An Architecture for Real-Time Active Content Distribution

Chengdu Huang, Seejo Sebastine, and Tarek Abdelzaher
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
{ch4pp, seejo, zaher}@cs.virginia.edu

Abstract

The phenomenal growth of the world-wide web has made it the most popular Internet application today. Web caching and content distribution services have been recognized as valuable techniques to mitigate the explosion of web traffic. An increasing fraction of web traffic today is dynamically generated and therefore intrinsically difficult for replication using present static approaches. Scalable delivery of such active content poses a myriad of challenges, including content replication, update propagation, and consistency management. This paper makes two contributions: (1) it describes a scalable architecture for transparent demand-driven distribution of active content; (2) the system can provide real-time delay guarantee on content access. Our approach involves migrating the scripts which generate the dynamic web traffic, and their data, from the server to active content distribution proxies (ACDPs) nearest to the requesting clients. Our system is implemented and deployed on PlanetLab [19], a real-world distributed Internet testbed. Experimental data show that significant improvements are observed in effective throughput and client response time, and delay bounds on content access can be guaranteed with a very high probability.

1. Introduction

The dramatic explosion of the Internet as an information source generates significant demand for efficient content delivery architectures. The World Wide Web is the dominant Internet application today, motivating a web-based content distribution service. Web caching and content delivery networks are currently the key performance accelerators in the web infrastructure. Such services have traditionally operated on static content.

An increasing fraction of the traffic on the web today is dynamically generated as many web sites evolve to provide sophisticated e-commerce and personalized services. Such dynamically generated content is uncachable using present static approaches. There has been considerable interest in caching/replicating dynamic content recently [8, 14, 16, 11, 23, 9, 20, 27]. Most approaches investigate caching the dynamic pages resulting from script execution on the server. However, since many dynamically-generated pages are personalized, the hit ratio on them is very low. Hence, in many cases, caching them does not aid much in reducing server load.

Instead, we propose a content distribution service in which any requested content-generating script (and thus the need for computing power) can be migrated transparently from its origin server to one of a network of active content distribution proxies (ACDPs) that is nearest to the requesting client. Active content refers to the scripts themselves that generate dynamic web pages, rather than the content they generate. When dynamically-generated content is requested from a proxy, the proxy requests the script which generates that content from the origin server (unless it already has it), installs it locally, and executes it to serve the request. The script is thus said to be replicated. File accesses made by replicated scripts are handled by the proxy’s I/O subsystem which is instrumented to fetch, upon the first reference, a copy of the accessed data objects from the origin server. This copy is subsequently kept consistent with its counterpart on the origin server using a QoS-aware consistency management protocol. The protocol allows one to specify QoS contracts on per-content-class basis. A QoS contract specifies a delay bound on content access and temporal consistency requirements that should be enforced by the run-time system. For example, one may specify that all client requests for the replicated object should be served within 500ms, all fetched premium objects should be no more than 10-minutes out of date, and all fetched basic objects should be no more than 30-minutes out of date.

Our architecture is useful for sites like my.yahoo.com where a very large number of personalized pages are dynamically generated from a much more limited base of underlying fairly static data objects. Such pages are often called pseudo dynamic content. Caching such pages is not
useful since they are not shared by multiple clients. Caching the underlying data objects and their scripts, on the other hand, can significantly reduce the load on the origin server. Such architecture provides the benefits of better scalability and improves client-observed access latencies.

In our architecture, ACDPs and content providers must mutually authenticate themselves before active content can be migrated such that this content (i) is shared with trusted ACDPs only, and (ii) is itself trusted by the ACDPs. While we do not develop authentication services ourselves, schemes proposed recently for secure-aware web caches [17] can be applied directly to our architecture as described in related work. In this paper, we focus on the real-time demand-driven active content distribution protocol. Content providers can subscribe to our content distribution service (presumably at a price) to benefit from lower origin server load, improved service scalability, and lower latency to their clients. The service behaves as a load-balancer for the original content web servers in the sense that the request load for a popular server gets distributed among multiple proxies. The only requests that are filtered through to the origin servers are ‘preliminary’ page accesses and active proxy requests to maintain the data consistency. From a local ISPs’ perspective, our service reduces backbone traffic for which the ISP is responsible to the backbone provider, as more requests for active content are satisfied locally. This reduces the costs paid by the ISP. Clients also observe an improvement in access latencies since more pages are served to the client from a closer location on the Internet.

The rest of this paper is organized as follows. Section 2 presents a survey of related work. Section 3 elaborates the active content distribution architecture and its various components. Section 4 describes the implementation details. Section 5 presents the experimental evaluation of our architecture. Finally, Section 6 concludes the paper and discusses avenues for future work.

2. Related Work

Proxy caching of static content [7, 10, 26] has been discussed at length in prior literature. The benefits of static web content caching, however, are limited considering the significantly increasing fraction of uncacheable objects under the current web caching schemes [25].

Web server workload can be reduced by server-side caching of dynamically generated content [14, 16, 9, 22, 28]. Smith et al. [14] proposed a solution for cooperative caching of dynamic web content on a distributed web server. In their system, clustered nodes collaborate with each other to cache and maintain result consistency using Time-To-Live methods. Iyengar et al. [9, 16] propose that cache servers export an API that allows individual applications to explicitly cache and invalidate application-specific content. These techniques are developed for caching identical requests at origin content providers and are not very feasible for proxy caches. An efficient consistency management protocol is proposed in [28] for the dynamic content cached at the front-end of web servers. Dynamic pages are classified based on the URL patterns and can be invalidated by an application. This is a complementary approach to the fine-grained invalidation/update scheme based on dependency graphs [9].

Beck et al. [4] discussed the issues of replicating dynamic content due to platform heterogeneity and proposed using a portable channel representation. Candan et al. [6] addressed the problem of making changes in database content reflected to cached dynamically generated web pages by intelligently invalidating dynamic content in caches based on the state changes in the database.

Dynamic content caching can have inherent limitations in terms of performance improvement when deployed at web servers. For example, the user perceived latency due to the backbone delay can not be reduced. In contrast, proxy caching can reduce the backbone traffic and the user response time due to the backbone delay by caching dynamic content near the clients. In this paper, we focus on the proxy caching of the scripts that generate dynamic content.

Cao et al. [8] proposed an elegant architecture in which a piece of Java code (called a cache applet) is attached to each dynamic document and cached on the proxy. This cache applet has the task of deciding whether a cached version of the dynamic document is to be returned, whether a new version is to be created on the proxy, or whether the request is to be forwarded to the origin server for regeneration. The cache applet runs on the proxy whenever a request for a cached document is received. This approach is very flexible and can be used to maintain consistency in an application-specific manner as well as dynamically modify existing documents. However, this flexibility is achieved at a price. It requires starting up a new Java process (virtual machine or compiled code) for every request in order to execute the applet. In addition, keeping track of result equivalence requires creation and maintenance of a pattern-matching network. Since the cache applet for each document is independent (for security reasons), cache-applets used to implement result-equivalence-based caching would need to save and restore their pattern-matching network to persistent storage, which can be a significant performance constraint. Besides, the cache applet can not handle the situations in which the dynamic content is not written in Java.

Calo and Verma [5] independently proposed a Java-based architecture for active content caching very similar to ours. In their architecture when proxy caches receive dynamic content requests, they determine the execution pa-
rameters that may be needed for this invocation and download the program and configuration information from the origin server. Our scheme differs from theirs in that our service explicitly addresses QoS guarantees on content access of different classes.

Akamai [1] uses Edge Side Include (ESI) [12, 21] technology to handle dynamic content in their content distribution network service. ESI is a markup language that developers can use to identify content fragments for dynamic assembly at the network edge servers. Therefore, only those non-cachable or expired fragments need to be fetched from origin servers, thereby lowering the need to retrieve complete web pages and decreasing the workload of origin servers. In this scheme, dynamic content still can not be cached in proxy cache servers.

Rabinovich et al. proposed a CDN architecture [20] for replicating dynamic content. They use a metafile to describe the relationship between an executable file and the data files it needs. The focus of [20] is on how to create and delete replicas in CDN according to server workload and how to redirect client requests to the best replica. Yet no mechanism is provided to give delay guarantees on content retrieval.

Our own previous work [15] has studied the feasibility of a CDN service that replicates content on the Internet so that a global delay bound is guaranteed on access requests from anywhere in the CDN. The focus of [15] has been on static content only. The main contribution was in selecting replica locations to guarantee bounded network latency.

Proxy caches that are capable of delivering dynamic content are both more vulnerable to bad content and may have more control in that in some applications they can change state at the origin server (e.g., by selling products on behalf of an e-commerce site). Traditional end-to-end security mechanisms are no longer sufficient to ensure integrity. Gemini [17] is a security-aware publisher-centric web cache. Each content object from the origin server is associated with an access control list specifying which caches are allowed to store this object. The technical details of Gemini are beyond the scope of this paper but it is worth mentioning that the security mechanism can be adopted within our architecture to authenticate caches to servers and vice versa via their digital signatures, and ensures that active content is replicated and executed only in the presence of mutual trust.

3. Architecture

In this section we present the architecture for our real-time active content distribution service.

In our architecture, some content distribution proxies are capable of replication and execution of active objects, in addition to the traditional caching of static objects. We call them active content distribution proxies (ACDPs). Content providers’ servers that export active content for replication are called origin servers. We call such proxies and servers that support our extensions, compliant. The ACDPs take care of maintaining the scripts and data on the cache consistent with that on the origin server and providing real-time guarantee on content access requests from clients. In the following three subsections we discuss, respectively, protocol and architectural issues regarding origin servers, ACDPs, and consistency management with mechanisms for enforcing real-time guarantees.

3.1. The Origin Server

An origin server is one which contains the original copy of a particular site. In this paper, we associate each site with a single origin server. Any updates to the content of the site originate at that server. To support on-demand active content replication, the server software needs to be slightly modified. However, the extensions should be integrated in a way that does not violate current standard protocols when a compliant server communicates with a non-compliant (i.e., legacy) content distribution network server/web cache or vice-versa.

The current version of HTTP allows for clients and/or proxies to negotiate the type of requested content with the servers. For example, a client or proxy can specify that it accepts only documents written in French. Our departure point lies in introducing content type definitions that refer to active content (to be used in requests to fetch the content-generating scripts themselves). We also introduce an HTTP header option that alerts the receiver that active content is requested. In our architecture, active content types are classified by the language of the script (for which the ACDP should have a compiler or interpreter) and/or the operating system for which the script is designed. Thus, one type of scripts, for example, would be Perl scripts on UNIX. A non-compliant server that receives a request for a URL such as http://www.foo.com/cgi-bin/prog.cgi would execute the named script locally by default, sending the resulting page back to the client or proxy. When a compliant server receives such a request, it looks for the new header option. If it is not found it serves the result of script execution, as an ordinary server would. Otherwise, it identifies the request as coming from a compliant sender (an ACDP) and examines the types of scripts the proxy says it supports. If the list does not include the requested script type, the server executes the script locally. If the particular script type is supported by the proxy, the replication protocol is invoked and the script is replicated. Our extensions are therefore fully interoperable with the current HTTP standards. The software of the origin server requires only a few changes in order to recognize requests from ACDPs. A dy-
namic content module is implemented that determines if a requested script to be executed locally or sent to the requester. In the latter case, the server may send additional information pertaining to consistency management of the script replica as will be discussed shortly. This information is called a ticket.

The active content distribution protocol is illustrated in Figure 1. Steps 1, 2, and 7 of the protocol (shown in Figure 1) are identical to standard HTTP. Steps 3, 4, 5, and 6 are specific to active content distribution.

![Figure 1. The Active Content Distribution Protocol](image)

### 3.2. The Active Content Distribution Proxy

The ACDP is the architectural component responsible for replicating scripts from remote origin servers. Each ACDP is composed of three primary components; a proxy cache, an active server and a consistency manager, as shown in Figure 2. The proxy cache handles the job of caching static content, as well as the dynamic content generated by executing server scripts if needed. The active server handles the job of executing one or more types of related dynamic content, say for example cgi, fast-cgi etc. Hence the ACDP can have multiple active servers to handle different types of dynamic content. For example, one server can handle Perl scripts, while another can handle ASP content. We have used a modified version of the freely available Squid proxy cache [24] as our proxy cache component and the Apache web server [2] as our active server. The Squid proxy cache includes a redirector component. Clients contact the Squid cache component with content requests. If the requests are for static content, Squid directly contacts the origin server, retrieves the results, and caches them for future accesses. However, if the requests are for dynamic content, Squid uses its redirector to modify the client requested URLs. Squid redirects such requests for dynamic content to the Apache active server component, which then fetches the script from the origin server, executes it and returns the results to Squid. Since both proxy and active server components are either running on the same physical machine or on machines on the same local network, the overhead involved is far less than going over the Internet backbone to the origin server.

In addition, the Squid proxy cache can cache the results of execution of the script on the active server if so configured. Hence future requests for such scripts, with the same inputs, may be served from the Squid cache. The other components of the architecture as shown in Figure 2 are a URL rewriting engine, and a script cache, both of which are part of the active server component. The URL rewriting engine is invisible to the outside world and serves only those requests (for active content) which are forwarded from the proxy cache component. The URL rewriting engine then rewrites these URLs as proxy requests to the server which the client had originally requested.

![Figure 2. Components of an ACDP](image)

When the active server component in the ACDP receives a reply from the origin server, it executes it with the client requested parameters. The fetched script is also saved in its local script cache. Any data that the script requires in order to execute is also fetched from the server and replicated as will be detailed in the next section. In this paper, we deal only with read-only scripts, i.e., those that do not modify their data. This simplifies consistency management.

### 3.3. Consistency Management

The consistency management protocol ensures that each script and its data objects are kept (weakly) consistent with the server. Each ACDP has a consistency manager module which handles the job of maintaining consistency. Consistency management is based on periodic polling, which is similar to the common time-to-live (TTL) approach. Each cached script is associated with a TTL. The scripts can be re-executed as long as the TTL has not expired. We believe that in practice the scripts will not change very often. Hence, weak consistency is sufficient. The data objects used by the scripts are associated with a polling period at which they should be retrieved from the origin server. They are considered consistent with the origin server as long as their period has not expired. During that time, these cached data objects can be accessed by the scripts upon user requests.
A new copy of the data objects is requested from the origin server upon period expiration. To reduce periodic polling, the server may associate each such data object with a ticket. The ticket specifies the number of accesses that should occur during one polling period for periodic polling to continue. If the required number of accesses does not occur during some polling period, the cache does not contact the server upon period expiration. Instead, it contacts it upon the arrival of the next request. Hence, if the request rate is low no extra polling overhead is observed.

In our service model, all content is divided into classes. Each class \( class_i \) has an associated deadline, minimum consistency, and nominal consistency requirements. The deadline, \( D_i \), specifies the desired response time for this class. It sets the order in which polling requests are made to the origin servers. The minimum and nominal consistency requirements specify the maximum and nominal tolerable staleness of data objects in that class, denoted \( S_{max} \) and \( S_{nom} \) respectively. For example, stock data may have a nominal tolerable staleness \( S_{nom} = 3 \) minutes, and a maximum tolerable staleness \( S_{max} = 15 \) minutes. A request is a “timely hit” if it meets the deadline constraint.

The consistency manager maintains the mapping between the scripts and their data files. It contains a timer thread which is invoked at constant intervals of time. At each timeout, the consistency manager checks which data files need to be re-fetched from origin servers. These files are fetched in parallel by concurrent threads. The polling period for content can be dynamically adapted in a QoS-sensitive manner in response to network and server workload conditions.

A main objective of our active content distribution service is to provide delay bounds with high confidence for content access. It is generally desirable that all content can some nominal consistency \( S_{nom} \). However, keeping stringent consistency entails more update traffic. Hence, upon overload, consistency may be degraded upto a maximum staleness threshold \( S_{max} \). We argue that for many applications, getting prompt response from the server is more important than being served with the freshest content. Hence, we can sacrifice consistency of content in favor of conforming to the latency bound. When the timely request hit ratio drops to a certain value (called low watermark), we degrade consistency of content gradually. Since all content is categorized into different classes, the consistency degradation will be started with the lowest priority class. It will be continued until hit ratio rises up to a certain level or all content classes are already at their maximal tolerable staleness. When timely hit ratio reaches a certain point (called high watermark), the system will try to tighten the consistency again, starting from the most important class, until timely hit ratio drops to the low watermark or all classes are already at their nominal consistency level.

4. Implementation Details

In this section we discuss the implementation details of our architecture and protocol. In our ACDP, we chose to use the freely available Squid proxy cache \([24]\) as the proxy cache component and the Apache web server \([2]\) as the active server component, as mentioned in Section 3.

4.1. Active Content Distribution Protocol

We describe our implementation by following the journey of a request arriving from a client to an ACDP. Upon arrival of the request, we use a redirector component to selectively rewrite URLs requested by the client. In our case, we redirect client URLs requesting dynamic content to our active server rather than the requested origin server. Client requests for static content are not redirected and are directly forwarded to the requested origin server. We use the redirect capabilities available in Squid for the redirector component. We chose jesred as an external redirector of Squid. By default, Squid rewrites any HTTP “Host:” header in redirected requests to point to the new host to which the URL has been redirected. However, we turn this option off, because we would like to save the name of the host for whom the request was originally meant, to eventually forward the request to that host.

When the Squid proxy cache receives a request for dynamic content, it first checks to see if it already has a cached version of that content. After performing checks to determine if the cached content is still valid, it either returns the cached page, or redirects the request to the appropriate active server to obtain the most up-to-date version of that content. When an active server receives such a redirected request, it first checks to see if the script that has been requested is already cached locally. If so, it executes the script with the current inputs and returns the results to the Squid proxy cache. In case a cached version of the script is not available locally, the active server initiates a URL rewrite operation. The URL is rewritten to fetch the script from the server specified in the HTTP “Host:” header, which in this case, holds the name of the server for which the request was originally meant. We have used the functionality provided by Apache’s “mod_rewrite” module to perform the URL rewriting.

Since we wish to cache the script that is returned, we force these requests to be proxy requests. These proxy requests are handled by Apache’s “mod_proxy” module. For example, if a request meant for \( http://foo.com/cgi-bin/prog.cgi \) was redirected to the active server residing at \( http://t2.cs.virginia.edu/ \), then the active server on determining that the URL was for a script called \( prog.cgi \), which is not cached locally, rewrites the URL again to \( http://foo.com/cgi-bin/prog.cgi \).
When the origin server receives such a proxy request for dynamic content, it has to be able to distinguish that the request originated from an ACDP. In order to facilitate this, the active server modifies the HTTP request header before forwarding the request. It modifies the HTTP [13] Accept header, by adding the string

```
```

```
```
to the header. When the dynamic content module on the destination server recognizes this special extension, it comprehends that the script is not to be executed locally, but is to be returned to the requester as is. If security mechanism (like in [17]) is available to do authentication, the server would also have to look at the digital signature of the requesting proxy to decide whether it can be trusted with the script. If the proxy is not trusted, the script is executed locally.

When a trusted active server receives a reply from the origin server, it checks the origin server’s signature versus those supplied at content-provider subscription time. If the server is trusted, it caches the reply (i.e the script) in its local script cache. We have used Apache’s proxy caching capabilities (provided by the mod_proxy module) for this purpose. Finally, it executes the script with the parameters supplied by the client and returns the results to the Squid proxy cache.

### 4.2. Script API

In order for the active server to be able to fetch the data for the script from the origin server, we require that the script I/O be performed using our read/write primitives. We also require that the script executable be linked with our library. This allows us to implement location transparency such that a script always finds its needed data objects locally regardless of where it executes, even when our consistency manager has to fetch them first from the origin server. The requirement to use a particular middleware API is arguably a limitation. This limitation, however, is shared by all technologies that depart from legacy code. For example, while Java is recognized as an important vehicle for software portability, at the time of its inception an obvious limitation of having that portability was that a user had to learn a new programming language and translate into it all current code. Similar arguments can be made regarding CORBA, RPC and other languages or middleware. In this paper, we simply present our tools and their potential, without engaging in the argument over the commercial viability of their interfaces in their current form. In the future, it may be worthwhile to investigate hiding our API calls beneath a standard I/O library interface making them transparent to the programmer.

### 4.3. Real-time Guarantee Mechanisms

As discussed in Section 3.3, the consistency manager of an ACDP dynamically adjusts the tolerable staleness of different content classes to keep the timely hit ratio high.

Among all the components in an ACDP, the proxy cache, being the interface of the ACDP to clients, has the most accurate information of the total processing time of each client request. So we make the proxy cache monitor the timely hit ratio and periodically report to the consistency manager. With this information, the consistency manager will be able to make decision on adjusting consistencies of different content classes. The period of adjusting consistency should be comparable to consistency of content classes. The minimal adjustment a consistency manager makes to consistency values of content classes is called adjustment granularity. Right after the consistency manager determines the adjustment to consistency, the result is sent back to the proxy cache. Proxy cache adjusts the expiration time of data (the results of executed scripts) it cached to synchronize with the consistency manager.

We implemented two mechanisms to do the adaptive consistency adjustment:

**Linear Adjustment** Every time the consistency manager receives reports of timely hit ratio, it makes a fixed change (of at most one adjustment granularity), for either relaxing or tightening consistency.

**Proportional Adjustment** The change the consistency manager makes is determined by 
\[(HR_c - low\_watermark)/c\] where \(HR_c\) is the current timely hit ratio reported by proxy cache, and constant \(c\) is a configurable system parameter. This mechanism is "proportional" in that the adjustment to be made is proportional to the difference between current and targeted timely hit ratio. Compared to linear adjustment, proportional adjustment can be faster in bringing timely hit ratio back to the low watermark but also has the potential of overreacting because the effect of last adjustment may have not been fully taken when the consistency manager is about to make another adjustment.

We compare the performance of these two mechanisms in Section 5.

### 5. Evaluation

In this section, we evaluate the performance of our system with experiments in both emulation environment and PlanetLab [19], a real-world Internet testbed. Our experimental results demonstrate that our system can lead to significant average client perceived latency improvement and better service scalability. We also show that our mechanisms provide real-time guarantees for client requests, leveraging
the trade-off between client request processing latency and content staleness.

Our testbed consists of a full implementation of an ACDP and one origin server using instrumented Squid and Apache servers as discussed in Sections 3 and 4. To emulate a large number of clients sending requests, we use SURGE (Scalable URLs Generator)[3], a tool that generates references matching empirical measurements of web request size distribution, relative file popularity, embedded file references, idle periods of individual clients, and locality of reference.

To show the general performance gain in client-observed response time, two sets of experiments are conducted for scenarios with and without the ACDP. Over 50,000 client requests for more than 500 scripts were generated and sent to the server within approximately 15 minutes. In the first set of these experiments, these requests were sent directly to the origin servers. In the second set, the requests were sent via an ACDP. We conducted these experiments on PlanetLab [19]. The origin server and ACDP server are deployed in Italy, Europe (Universita di Bologna) and North Carolina, USA (Duke University) respectively. Clients running SURGE are in Virginia, USA (University of Virginia). Clients are relatively close to the ACDP because while origin servers can be virtually anywhere in the world, in a widely deployed CDN system, every client should be able to connect to a nearby ACDP server.

The end-to-end delay consists of the Internet backbone delay from clients to the ACDP server, the delay from the ACDP server to the origin server, the server and/or proxy processing time and the client delay. Our experimental results (Figure 3) show that there is a 200% reduction in latency on average.

**Figure 3. Average Latency Improvement**

Figure 4 plots the observed response time of a community of users accessing the same script with and without an ACDP. The client observed latency is limited below 80 milliseconds for most of the time when the ACDP is present. Note how most of the time the script is executed on the active server and results are sent back to the clients from the ACDP. The latency is correspondingly low. A few spikes occur when the active server goes across the wide area network to re-fetch the data files from the origin server according to the consistency requirement of the script. Without ACDP, the latency is not only much higher compared to that of with ACDP but also has a very large variability which can be attributed to the instability of the long transoceanic Internet path.

In the following, we are going to illustrate and evaluate our system’s performance under high workload. Since PlanetLab is a shared platform, we were explicitly discouraged from running extensive overload experiments on its servers. Instead, we used NIST Net [18] as a WAN emulator to emulate the environment of Planet-Lab servers we used. NIST Net is a Linux Kernel module that can add configured latencies to incoming packets. We used two servers in our LAN as the origin server and ACDP server, and configured the NIST Net module according the network latency data we gathered from PlanetLab. To demonstrate the fidelity of this emulated WAN environment, we redid the experiments above in our LAN with NIST Net. Figure 3 shows that the results of emulated WAN matches fairly well with the those of PlanetLab. From this point on, our experiments were conducted on the emulated WAN with servers in our LAN. We intentionally used slow machines (AMD400MHz, 256M RAM) as the origin server and ACDP to make server overload easier.

Our architecture achieves reduction of workload on the origin server and client-perceived latency at the cost of adding workload to the ACDP. Figure 5 depicts one of the important metrics of workload – CPU utilization – for both the origin server and ACDP. It is shown that the ACDP considerably reduces the CPU utilization of the origin server, effectively shifting the computational demand from the remote server to the local proxy. Figures 5-i, 5-ii, and 5-iii show the CPU utilization of the ACDP and the origin server
CPU utilization(%) polling=15s

(i) Observations

ACDP
Origin Server(with ACDP)
Origin Server(without ACDP)

CPU utilization(%) polling=30s

(ii) Observations

ACDP
Origin Server(with ACDP)
Origin Server(without ACDP)

CPU utilization(%) polling=60s

(iii) Observations

ACDP
Origin Server(with ACDP)
Origin Server(without ACDP)

(iv) Observations

15s
30s
60s

Figure 5. CPU utilization and impact of polling period

(with and without the cache) for different polling periods at the ACDP. As might be expected, higher polling periods correspond to higher utilization, which quantifies the overhead of the scheme. Active cache utilizations for different polling periods are compared in Figure 5-iv.

Now we are going to evaluate the two consistency adjustment mechanisms discussed in Section 4.3. In the experiments, we have three content classes, all of which have 10 seconds as nominal staleness and 60 seconds as maximum tolerable staleness. We chose the low watermark and high watermark to be 0.95 and 0.98. The proxy cache reports the current timely hit ratio to the consistency manager every 20 seconds.

Figure 6 illustrates the workload used in our experiments. At the beginning, there are only a small number of clients requesting scripts of a small set (labeled “working set”). At time unit 3 and 4, we introduce many more clients to the system whose requests span a large set of scripts. The change in the number of clients and the gradual expansion of the consistency manager’s working set over time are plotted in Figure 6.

In two sets of the experiments, we choose 300ms and 500ms as the latency bounds for all three content classes respectively. As we can see in Figure 7, 8, 9, and 10, the client requests introduced at time unit 3 and 4 significantly impact the timely hit ratio at those times. The consistency manager immediately takes action to relax consistency. The timely hit ratio gradually rises back. To comparison purpose, we also include the hit ratio without adaptively adjusting consistency of different classes in Figure 7-10. We choose 30 seconds as the consistency for all the classes and make this value fixed during the experiments.

Figure 7 and 8 depict the performance of linear adjustment and proportional adjustment for tight latency bound (300ms). From the figures we can see that linear adjustment is conservative in relaxing consistency and makes a minimal change every time unit (adjustment granularity) and finally relaxes all three content classes to their maximum tolerable consistency. In contrast, proportional adjustment makes abrupt changes to content’s consistency when a large latency bound miss ratio is seen. Hence, all the content classes quickly reach their tolerance bound. Note that proportional adjustment’s aggressiveness does help in pulling up the timely hit ratio faster. Also note that in these experiments no class has chance to get better consistency because...
the latency bound is tight.

Figure 9 and 10 show the results with a less stringent latency bound (500ms). Different from the previous case, not all content classes have to be pushed to maximum tolerable staleness because of the larger latency bound. The process of first relaxing then tightening the consistency of content was seen in both linear and proportional adjustment cases. Proportional adjustment still enjoys a slightly faster recovery of timely hit ratio than linear adjustment. On the other hand, linear adjustment achieves better consistency for content. In Figure 9 we can tell that with linear adjustment, class 0, the most important class, enjoyed the nominal staleness all the time and at time unit 19 class 1 also got back to the nominal staleness while with proportional adjustment class 0 suffered a long period of degradation to the maximum staleness and the other two classes did not get as good consistency as those in linear adjustment case, as shown in Figure 10.

Three additional observations can be made from the figures. First, when the consistency manager stabilizes, the timely hit ratios are generally very high (> 90% for 300ms as latency bound, > 95% for 500ms as latency bound). Second, for all cases, the three classes had roughly the same timely hit ratios which indicates fairness. Third, it is clear that consistency of the lowest priority class gets relaxed first and gets tightened last, as it should be.

6. Conclusion and Future Work

In this paper, we proposed and implemented a scalable architecture for dynamic web content replication with real-time guarantees. Our solution involves replicating the scripts which generate the active web content, and their data, from the server to ACDPs which are nearer to the clients and a mechanism to maintain consistency and achieve latency bounds on content access. Our experimental results showed significant improvement of effective throughput and client response times and attainment of the real-time delay guarantee with high probability. Reducing workload on origin servers also improves service scalability. In addition our architecture provides customizable consistency semantics.

Acknowledgement

The work reported in this paper was supported in part by the National Science Foundation under grants ANI-0105873 and CCR-0093144. We are grateful to the anonymous reviewers for their valuable comments.
Figure 10. Proportional Adjustment (bound = 500ms)

References