Computing on the Edge: A Platform for Replicating Internet Applications

Michael Rabinovich, Zhen Xiao, and Amit Aggarwal*

AT&T Labs - Research
180 Park Avenue, Florham Park, NJ 07932
{misha, xiao}@research.att.com

Abstract

Content delivery networks (CDNs) improve the scalability of accessing static and, recently, streaming content. However, proxy caching can improve access to these types of content as well. A unique value of CDNs is therefore in improving performance of accesses to dynamic content and other computer applications. We describe an architecture, algorithms, and a preliminary performance study of a CDN for applications (ACDN). Our system includes novel algorithms for automatic redeployment of applications on networked servers as required by changing demand and for distributing client requests among application replicas based on their load and proximity. The system also incorporates a mechanism for keeping application replicas consistent in the presence of developer updates to the content. A prototype of the system has been implemented.

1 Introduction

Content delivery networks (CDNs) have become a popular method for providing scalable access to Web content. They currently provide access to static and streaming content. However, proxy caches can improve the delivery of these content types as well. In particular, as shown in [7], if proxies were deployed ubiquitously, the additional benefit of CDNs in delivering static content would be marginal.

A unique value of CDNs is in delivering dynamic content because this content cannot be cached by proxies. We refer to such a CDN as an Application CDN, or ACDN. An ACDN will allow a content provider (application provider in this case) to not worry about the amount of resources provisioned for its application. Instead, it can deploy the application on a single computer anywhere in the network, and then ACDN will replicate or migrate the application as needed by the observed demand.

One could implement an ACDN using general utility computing systems such as Ejacent [20] and vMatrix [1], which migrate the entire dynamic state of the running application from one server to another. These are complex systems that have to address all process migration issues that have been a subject of many years of research. Our key observation is that, because Web service applications have well-defined boundaries between processing individual requests, and that usually the server that starts processing a request is the one required to complete it, these applications do not have to be migrated at an arbitrary time. We exploit this specificity of our target application class by allowing applications to migrate or replicate only at the request boundaries, when the dynamic state needed to be transferred is minimal. In a sense, instead of *migrating* an application to the new server, we simply *deploy* the application at the new server. Once the new server is up and running, the system can optionally decommission the application at the old server. Automatic deployment of an application is a much simpler task than the migration of a running application used in utility computing. Indeed, the latter is at the same time more fine-grained, in that the migration can occur at any time, and more heavy-weight since the transferred state must include the entire memory footprint of the application at the time of the transfer. We extend a typical approach used by software distribution systems to implement automatic deployment.

Another simplification is that we currently maintain replica consistency only for updates to the application by the content provider. For updates that occur as a result of user accesses, we either assume they can be merged periodically off-line (which is the case for commutative updates such as access logs) or that these updates are done on a shared back-end database and hence they do not violate replica consistency.

This paper presents a system design of an ACDN that

* This author is currently with Microsoft.

uses the above approach and proposes the algorithms for deciding when and where to replicate or migrate an application, and how to distribute incoming requests among available replicas. We also report the results of a preliminary study of the performance of our approach. A functional prototype of our ACDN has been implemented and its demo has been presented at SIGMOD'2002 [11].

2 Issues

An ACDN has a fundamental difference with a traditional CDN that delivers static content. The latter uses caches as CDN servers. Each CDN server is willing to process any request for any content from the subscriber Web site. The CDN will either satisfy the request from its cache or obtain the response from the origin server, send it to the requesting client, and store it in its cache for future use. In contrast, to be able to process a request for an application locally, an ACDN server must possess a deployed application, including executables, underlying data, and the computing environment. Deploying an application at the time of the request is impractical; thus the ACDN must ensure that requests are distributed only among the servers that currently have a replica of the application; at the same time, the applications are redeployed among ACDN servers asynchronously with requests.

Thus, ACDN must provide solutions for the following issues that traditional CDNs do not face:

- Application distribution framework: ACDN needs a mechanism to dynamically deploy an application replica, and to keep the replica consistent. The latter issue is complicated by the fact that an application typically contains multiple components whose versions must be mutually consistent for the application to be able to function properly.

- Content placement algorithm: the system must decide which applications to deploy where and when. Content placement is solved trivially in traditional CDNs by cache replacement algorithms. However, an ACDN must make explicit decisions to replicate or migrate an application because it is too costly to replicate an application, especially at the time of the request.

- Request distribution algorithm: in addition to load and proximity factors that traditional CDNs must consider in their request distribution decisions, the request distribution mechanism in ACDN must be aware of where in the system different applications are currently deployed. Indeed, while a traditional CDN can process a request from any of its caches, an ACDN must assign the request to one of the servers that possess the application.

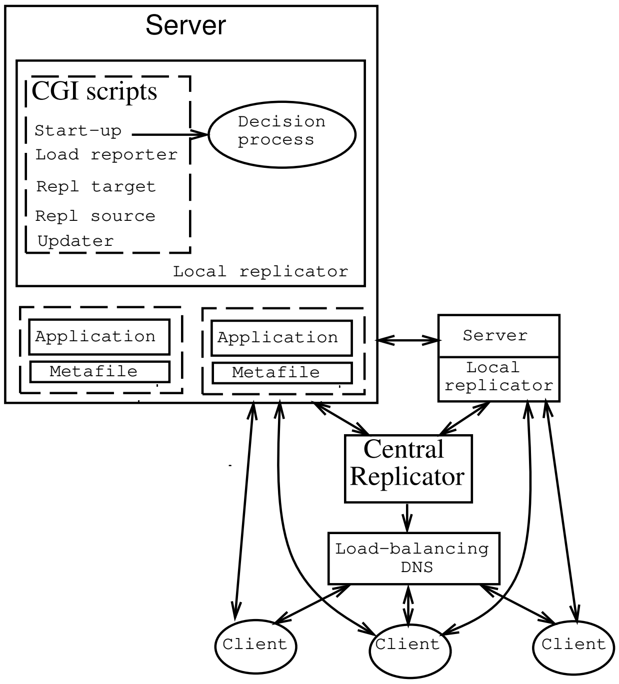


Figure 1: ACDN prototype architecture

- System stability: under steady demand, the system behavior should reach a steady state with respect to request distribution and replica placement. Highly oscillating request distribution necessarily implies periods of highly suboptimal distribution; endless re-deployment of application replicas only consumes bandwidth and adds load on the servers.

- Bandwidth overhead: creating a remote application replica consumes bandwidth for sending the application from the source to the target server. Thus, one has to be cautious and create replicas only when there is a reason to believe that the benefits will overweight this overhead.

In the following sections we describe our solutions to these problems.

3 Architecture Overview

The general architecture is straight-forward and depicted in Figure 1. Its main goal is to rely completely on the HTTP protocol and Web servers without any modifications. Not only does this simplify the adoption of the system, but it also allows easy firewall and NAT traversal and hence a deployment of the system over public Internet.

Each ACDN server is a standard Web server that also contains a *replicator*, which implements ACDN-specific functionality. In our initial design, we assume homogeneous servers to simplify the comparison of their loads. The replicators are implemented as a set of CGI scripts and so are a pure add-on to any standard Web server[1]. There is also a global *central replicator* that mainly keeps track of application replicas in the system. Although the central replicator is theoretically a bottleneck, this is not a concern in practice since the amount of processing it does is minimal; it can in fact be physically co-located with the DNS server. The central replicator is also a single point of failure. However, it is not involved in processing user requests. Thus, its failure only leads to a stop in application migrations or replications and does not affect the processing of requests by the existing replicas. Furthermore, the central replicator only contains soft state that can be reconstructed upon its recovery or replacement.

The server replicator contains the following CGI scripts: the start-up script, the load reporter, the replica target script, the replica source script, and the updater script. The *start-up script* is invoked by the system administrator when (s)he brings a new ACDN server on-line. The script forks a *decision process* that periodically examines every application on the server and decides if any of them must be replicated or deleted. The *load reporter* is a script periodically invoked by the central replicator. The script returns the load of the server, which we measure as an output of a Unix *uptime* command.

The rest of the scripts implement the application distribution framework and are considered below.

4 Application Distribution Framework

Our implementation of the framework is based on the concept of a metafile, inspired by ideas from the area of software distribution such as the technology by Marimba Corp. [8]. The next subsection describes the metafile and the subsequent subsections show how it is used to implement the application distribution framework.

4.1 The Metafile

Conceptually, the metafile consists of two parts: the list of all files comprising the application along with their last-modified dates; and the *initialization script* that the recipient server must run before accepting any requests. Figure 2 provides an example of a metafile[2] that represents a map-drawing application consisting of three files: an executable file that is invoked on access to this application and two data files used by this executable to generate responses. The metafile also contains the initialization script that creates a directory where the application collects usage statistics and sets the corresponding environment variable used by the executable. The initialization script can be an arbitrary shell script. When the initialization script is large, the metafile can include just a URL of the file containing the script.

```
FILE /home/apps/maps/query-engine.cgi 1999.apr.14.08:46:12
FILE /home/apps/maps/map-database 2000.oct.15.13:15:59
FILE /home/apps/maps/user-preferences 2001.jan.30.18:00:05
SCRIPT
   mkdir /home/apps/mapping/access-stats
   setenv ACCESS_DIRECTORY /home/apps/maps/access-stats
ENDSCRIPT
```

Figure 2: An example of a metafile

The metafile is treated as any other static Web page and has its own URL. Using the metafile, the entire application distribution framework can be implemented entirely over standard HTTP. Indeed, the operations that the framework must support include replica creation, replica deletion, and replica consistency control. Migration of an application is accomplished by replica creation followed by the deletion of the original replica. Let us consider these operations in turn.

4.2 Replica Creation

The replicator on each ACDN server contains replica target and replica source CGI scripts. The process of replica creation is initiated by the decision process on an ACDN server with an existing replica (called the source server below) and entails the following steps.

1. If the reason for replication is the overload of the source server, the source server queries the central replicator for the least-loaded server in the system, which will be asked to create a new application replica (referred to as the target server). A subsequent negotiation between the source and target servers prevents a herd effect when many servers attempt to replicate their applications to the the same target server at once. When the reason for replication is improving proximity to demand, the source server

[1] All scripts in our prototype are implemented as FastCGI for scalability. Using servlets in place of CGI scripts is also possible.

[2] Our prototype uses an XML markup for encoding metafiles, which is stripped from the example in Figure 2 for clarity.

identifies the target server locally from its replica usage (see Section 5 for details).

2. The source server invokes the replica target CGI script on the target server, giving the URL of the application metafile as a parameter. This URL also serves as a logical application ID inside the system.

3. The replica target script at the target server invokes the replica source script at the source server, with the metafile URL as a parameter.

4. The source server responds with the tarball of the application, which the replica target script unpacks and installs. The replica target script also executes the initialization script from the metafile.

5. Upon the execution of the initialization script, the replica target script informs the central replicator about the new application replica. The central replicator sends this update to the DNS server, which recomputes its request distribution policy based on the new replica set (see Section 5).

All the above interactions occur over standard HTTP, and require only a few CGI scripts.

4.3 Replica Deletion

Replica deletion is initiated by the decision process on a server with the application replica and involves the following steps.

1. The server sends to the central replicator a request for permission to delete its replica.

2. If this is not the last replica in the system, the central replicator sends the deletion update to the DNS server, which recomputes the request distribution policy that excludes this replica.

3. Once the DNS server confirms the adoption of the new request distribution policy, the central replicator responds to the ACDN server with the permission to delete the replica. The permission response contains the DNS time-to-live (TTL) associated with the domain name of the application.

4. The ACDN server marks the replica as "to be deleted" and actually deletes it after the TTL time provided by the central replicator. This delay is required because residual requests for the application might still arrive due to earlier DNS responses still cached by the clients.

4.4 Consistency Maintenance

In general, an application may change either because of the developer updates, defined as any modification to the application by the application authors or maintainers, or user updates, which occur as a result of user accesses. Developer updates can affect code (i.e., application upgrades) or underlying data (i.e., product pricing), while user updates involve data only (i.e., e-commerce transactions). We so far have implemented a solution to the developer updates, so our current prototype is suitable for applications that are read-only from the user perspective, such as informational sites. For updates that occur as a result of user accesses, we either assume they can be merged periodically off-line (which is the case for commutative updates such as access logs) or that these updates are done on a shared back-end database and hence they do not violate replica consistency.

There are three related issues in handling developer updates: replica divergence, replica staleness, and replica coherency. Replica divergence occurs when several replicas receive conflicting updates independently at the same time. Replica can become stale if it missed some updates. Finally, a replica can become incoherent if it acquired updates to some of its files but not others and so there is version mismatch between individual files.

Our system avoids replica divergence by allowing developer updates only to the *primary* application replica, so that all the updates can be properly serialized. The primary replica is appointed and kept track of by the central replicator. In particular, the central replicator selects a new primary before giving permission to delete the old primary replica.

The metafile provides an effective solution to the replica staleness and coherency problems. With the metafile, whenever some objects in the application change, the application's primary server updates the metafile accordingly. Whenever other Web servers detect that their cached copies of the metafile are not valid, they download the new metafile and then copy all modified objects together as prescribed in the metafile.

Thus, the metafile reduces the application staleness and coherency problems to cache consistency of an individual static page (the metafile). Once a replica detects that its cached metafile is stale it always obtains the coherent new version of the application. Any existing cache consistency mechanism for static pages can be used to maintain cache consistency of the metafile, including various validation and invalidation techniques (see [17], Chapter 10). The only difference is that updating an application must be asynchronous with request arrivals since doing it at the request time may create a prohibitive delay to the user latency. In our current prototype, the central replicator periodically invokes the updater script on all

ACDN servers with a replica of the application; the updater script then validates the local metafile by issuing a GET If-Modified-Since request to the primary server, and acquires necessary updates for the corresponding application. In the meantime, the server keeps using the old version until the new version is ready. The details of the interaction between the server with a stale application and the application's primary server should be straightforward given the description of the replica creation procedure in Section 4.2 and omitted for brevity.

5 Algorithms

Two types of algorithms are inherent in any ACDN: an algorithm for content placement, which decides on the number and location of the application replicas, and an algorithm for request distribution, which chooses a replica for a given user request. We consider these algorithms in the next two subsections.

5.1 Content Placement Algorithm

The algorithm is executed periodically by an ACDN server, which makes a local decision on deleting, replicating, or migrating its applications. Allowing each server to decide autonomously ensures the scalability of our approach. The algorithm for a given application is shown in Figure 3. The algorithm utilizes three parameters, the *deletion threshold*, the *redeployment* threshold, and the *migration* threshold. The deletion threshold D characterizes the lowest demand that still justifies having a replica. In our experiments, it is expressed as the total number of bytes served by the server since the previous run of the of the algorithm[3]. The choice of the deletion threshold depends on the characteristics of the application as well as the underlying system and network.

The redeployment threshold reflects the amount of demand from the vicinity of another server that would warrant the deployment of an application replica at that server. Consider the decision to replicate the application at server i. Let B_i be the total number of bytes the current server served to clients from the vicinity of server i since the previous run of the algorithm. Let A be the total size of the application tarball (possibly compressed), and U be the total size of updates received in the same period. If $B_i/(A+U) > 1$ then the amount of bandwidth that would be saved by serving these requests from the nearby server i would exceed the amount of bandwidth consumed by shipping the application to server i and by keeping the new replica fresh. Hence, the bandwidth overhead

[3] An alternative is to express it as the request rate. The implications and analysis of these two ways to express the deletion threshold are left for future work.

```
DecidePlacement():
/* Executed by server s */
  if load_s > HW, offloading = Yes;
  if load_s < LW, offloading = No;
  for each application app
    if B_total ≤ D
      delete app unless this is the sole replica
    elseif B_i/(A + U) > R AND B_i > D for some server i
      replicate app on server i
    elseif B_i/B_total > M AND B_i/A > R for some server i
      if server i accepts app migrate app to server i
    endif
  endfor
  if no application was deleted, replicated or migrated
  AND offloading = Yes
    find out the least loaded server t from the central replicator
    while load_s > LW AND not all applications have been examined
      let app be the unexamined application with the highest
      ratio of non-local demand, B_total/B_s;
      if B_s > D and B_t > D
        replicate app on t
        load_s = load_s − load_s(app, t)
      else
        if server t accepts app
          migrate app to t
          load_s = load_s − load_s(app)
        endif
      endif
    endwhile
  endif
end
```

Figure 3: Replica placement algorithm. $Load_s$ denotes load on server s, $load_s(app, t)$ denotes load on server s due to demand for application app coming from clients in server t's area, and $load_s(app)$ denotes load on server s due to application app.

of replication would be fully compensated by the benefits from the new replica within one time period until the next execution of the placement algorithm provided the demand patterns remain stable during this time. In fact, we might choose to replicate even if this ratio is less than one (e.g., if we are willing to trade bandwidth for latency), or only allow replication when this ratio exceeds a threshold that is greater than one (e.g., to safeguard against a risk that a fast-changing demand might not allow enough time to compensate for the replication overhead). Hence, the algorithm provides the redeployment threshold R, and replicates the application on server i if $B_i/(A+U) > R$.

One caveat is the possibility of a vicious cycle of creating a new replica without enough demand, which will then be deleted because of the deletion threshold. Thus, we factor in the deletion threshold into the replication decision and arrive at the following final replication criterion: To create a replica on server i, the demand from i's vicinity, B_i, should be such that $B_i/(A+U) > R$ and $B_i > D$.

The migration threshold M governs the migration decision. The application is migrated only if the fraction of demand from the target server exceeds M, $B_i/B_{total} > M$, and if the bandwidth benefit would be sufficiently high relative to the overhead, $B_i/A > R$. Note that the latter condition does not include the updates bytes because migration does not increase the number of replicas. To avoid endless migration back and forth between two servers, we require that the migration threshold be over 50%; we set it at 60% in our experiments.

The above considerations hold when the server wants to improve proximity of servers to client requests. Another reason for replication is when the current server is overloaded. In this case, it might decide to replicate or migrate some applications regardless of their proximity or their demand characteristics. So, if the server is overloaded, it queries the central replicator for the least-loaded server and, if the application cannot be replicated there (because the new replica might be deleted again), migrates the application to that server unconditionally. To add the stability to the system, we use a standard watermarking technique. There are two load watermarks, high watermark HW and low watermark LW. The server considers itself overloaded if its load reaches the high watermark; once this happens, the server continues considering itself overloaded until its load drops below the low watermark.

Again, one must avoid vicious cycles. The danger here is that after migrating an application to server i, the current server's load drops to the normal range, and the application would then migrate right back to the current server (because its initial migration worsened the client proximity). To prevent this, the target server only accepts the migration request if its projected load after receiving the application will remain acceptable (that is, below low watermark). This resolves the above problem because the current server will not accept the application back. This also prevents a herd effect when many servers try to offload to an underloaded server at once. To allow load predictions, the source server must apportion its total load to the application in question, $load_s(app)$, and to the requests that come from the target server's area, $load_s(app, t)$. As a crude estimate, we can apportion the total load in proportion to the number of relevant requests.

```
for each region R do
  for every server j in the system
    Prob(j) = 0;
  endfor
  for each server i_k with a replica of the application do
    if load(i_k) < LW
      Prob(i_k) = 1;
    elseif load(i_k) > HW
      Prob(i_k) = 0
    else
      Prob(i_k) = (HW - load(i_k))/(HW - LW)
    endif
  endfor
  residue = 1.0
  Loop through the servers with a replica of the application
  in the order of increasing distance from region R
  for each such server i_k do
    Prob(i_k) = residue * Prob(i_k)
    residue = residue - Prob(i_k)
  endfor
  let total be the sum of the Prob array computed above
  if total > 0
    for each server i_k with a replica of the application
      Prob(i_k) = Prob(i_k)/total
    endfor
  else
    for each server i_k with a replica of the application
      Prob(i_k) = 1/n, where n is the number of replicas
    endfor
  endif
  output (R, Prob)
endfor
```

Figure 4: Algorithm for computing request distribution policy for a given application.

5.2 Request Distribution Algorithm

The goal of the request distribution algorithm is to direct requests to the nearest non-overloaded server with a replica of the application. However, an intuitive algorithm that examines the servers in the order of increasing distance from the client and selects the first non-overloaded server for the request (similar to the algorithm described in [2]) can cause severe load oscillations due to a herd effect [3]. Furthermore, our simulations show that randomization introduced by DNS caching may not be sufficient to eliminate the herd effect. The algorithm used in the RaDaR system [16] does not suffer from the herd effect but often chooses distant servers even when closer servers with low load are available. Thus, our main challenge was to find an algorithm that never skip the nearest non-overloaded server and yet reduce oscillations in request distribution.

An additional challenge was to make the algorithm compatible with our system environment. We use iDNS as our load-balancing DNS server [2]. For a given application[4] iDNS expects a *request distribution policy* in the form of tuples $(R, Prob(1), \ldots, Prob(N))$, where R is a region and $Prob(i)$ is the probability of selecting server i for a request from this region. Regions can be defined in a variety of ways and can be geographical regions (e.g, countries) or network regions (e.g., autonomous systems or BGP prefixes). For the purpose of this paper, we assume that each server i in the system is assigned a region R_i (represented as a set of IP addresses) for which this server is the closest. We also assume some distance metric between a region R_i and all servers that allows one to rank all servers according to their distance to a given region. The question of what kind of a distance metric is the most appropriate is a topic of active research in its own right; different CDNs use proprietary techniques to derive these rankings, as well as to divide client IP addresses into regions.

In our ACDN, the central replicator computes the request distribution policy in the above format and sends it to iDNS. The policy is computed periodically based on the load reports from ACDN servers (obtained by accessing load reporter scripts as discussed in Section 3), and also whenever the set of replicas for the application changes (i.e., after a replica deletion, migration or creation).[5] The computation uses the algorithm shown in Figure 4.

Let the system contain servers $1, \ldots, N$, out of which servers $i_1, \ldots, i_n$ contain a replica of the application. The algorithm, again, uses low watermark LW and high watermark HW, and operates in three passes over the servers. The first pass assigns a probability to each server based on its load. Any server with load above HW gets zero weight. If the load of a server is below low watermark, the server receives unity weight. Otherwise, the algorithm assigns each examined server a weight between zero and unity depending on where the server load falls between the high and low watermarks. In the second pass, the algorithm examines all servers with a replica of the application in the order of the increasing distance from the region. It computes the probabilities in the request distribution policy to favor the selection of nearby servers. The third pass simply normalizes the probabilities of these servers so that they sum up to one. If all servers are overloaded, the algorithm assigns the load evenly among them.

[4] We assume that every application uses a distinct (sub)domain name.

[5] Technically, the policy is computed by a control module within iDNS that we modified; however, because this module can run on a different host from the component that actually answers DNS queries, we chose to consider the control module to be logically part of the central replicator.

6 Performance

In this section, we evaluate the performance of ACDN using a set of simulation experiments. We first evaluate the effectiveness of the request distribution algorithm in achieving a good balance of load and proximity. We then study the effectiveness of our content placement algorithm in reducing bandwidth consumption and user latency.

6.1 Request Distribution

A request distribution algorithm would ideally direct client requests to their nearby replicas in order to reduce latency. At the same time, it must avoid overloading a replica in a popular region with too many requests. The simulation was conducted in a system with three replicas with decreasing proximity to the clients: replica 1 is closest to the clients, replica 2 is the next closest, and replica 3 is the farthest. The high watermark and the low watermark for the replicas are 1000 and 200 requests per second, respectively. Initially, there are 10 clients in the system. Each client starts at a randomized time and sends 5 requests per second. Then we gradually increase the number of clients to study the behavior of the algorithm when the replicas are overloaded. After that, we gradually decrease the number of clients to the original level to simulate a situation where the "flash crowd" is gone. To simulate the effect of DNS caching, each client caches the result of replica selection for 100 seconds. The results are shown in Figure 5. The left figure plots the number of requests served by each replica in sequential 100-second intervals. The bar graph on the right plots the request distribution every 10 minutes.

As can be seen from the figures, when the number of clients is small, the closest replica absorbs all the requests – the request distribution is determined by proximity. As the number of clients increases, load balancing kicks in and all replicas begin to share the load. Proximity still plays a role, however: note that replica 1 serves more requests than replica 2, which in turn serves more requests than replica 3. Also note that the load of none of the replicas ever exceeds the high watermark, which is usually set to reflect the processing capacity of the underlying server. When the number of clients decreases, the load in the three replicas decreases accordingly. Consequently, proximity starts playing an increasingly important role in the request distribution algorithm. When the load falls back to the original level, replica 1 again absorbs all the requests. The results indicate that our algorithm is efficient in utilizing proximity information while avoid overloading the replicas.

As targets for comparison, we also simulated two other algorithms: a pure random algorithm and the algorithm previously used in CDN brokering [2]. The pure random algorithm distributes requests randomly among the replicas regardless of the proximity. As can be seen from Figure 6, it achieves perfect load balancing among the three replicas. However, the clients might suffer unnecessarily from high latency due to requests directed to remote replicas even during low load.

The request redirection algorithm in the previous CDN brokering paper [2] works as follows:

- Select the closest replica whose load is less than 80% of its capacity. In our simulation, we set the capacity of all replicas to 1000 requests per second, the same as the high watermark used in ACDN.
- If no such replica exists, distribute the load evenly across all replicas.

The results are shown in Figure 7. When the load is low, replica 1 absorbs all the requests. When the load increases, the algorithm exhibits oscillations between the first and the second replicas, while the third replica remains unutilized. Moreover, note that the load on replica 1 can go substantially above the high watermark. These results show that, although this algorithm also takes into account both load and proximity, it does not perform as well overall as our ACDN algorithm. This is because it relies on a single load threshold (i.e. 80%) to decide whether a replica can be selected, which makes it susceptible to the herd effect. Although residual requests due to DNS caching add randomization to load balancing, they were not sufficient to dampen the herd effect. In contrast, our algorithm uses a pair of high watermark and low watermark to gradually adjust the probability of server selection based on the load of the server.

6.2 Content Placement

The goal of the content placement algorithm is to detect hot regions based on observed user demands, and replicate the application to those regions to reduce network bandwidth and client perceived latency. The simulation was conducted using the old UUNET topology with 53 nodes which we used for our previous study [16]. 10% of the nodes represent "hot" regions: they generate 90% of the requests in the system. The rest 90% of the nodes are "cold" regions and generate 10% of the requests. In this simulation, the request rates from each hot region and each cold region are 810 and 10 requests per second, respectively. The simulation starts with an 100 second warm-up period during which clients in each region start at randomized times. To simulate the effect of changing demand patterns, the set of hot regions changes every 400 seconds. The application size used in the simulation is 10M bytes. The size of an application response message is 20K bytes. Every minute we introduce an update into the system that needs to be propagated to all the replicas. The size of the update is 5% of the application size. The redeployment threshold used in the simulation is 4. The deletion threshold is set to half the redeployment threshold times the size of the application. The algorithm makes a decision whether it needs to replicate or migrate every 100 seconds.

We compare our approach with two other algorithms: a static algorithm and an ideal algorithm. In the static algorithm, a replica is created when the simulation starts and is fixed throughout the simulation. In the ideal algorithm, we assume that the algorithm can get instantaneous knowledge as which regions are hot or cold and then replicates or deletes applications accordingly. It represents the optimal case which cannot be implemented in practice. The results of the simulation are shown in Figure 8. The left figure shows the amount of network bandwidth consumed in the simulation per second. This is measured as the product between the number of bytes sent and the number of hops they travel. For example, if a replica sends 1000 bytes to a client which is 3 hops away, the amount of network bandwidth consumed is 3000 bytes. The right figure shows the average response latency among all clients in the system. We assume that the latency on each link in the topology is 10ms. For this preliminary study, we also assume that the processing overhead at the replicas is negligible. Both figures indicate that our ACDN algorithm can quickly adapt to the set of hot regions and significantly reduce network bandwidth and response latency. The spikes in the left figure are caused by the bandwidth incurred during the application replication process.[6] The migration algorithm was never triggered in this simulation because no region contributed enough traffic.

6.3 Redeployment Threshold

Choosing an appropriate value for the redeployment threshold is essential for achieving good performance of the protocol. With a low threshold, more replicas will be created in the system. This way the protocol is faster to detect and replicate to hot regions. Consequently, it can reduce the amount of network bandwidth due to application request and response traffic and improve end users' experiences. However, it also increases the amount of protocol overhead due to application replication as well as application updates. In the extreme case, an application may be replicated to all the nodes in the system. This allows all the requests to be served by a nearby server,

[6]Note that in some part of the curve the ACDN algorithm appears to perform slightly better than the ideal algorithm. This is due to random fluctuation in the simulation.

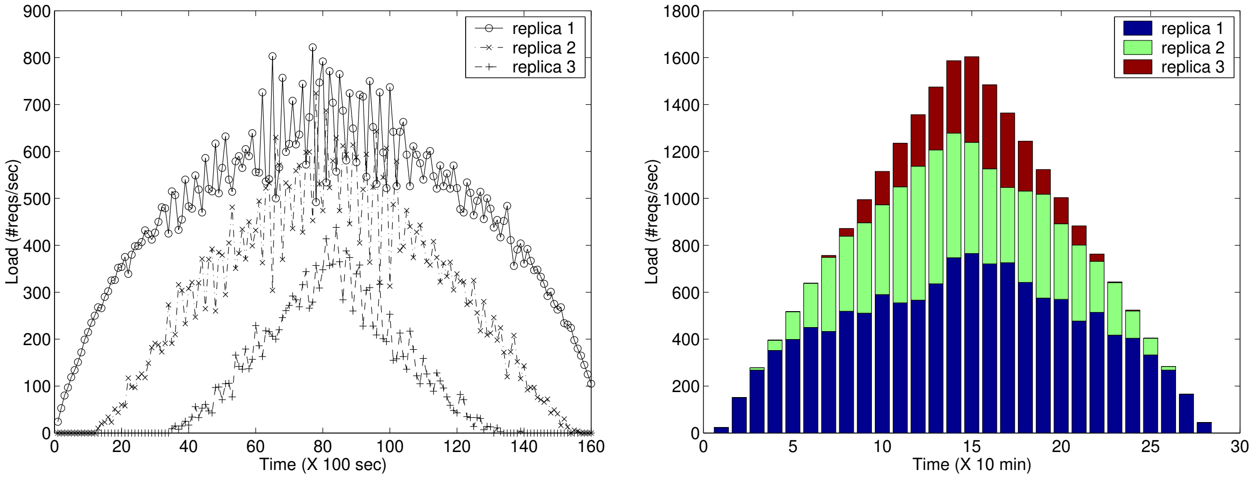


Figure 5: Request distribution in ACDN.

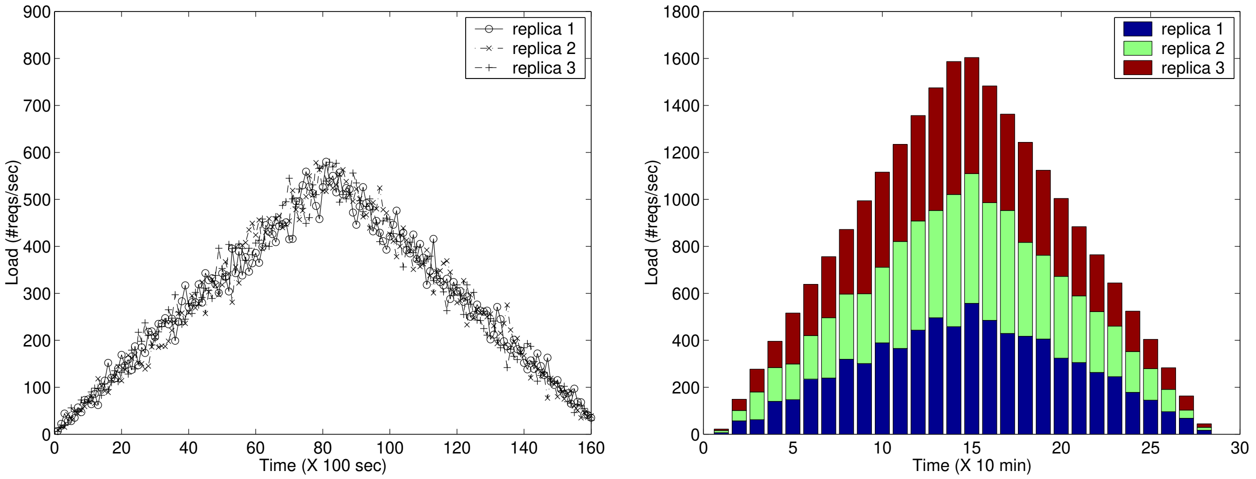


Figure 6: Request distribution in the random algorithm.

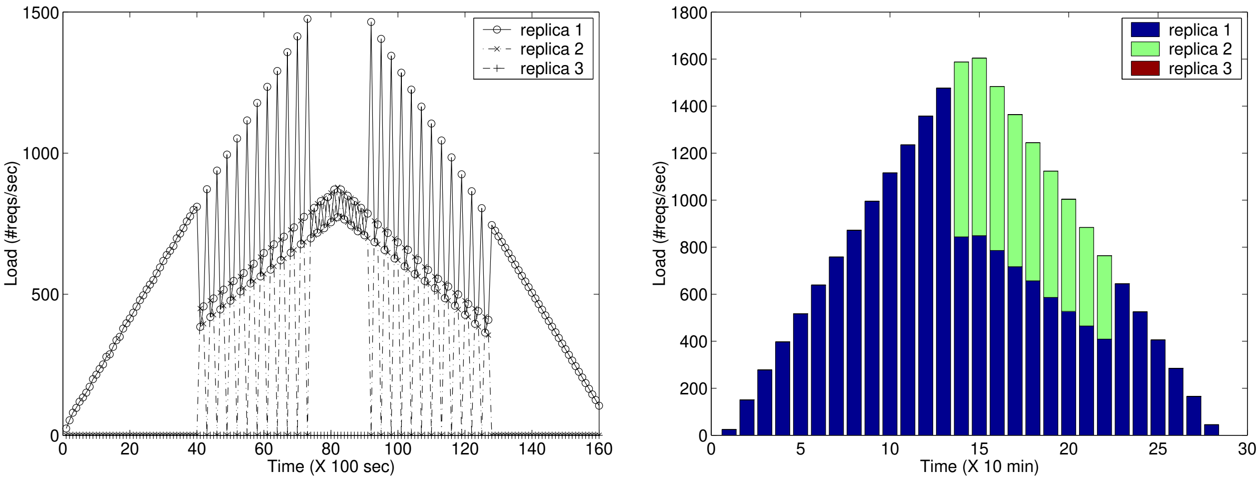
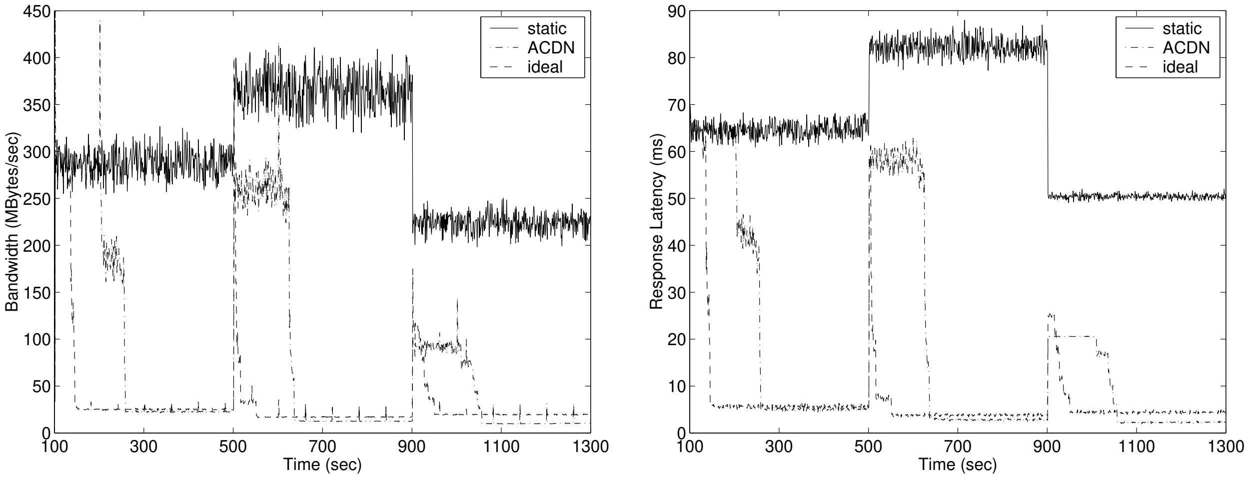


Figure 7: Request distribution in the brokering system.



(a) Network Bandwidth Consumption (b) Response Latency

Figure 8: Effectiveness of dynamic replication in ACDN.

but each update to the application has to be propagated throughout the entire system. On the other hand, with a high redeployment threshold, only a few replicas will be created in the system, corresponding to the most popular regions. This reduces the overhead due to application replication or updates, but increases the overhead of serving application requests.

Finding the right threshold is not a trivial task, however. It depends on many factors: the size of the application, the sizes and frequency of its updates, the traffic pattern of user requests, etc. We did some preliminary experiment to explore this trade-off for an 100M bytes application in the UUNET topology. In this simulation, the request rates from each hot region and each cold region are 81 and 1 requests per second, respectively. As before, we introduce an update into the system every minute that needs to be propagated to all the replicas. We used higher application size and lower request rates in this experiment to emphasize the effects of the overhead of creating and maintaining extra replicas relative to the benefits from increased proximity of replicas to requests. We vary the redeployment threshold and see its impact on the total amount of traffic on the network. The results are shown in Figure 9. The x-axis is the redeployment threshold used in the protocol, and the y-axis the total amount of bandwidth consumed in the entire simulation. The figure indicates that there is a "plateau" of good threshold values for this application: thresholds that are either too high or too low result in increased bandwidth consumption. As future work, we plan to investigate algorithms for adjusting the threshold automatically to optimize the performance of an application.

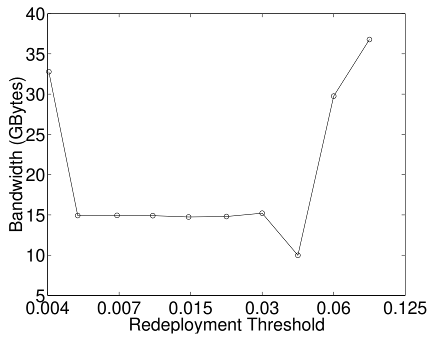


Figure 9: Effect of Redeployment Threshold

7 Related Work

The importance of supporting dynamic content in a CDN has been recognized and several proposals have been described that address this problem to various extents. Most of this work concentrates on caching dynamic responses and on mechanisms of timely invalidation of the cached copies, or on assembling a response at the edge from static and dynamic components [4, 5, 18, 9]. The fundamental difference between ACDN and these approaches is that the former replicates the computation as well as the underlying data used by the application while the latter handles only responses and leaves the computation to the origin server.

The Globule system [15] uses an object-oriented approach to content replication. It encapsulates content into special Globule objects, which include replication functionality and can be used to distribute static or dynamically generated content. Compared to our ACDN, Globule gives each object a flexibility to use its own policy for distribution and consistency maintenance while ACDN applies its policies to all hosted applications. On the other hand, Globule uses its own protocols to implement distributed objects and it requires compiling applications into Globule objects as well as modifying the Web server to be able to use Globule objects. Our ACDN is built entirely over HTTP, which simplifies firewall traversal, and works with existing applications and unmodified Web servers.

As discussed in the introduction, one can run an ACDN on top of a general process migration system such as Ejasent and vMatrix [20, 1]. Finally, one can also run ACDN servers on top of a distributed file system where each server acts as a client of the global file system and where each file is replicated among CDN servers through caching within the file system. This approach replicates computation but is limited to only very simple applications since it does not replicate the environment, such as resident processes. Also, ensuring that different components of the application are always mutually consistent becomes difficult since consistency is maintained for each file individually.

Turning to algorithms, ACDN involves two main algorithms - an algorithm for application placement and an algorithm for request distribution. Request distribution algorithms are closely related to load balancing and job scheduling algorithms. In particular, the issue of load oscillation that we faced has been well-studied in the context of load balancing (see. e.g., [3] and references therein). However, ACDN has to address the same issue in a new environment that takes into account client proximity in addition to server load. The algorithms by Fei et al. [6] and by Sayal et al. [19] use client-observed latency as the metric for server selection and thus implicitly account for both load and client proximity factors. Both algorithms, however, target client-based server selection, which does not apply to a CDN.

Many algorithms have also been proposed for content or server placement. However, most of them assume static placement so that they can afford to solve a mathematical optimization problem to find an "optimal" placement (see, e.g, [12, 21] and references therein). Even with empirical pruning, this approach is not feasible if content were to dynamically follow the demand. Some server placement algorithms use a greedy approach to place a given number of servers into the network. These algorithms still require a central decision point and are mostly suitable for static server placement. Our ACDN placement algorithm is incremental and distributed. Among the few distributed placement algorithms, the approach by Leff et al. [13] targets the remote caching context and does not apply to our environment where requests are directed specifically to servers that already have the application replica. The strategies considered by Kangasharji et al. [10] assume a homogeneous request pattern across all regions. Our algorithm can react to different demands in different regions and migrate applications accordingly. Finally, the strategy mentioned by Pierre et al. [14] places a fixed number of object replicas in the regions with the highest demand. Our algorithm allows a variable number of replicas depending on the demand and takes into account the server load in addition to client proximity in its placement decisions.

Our ACDN content placement algorithm is an extension of our earlier RaDaR algorithm [16]. However, because ACDN replicates entire applications, its placement algorithm is different from RaDaR in that it takes into account the size of the application and the amount of application updates in content placement decisions.

8 Conclusions

This paper describes an ACDN - a middleware platform for providing scalable access to Web applications. Accelerating applications is extremely important to CDNs because it represents CDNs' unique value that cannot be offered by client-side caching platforms. ACDN relieves the content provider from guessing the demand when provisioning the resources for the application and deciding on the location for those resources. The application can be deployed anywhere on one server, and then ACDN will replicate or migrate it as needed using shared infrastructure to gain the economy of scale.

We presented a system design and algorithms for request distribution and replica placement. The main challenge for the algorithms is to avoid a variety of "vicious cycles" such as endless creation and deletion of a replica, or migration of a replica back and force, or oscillations in request distribution, and yet to avoid too much deviation from optimal decisions in a given instance. Our preliminary simulation study indicated that our algorithms achieve promising results.

To date, we only experimented with one read-only application as a testbed for ACDN [11]. In the future, we would like to gain more experience with the system by deploying a variety of applications on it. In particular, we would like to explore various ways to support user updates to the application data, from relying on a shared back-end database to possibly replicating these updates among application replicas in a consistent way.

Acknowledgments

We would like to acknowledge Pradnya Karbhari who implemented the first version of the ACDN prototype and Fred Douglis for his involvement in the first prototype. We also thank the anonymous reviewers for comments on an early draft of the paper.

References

[1] Amr A. Awadallah and Mendel Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *7th Int. Workshop on Web Content Caching and Distribution (WCW 2002)*, August 2002.

[2] A. Biliris, C. Cranor, F. Douglis, M. Rabinovich, S. Sibal, O. Spatscheck, and W. Sturm. CDN brokering. In *6th Int. Workshop on Web Caching and Content Distribution*, 2001.

[3] M. Dahlin. Interpreting stale load information. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1033–1047, October 2000.

[4] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the Symposium on Internet Technologies and Systems*, pages 83–94. USENIX, December 1997.

[5] ESI - accelerating e-business applications. `http://www.esi.org/`.

[6] Zongming Fei, Samrat Bhattacharjee, Ellen W. Zegura, and Mostafa H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *INFOCOM*, pages 783–791, 1998.

[7] S. Gadde, J. Chase, and M. Rabinovich. Web caching and content distribution: A view from the interior. In *5th Int. Web Caching and Content Delivery Workshop (WCW5)*, 2000.

[8] A. Van Hoff, J. Payne, and S. Shaio. Method for the distribution of code and data updates. U.S. Patent Number 5,919,247, July 6 1999.

[9] Arun Iyengar and Jim Challenger. Improving Web server performance by caching dynamic data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (ITS-97)*, pages 49–60, Berkeley, December 8–11 1997. USENIX Association.

[10] Jussi Kangasharju, James W. Roberts, and Keith W. Ross. Object replication strategires in content distribution networks. In *Proceedings of the Sixth Int Workshop on Web Caching and Content Distribution (WCW)*, 2001.

[11] Pradnya Karbhari, Michael Rabinovich, Zhen Xiao, and Fred Douglis. ACDN: a content delivery network for applications (project demo). In *ACM SIGMOD Int. Conference on Management of Data (SIGMOD-02)*, pages 619–619, 2002.

[12] Shay Kutten, Israel Cidon, and Ran Soffer. Optimal allocation of electronic content. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, pages 1773–1780, Los Alamitos, CA, April 22–26 2001. IEEE Computer Society.

[13] Avraham Leff, Joel L. Wolf, and Philip S. Yu. Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1185–1204, November 1993.

[14] Guillaume Pierre, Ihor Kuz, Maarten van Steen, and Andrew S. Tanenbaum. Differentiated strategies for replicating Web documents. *Computer Communications*, 24(2):232–240, February 2001.

[15] Guillaume Pierre and Maarten van Steen. Globule: a platform for self-replicating Web documents. In *6th Int. Conference on Protocols for Multimedia Systems*, pages 1–11, October 2001.

[16] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 101–113. IEEE, May 1999.

[17] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison-Wesley, 2001.

[18] Michael Rabinovich, Zhen Xiao, Fred Douglis, and Chuck Kalmanek. Moving Edge-Side Includes to the Real Edge—the Clients. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[19] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. In *Workshop on Internet Server Performance*, June 1998.

[20] Utility computing: Solutions for the next generation it infrastructure. http://www.ejasent.com/whitepaper2.shtml.

[21] Adam Wierzbicki. Models for internet cache location. In *The 7th Int. Workshop on Web Content Caching and Distribution (WCW)*, 2002.