the user. It is similar to Greedy User, except the order to evaluate users follows the arrival order for the first request of each user, and only the volume for the first request is considered for each user in making replica provision and user redirection decisions. Note that Greedy Request (GR) works in both offline and online settings.

Algorithm 3 Greedy Request (GR)

1: for all request \( q \) in increasing arrival time do
2: Request \( q \) is from \( U_k \) and of size \( w_q \).
3: if \( U_k \) is assigned to \( C_{j^*} \) then
4: Redirect \( q \) to \( C_{j^*} \).
5: else
6: \( F_k \) is the set of sites that \( U_k \) can be assigned to.
7: \( j^* \leftarrow \arg\min_{j \in \{j | C_j \text{ closed}\}} (O_j + w_q D_j) \).
8: Redirect \( q \) to \( C_{j^*} \).
9: if \( C_{j^*} \) is CLOSED then
10: Open \( C_{j^*} \).

VI. PERFORMANCE EVALUATION

In this section, we evaluate our heuristics via Web trace-based simulation.

A. Potential replica sites, request pattern and cost

The information on potential replica sites is extracted from the iPlane Internet PoP topology [27], which provides PoP-node-to-IP mappings and inferred links between PoP nodes with latency information. For each problem instance we evaluate, we randomly pick one PoP node as the origin server \( C_0 \) and a subset of PoP nodes as cloud sites.
We then place the group of chosen PoP nodes including the origin server and the cloud sites onto a metric field using the geographical information of each node. To place a node, we first look up its IP address in the GeoLite city database [28] and convert the IP address to its corresponding (latitude, longitude) pair. However, a node may be associated with multiple IP addresses. In this case, we estimate its location with a technique similar to that proposed in [29]. Refer to [23] for details of this technique.

We vary the number of cloud sites from 20 to 48. We choose such a small number because it reflects the fact that there are few cloud providers today. However, simulations with 200-300 cloud sites (results not shown here) confirm that the performance with a large number of sites is similar to that with a small number.

We extract end user request patterns from Penn State CSE Web access traces\(^5\), which includes the host name and IP address information between Oct 1st and Dec 31st in 2010. In the 92-day trace, there are 1,186,213 user requests with an average request size of 25 KB and 268 MB requests per day. We sum all the bytes for distinctive Web page URLs to derive the default Web site replica size of 175 MB. By default, we run simulations for 30 days using traces from Nov. 1-30, 2010; we always use Nov. 1st as day one when we vary the trace period in simulations. The 30-day trace from Oct. 2-31st is used for preallocation in online settings.

Using end user IPs from web server access traces, we map end users onto the same metric field with the origin server \(C_0\) and the cloud sites. Then we assign distance value \(L_{jk}\) from cloud sites to users. Padmanabhan et al. [30] contend that in recent years there is significant correlation between network delay and geographic distance. Because we do not have actual hop count information between users and cloud sites, we use geographic distance as an indicator of delay. In fact, the choice of distance metric does not impact the performance of our algorithms; any distance metric that is capable of describing the QoS requirement is applicable. In our simulation, we set the default QoS distance to be 5 (about 350 miles converted to geographical distance). A value too small does not guarantee that every user has at least one feasible site while a value too large makes every site fall within the QoS range which essentially renders the QoS constraint ineffective.

To summarize, in our simulations, the origin server, cloud sites and end users are located in the same metric field. The distance \(L_{jk}\) from \(C_j\) to \(U_k\) is the geographic distance.

Typical storage cloud costs in 2010 are shown in Table III. We can see that for today’s cloud provider sites, unit price for storage, upload and download is comparable. As we need to evaluate more than four cloud sites, we use these typical costs as a guideline and randomly group them to generate new price combinations for different cloud sites. For example, we may use incoming cost of Nirvanix USA/EU, outgoing cost of Amazon S3 USA and storage cost of Amazon S3 EU as the price for one cloud site in our simulation.

\(^5\)http://www.cse.psu.edu/

We vary one parameter at a time while fixing the rest at their default values. This results in 40 parameter settings. For each parameter setting, we generate 50 problem instances. This results in \(50 \times 40 = 2000\) total problem instances. For each instance, we first randomly select from the iPlane topology, a subset of the nodes to be cloud sites and the origin server; we then assign a random combination of prices to each site. The length of trace period determines the storage cost, which is the monthly cost for any period up to 31 days and twice the monthly cost for any period from 32 to 61 days.

For each problem instance, we obtain the optimal cost by solving it using the CPLEX solver [25]. Figure 2 shows a CDF of the optimal costs in US Dollars in all tests. As we can see, 80 percent of the tests incurred costs less than 2.62 US Dollars, and almost all costs are below 60 Dollars. This is much less than the pricing of a traditional CDN\(^6\), whose minimum price per month is 99 Dollars\(^7\). Traditional CDNs may offer a lower per GB price for large Web sites with high volume of traffic. However, we argue that cloud CDN is more flexible and mainly focuses on benefiting small Web sites such as the one in our tests.

\(^7\)Other major CDN providers such as Akamai do not publish their prices, but they generally require similar or even higher minimum monthly fee.

![Fig. 2: CDF of the Optimal Costs in All Tests](image-url)
achieve relative cost of 1.5 most of the time. GS is slightly better than GU which is better than GR. The 95th percentile value of the relative cost is 1.14, 1.20 and 1.91 respectively.

We plot an error bar in Figure 3(b) using the max, min and median of relative cost for each algorithm. GR has the highest worst and median value for relative cost and GS is the most stable with the steepest CDF curve (Figure 3(a)) and the lowest median cost value of 1.05. We recommend GS as the best offline algorithm, and use it to analyze past traces in order to conduct site preallocation for online algorithms.

Figure 4 demonstrates the effect of different parameters on the performance of online algorithms. To better understand the results, we look into the total cost structure.

Let $X^{OPT}W$ denote the optimal site opening cost and $\Delta XW$ denote the additional site opening incurred by an algorithm. Similarly, let $\sum_k Y_k^{OPT} w_k$ denote the optimal user access cost and $\sum_k \Delta Y_k w_k$ denote the additional user access cost. Then the relative cost plotted in the figures is:

$$\text{relative cost} = \frac{(X^{OPT} + \Delta X)W + \sum_k Y_k^{OPT} w_k + \sum_k \Delta Y_k w_k}{X^{OPT}W + \sum_k Y_k^{OPT} w_k}.$$

In one extreme case, when $W \gg \sum k w_k$,

$$\text{relative cost} \approx \frac{X^{OPT} + \Delta X}{X^{OPT}} = 1 + \frac{\Delta X}{X^{OPT}},$$

only the additional cost for site opening is observable. In the other extreme case, when $W \ll \sum k w_k$,

$$\text{relative cost} \approx \frac{\sum_k (Y_k^{OPT} + \Delta Y_k) w_k}{\sum_k Y_k^{OPT} w_k} = 1 + \frac{\sum_k \Delta Y_k w_k}{\sum_k Y_k^{OPT} w_k},$$

only the additional cost for user access is observable. In all other cases, the relative sizes of $W$ and $\sum k w_k$ decides which part of non-optimality (for either site opening or user access) is dominating in the total relative cost.

When $W \ll \sum k w_k$, the two offline algorithms achieve near-optimal performance (Figure 4(a), relative cost $\approx 1$), which indicates that $\sum_k \Delta Y_{k}^{GS, GU} \approx 0$. But GR incurs even higher relative cost when $W \ll \sum k w_k$ than it does when $W \gg \sum k w_k$ (Figure 4(a)). This indicates that

$$\frac{\sum_k (\Delta Y_{k}^{GR}) w_k}{\sum_k Y_k^{OPT} w_k} > \frac{\Delta X^{GR}}{X^{OPT}}.$$

Then we observed that, when $W \gg \sum k w_k$, GS and GU have smaller relative cost than GR, which means

$$\frac{\Delta X^{GR}}{X^{OPT}} < \frac{\Delta X^{GS, GU}}{X^{OPT}}.$$

In other words, in Figure 4(a) we observed that GS and GU incur near-optimal user access cost and less site opening cost than GR; while GR produces non-optimal cost in both two parts and the user access part weighs more in its non-optimality.

We then increase the trace period from 20 to 56 days, which has the same effect as increasing the user request size. As shown in Figure 4(b), relative costs for GS and GU decrease slightly while relative cost for GR increases slightly. We also vary update frequency and observe similar results to the one shown in Figure 4(a), for increasing frequency has the same effect as increasing replica size.

In the third test shown in Figure 4(c), we vary QoS distance from 5 to 14; in the forth test in Figure 4(d), we vary the number of sites from 20 to 48. These two tests illustrate an important factor that affects performance: the size of the solution space. When we increase the QoS distance, or increase the total number of sites, a user has more choices for site selection which increases the size of the solution space. A larger solution space makes the problem more difficult for our heuristics and therefore their relative cost increases. However, the performance of GS and GU vary only slightly within reasonable range of parameter values.

In summary, GU and GS both achieve near-optimal solutions in various problem instances. Instead of aggregating all requests from each user into one as in GU and GS, GR processes requests in the order they arrive and assigns users to a site solely based on its first request. This assignment order results in both more sites being opened and users being assigned to sites with higher download cost, which leads to higher relative cost than the other two heuristics.

### C. Performance of Online Algorithms

For the two heuristics requiring site preallocation, namely, Greedy Request with Preallocation (GRP) and Greedy Site Only (GSO), we first allocate sites based on prediction of user request patterns. We use the immediate past history as an indication of future user behavior. Specifically, when we simulate with a trace period of $T$ days starting on Nov 1st, 2010, we use the trace from the past $T$ days ending on Oct 31st, 2010; when $T$ is larger than 30 days, we only use the 30-day trace ending on Oct 31st, 2010 for preallocation. Therefore, we set the storage cost to the monthly cost during the preallocation process.

For every test, we report QoS violation ratio, which is the ratio of request volume (in bytes) that experiences QoS violation to the total request volume. In practice, the QoS violation clause of a SLA can also be written in different forms. One example is to measure the percentage of user requests that violate QoS requirements. Another example is to measure the percentage of users that suffer from QoS violation. The metric we select, request volume violation, puts more weight on large requests.

Rather than reporting relative cost, we introduce a new metric, normalized relative cost, to evaluate online algorithms. It is defined as:

$$\text{normalized relative cost} = \frac{\text{relative cost}}{1 - \text{QoS violation ratio}}.$$
We introduce this metric in order to add penalty for each request resulting in a QoS violation. It combines relative cost and QoS violation ratio into one metric. However, for a reasonably small QoS violation ratio, relative cost and normalized relative cost result in very similar value.

The three online schemes in decreasing performance or increasing cost order are GSO, GRP and GRO. The 95th percentile value of the normalized relative cost is 1.15, 1.80 and 1.91 respectively, as shown in Figure 6(a).

Out of the three heuristics, the static GSO has the lowest median cost very close to 1 (Figure 6(b)). It even results in lower cost than the offline optimal in 27% of the problem instances (Figure 6(a)). This is because GSO allows a user to be assigned to a site with a low downloading price while violating QoS constraints when no site in the preallocated set can meet the constraints. On the other hand, the offline optimal requires all QoS constraints to be satisfied, therefore it requires more sites to be opened than the solution by GSO.

Figure 7 shows statistics of QoS violation ratio of the three online heuristics. GSO has a much higher QoS violation ratio (median value around 0.7e-4), than the dynamic schemes (1.5e-5 for GRO and 2.4e-6 for GRP). GSO is the worst since no new sites can be opened, user requests are directed to preallocated sites even with QoS violations. Clearly GSO trades lower cost with lower user QoS satisfaction. GRO is better as only the first request from a new user triggers QoS violations if a new site has to be opened for this user. GRP is the best in the sense that it eliminates the instances of QoS violations from GRO when the first request from a new user is redirected to a site already opened by the preallocation process. Furthermore, in GSO, users who experience QoS violations are permanently assigned to the sites that violate their QoS requirements. This causes unfairness among users and may be prohibited by some SLAs.

Nevertheless, the results indicate that past request trace offers a good prediction for future request pattern of the same length in time. In addition, as we use the best offline algorithm Greedy Site (GS) to select sites for preallocation in GRP, GRP has a slight performance advantage over GRO.

It is interesting to see in Figure 5 that cost for the static GSO varies differently than the dynamic heuristics (GRO and GRP). In fact, GSO should perform similarly as GS does in offline settings, as it runs GS on past user requests; the other two should perform similarly as GR does, because they both run GR dynamically, regardless of whether they use preallocation or not.

Besides the performance similarity between each heuristic and the offline algorithm it is based on, we have other
observations as follows. Static allocation does not incur any cost for opening new sites after the preallocation phase. As a result, the longer time we run a test, the more advantage GSO has by assigning QoS violated requests to sites already open; thus we see a decrease in the normalized relative cost when we increase the trace period (Figure 5(b)). With increasing QoS distance (Figure 5(c)), the problem space becomes larger and therefore the relative cost is supposed to increase. However, the cost of GSO increases slightly and then decreases. The reason is that GSO can cover more user requests in the preallocation phase with a larger QoS value. In other words, a large portion of the online problem is solved by an offline heuristic. In an extreme case that the QoS value is large enough so that every site covers all users, GSO will have similar performance as the offline GS heuristic.

To summarize, GRP incurs the smallest QoS violation while achieving good performance in terms of cost. GRO incurs slightly more violations but has the advantage of running without past traces. GSO trades a much higher violation ratio for lower cost and may cause unfairness among users. We suggest the following guideline for using online schemes: use Greedy Request with Preallocation (GRP) as long as there is some past request pattern; otherwise use Greedy Request Only (GRO) which does not rely on the past request pattern. Only when QoS violation is not a large concern should we choose Greedy Site Only (GSO).

VII. CONCLUSION AND FUTURE WORK

In this paper, we investigated the problem of placing Web server replicas in storage cloud-based CDNs along with building distribution paths among them to minimize the cost incurred on the CDN providers while satisfying QoS requirements for user requests. We formulated the problem as an Integer Program and presented various offline and online greedy heuristics. We evaluated their performance against the optimal via Web trace-based simulations.

In the future, we plan to expand our formulation to explicitly consider both bandwidth and storage capacity at a cloud site. In reality, latency can vary because of congestion, network failures, route changes etc., which is not explicitly addressed in this paper. We plan to investigate mechanism to deal with latency variability as well as other aspect of the SLA. For certain web content, for example video, SLA metrics such as throughput is more important.

REFERENCES


