Exploiting Object Relationships for Deterministic Web Object Management*

Mikhail Mikhailov and Craig E. Wills

Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609
{mikhail,cew}@cs.wpi.edu

Abstract

This paper presents a novel Web object management mechanism called MONARCH. The primary goal of our mechanism is to provide strong cache consistency without requiring servers to maintain per-client state. MONARCH also seeks to reduce overhead incurred by heuristic-based cache consistency mechanisms.

Servers first classify objects at a site based on object change characteristics. Then the servers analyze relationships between the objects composing a page in conjunction with object change characteristics and compile that information into concise and explicit object management instructions. Client caches follow these instructions and deterministically manage objects, including those objects that are currently managed heuristically. The mechanism utilizes request-driven server invalidation, where servers invalidate only those objects that reside on the currently retrieved page. The approach works well when pages contain objects with a mix of change characteristics, which is often the case for content from popular sites.

We have implemented the proposed mechanism in a prototype system and present its design and implementation. We also present results of a simulation-driven performance evaluation of our approach. Simulation results show that strong cache consistency without any per-client server state can be achieved and that for many categories of Web pages the performance of our approach is better than that of heuristic policies in terms of generated traffic. We also show that although exposing constituent page components to clients tends to generate more requests that reach the server, the amount of cacheable content increases by 30%–50%.

Keywords: Web Caching, Distributed Object Management, Cache Consistency, Server Invalidation, Change Characteristics, Object Relationships, Object Composition.

1 Introduction

The content and services being offered on the World Wide Web are increasingly becoming richer, more frequently changing, and more personalized. There are now sites serving content encoded for wireless devices and providing Web services. The user population of the Web is not only growing, it is also changing in that more automated clients are being introduced.

The sheer size, diversity, and constant growth of the World Wide Web demand powerful

*This work is partially supported by the National Science Foundation Grant CCR-9988250.

techniques for scaling the Web and improving its performance. Since the early days of the Web, organizations successfully deployed caching proxy servers to lower bandwidth usage on their Internet connections and decrease the response time for their internal users. Academic and industrial efforts to advance the state-of-the-art in Web performance led to the introduction of caching hierarchies, interception proxies, surrogate servers and more recently to the deployment of Content Distribution Networks (CDNs). More recent proposals such as Akamai's EdgeSuite based on the Edge Side Includes (ESI) [8] allow servers to expose page structure to clients and allow clients to cache page fragments at the edge of the network. On-going work on mechanisms such as the Web Cache Invalidation Protocol (WCIP) [13] aims at having origin servers invalidate objects cached at CDN servers.

While caching and content delivery infrastructures are integral parts of the Web, typical browser and proxy caches still rely on heuristics for object management. Heuristic-based approaches are imprecise by definition and provide weak cache consistency where caches may serve stale objects to their clients. Heuristic approaches also generate unnecessary traffic when they revalidate objects that have not changed. In this paper we present an alternative approach to managing Web objects, called Managing of Objects in a Network using Assembly, Relationships and Change cHaracteristics (MONARCH). The goal of MONARCH is to provide strong cache consistency without requiring servers to maintain per-client state. MONARCH also seeks to reduce the amount of messages and bytes that the cache and server exchange. To achieve its goal, MONARCH exploits the relationships between objects composing a Web page and takes into account change characteristics of these objects. Based on that information, MONARCH selects an object management strategy that is specific to and most appropriate for a given set of objects. It then communicates the chosen management strategies to the caches. MONARCH works particularly well when pages contain objects with a mix of change characteristics (both frequently and rarely changing), which is often the case for content from popular sites.

In our previous work, we evaluated potential gains of our approach and reported encouraging results [20]. In this paper, we present the details of our approach and describe the design and implementation of the MONARCH prototype system that we built. We show that while MONARCH does require servers to maintain state, the amount of state does not depend on the number of clients. We also show that MONARCH performs better than heuristic policies and similarly to the optimal object management policy on a range of realistic scenarios.

The rest of the paper is organized as follows. In Section 2 we discuss how objects are currently managed on the Web. In Section 3 we introduce our classification of object change characteristics and present our approach to deterministic object management. Implementation of the MONARCH prototype system is discussed in Section 4. The simulator that we wrote and used for performance evaluation of MONARCH, simulation scenarios, and results are presented in Sections 5 and 6 respectively. We compare our work to ideas proposed by others in Section 7 and conclude the paper with a discussion of our on-going work.

2 Current Practice

We discuss how objects are managed on the Web today using the Web page shown in Figure 1. This example is motivated by home pages of popular news portals. The container object *CO* is changing frequently—every few minutes—because content designers update the top story and add and remove links leading to the major news articles. Irrespective of frequent updates to the news stories, every request for *CO* results in a different response because the origin server dynamically generates

CO, changing which ad banner image to display and where on the page to place it. Embedded objects *EO1—EO3* change only occasionally. Such changes are applied by a human, occur at unpredictable points in time, and are saved under the same name as the previous version of the object. Objects *EO4* and *EO5* never change. If changes are required, content designers save them under a different name, effectively creating a new object.

Let us now consider how these objects are currently managed within a proxy-style caching architecture. Caches cannot store the container *CO* because it changes frequently and either carries an explicit indication that it is uncacheable or has no cache control meta information associated with it. The five embedded objects *EO1—EO5* change rarely or never and may be cached. Currently, the more cache-friendly origin servers assign cacheable objects an expiration or a last modification time, via the `Expires` and `Last-Modified` HTTP response headers respectively. When an explicit expiration time is not available, caches, such as Squid [18], consider a configurable fraction of object's age to be a reasonable estimate for the freshness lifetime for that object. The age of an object is the difference between current time and that object's last modification time. The heuristic used here, referred to as the Alex protocol [2], suggests that the younger files are likely to change sooner than the older files. It is impossible for caches to deterministically know when cached objects *EO1—EO5* become stale because servers cannot accurately predict object expiration times and heuristic time-to-live is imprecise by definition. Also, servers can inadvertently provide misleading expiration and last modification times [19]. As a result, caches may serve stale objects to their clients. Caches also generate unnecessary traffic and place additional load on the origin servers when they validate objects that have expired in the cache but are unchanged at the origin server.

3 MONARCH—A Deterministic Approach to Object Management

Our work is motivated by the lack of determinism and the inefficiency exhibited by the heuristic approach to cache consistency. Studies show that unnecessary validation requests generated by caches represent 15-18% [11], 30% [16] and 37% [1] of all requests served by origin servers. These requests utilize server resources that could have been used to serve other requests. More importantly, revalidations increase client perceived latency. Prior research has shown that revalidations increase the average latency by 1–5.7 times and the median latency by 5.5–9.2 times for an individual object [6]. Depending on the network distance between the cache and server and on the HTTP protocol option used, client perceived per page latency increases due to cache validations by 2.6–5.1 times [12]. Also, as reported in [10], reducing the quality of embedded objects does not significantly improve client perceived latency. This result suggests that revalidation of embedded objects, particularly smaller ones, is not significantly better latency-wise than fetching these objects anew.

In this section we describe our approach to providing strong cache consistency and reducing the inefficiency of the heuristic cache consistency approach. We use the example in Figure 1 to illustrate how our approach works.

3.1 Object Change Characteristics

A key observation motivating our approach is that Web objects not only have different content types, but also exhibit distinct change characteristics. Our classification of object change characteristics is given in Figure 2. The three categories on the left—Static, Periodic and BoA—represent *predictable* changes. Objects in these predictable categories can be managed *deterministically*: cached and never validated, cached for a predetermined period, and always retrieved from the origin server re-

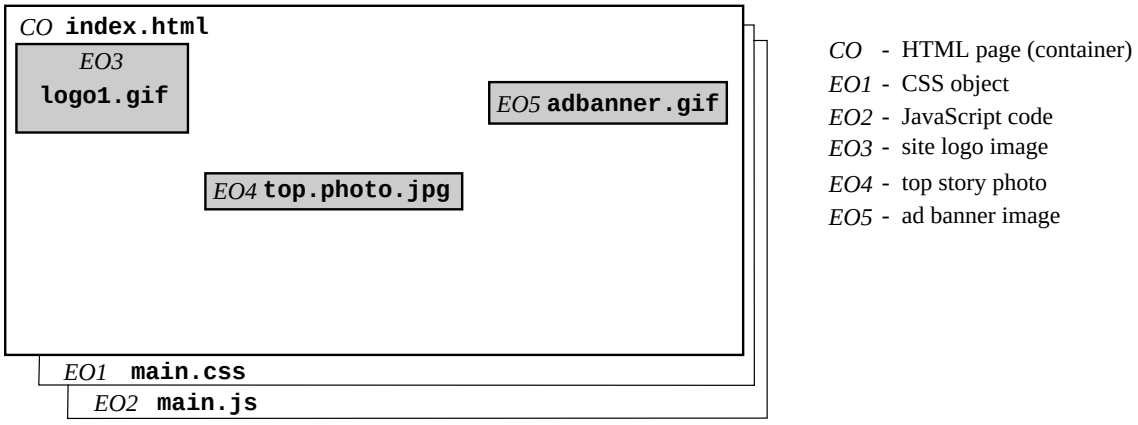


Figure 1: Home Page of a Popular News Site

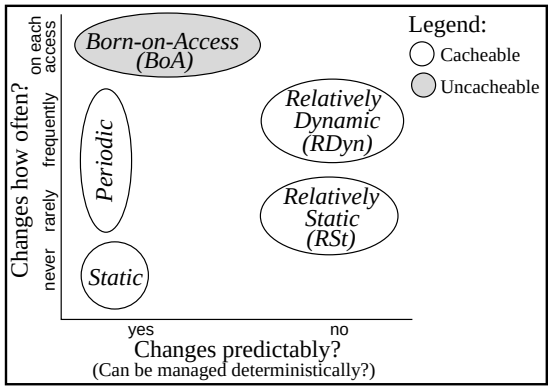
spectively. The two categories on the right—RDyn and RSt—cover objects that can be cached but change unpredictably and, therefore, cannot be managed deterministically. Existing caches manage objects in the latter two categories using heuristics.

One question about our approach is whether it is feasible to classify a large number of objects at a Web site based on these change characteristics. Our viewpoint is that servers already automatically generate many objects based on measurable events or at regular intervals. To mark the resulting objects with appropriate change characteristics is a trivial addition to these automated tasks. The type of an object or its location within a file system may define its change characteristic. For example, a manually created image, such as a digital photograph, may be marked as static. All objects in one directory may be treated as RSt and in another directory as RDyn. Another observation, that we believe makes this approach feasible, is that only the most popular objects at a site require classification, with others being managed using currently adopted mechanism. Further, even within popular pages, only frequently changing objects need to be marked as such, with others left unmarked and classified as RSt by default.

3.2 Design

The basic idea of MONARCH is to exploit the fact that some objects on a page change frequently and must be retrieved from the origin server. In our work, we use the notion of a *volume* [15], a collection of related objects, to group the set of objects composing a page. Upon requests for frequently changing objects, the server can provide fine-tuned targeted invalidation information to clients for objects residing in the same volume as the requested object. MONARCH identifies a single object—a *manager*—within a volume and uses it to help manage nondeterministically changing objects in the volume. The process of selecting a manager object is carried out as follows.

The server first classifies objects at a site based on their change characteristics. The server then performs per-page compilation by analyzing the relationships between objects constituting the page in conjunction with change characteristics of these objects, selecting the object with the most dominant change characteristic to be the manager for that page, and assigning *Content Control Commands* (CCCs) to all objects on the page. CCCs provide concise and explicit instructions to clients on how to handle each object. The server also keeps track of object updates and of changes in volume membership so that on repeat visits from clients it can invalidate those nondeterministically changing ob-



Class	Example
Periodic	periodically updated weather map
BoA	container *CO* in Figure 1
Static	objects *EO4* and *EO5* in Figure 1
RSt	objects *EO1*—*EO3* in Figure 1
RDyn	list of latest news stories

Figure 2: Classification of Object Change Characteristics

jects that are on the same page as the requested object. When volume membership changes, the server repeats the process of page compilation and may change the CCC commands associated with objects on the updated page.

In our example in Figure 1, the container *CO* has the most dominant change characteristic since it is changing on every access while all other objects either never change or change infrequently. Therefore, the server selects the container object to be the manager for the page and assigns appropriate CCCs to all objects based on that. The CCCs assigned to objects *EO4* and *EO5* notify caches that these objects can be cached for as long as necessary and do not need to be validated. Objects *EO1*—*EO3* belong to the RSt category and would be managed heuristically in current practice. In our approach, however, servers can use the deterministic retrieval of the container *CO* to invalidate the RSt objects embedded in it so that caches can store these RSt objects and treat them as fresh until the server sends an invalidation. Figures 3 and 4 show sample server responses to requests for a RSt object (*EO1* in this case) and for the BoA container *CO* respectively. The CCC associated with the container instructs caches to keep only meta information about the *CO* and provides name and current version of the volume associated with the page. Both responses were generated by the MONARCH prototype system, which we discuss in Section 4.

```
HTTP/1.1 200 OK
Date: Sat, 04 May 2002 10:04:31 GMT
Server: Apache/1.3.19 (Unix)
CCC: cmd=C; version=996109498
```

Figure 3: Server response with CCC for *EO1*—*EO3*

```
HTTP/1.1 200 OK
Date: Sat, 04 May 2002 10:11:00 GMT
Server: Apache/1.3.19 (Unix)
CCC: cmd=CM; version=1020507067;
     vname=index.html;
     vversion=1008104231-1020507067
```

Figure 4: Server response with CCC for container *CO*

The server reply to a request for the container *CO* looks slightly different if the server sends an invalidation. Suppose a client requested and cached the objects shown in Figure 1. Later on, after object *EO1* is updated, the same client is requesting the container *CO* again, indicating to the server that it had previously seen version `1008104231-1020507067` of the volume named `index.html`. The server determines which objects the volume contained at the time of the previous request and invalidates the RSt objects that have been updated—*EO1*. For efficiency, the server piggybacks invalidation information onto its response to the client [11], as shown in Figure 5.

The client invalidates object *EO1* in its cache

```
HTTP/1.1 200 OK
Date: Sat, 04 May 2002 10:45:58 GMT
Server: Apache/1.3.19 (Unix)
CCC: cmd=CM; version=1020509159;
     vname=index.html;
     vversion=1008104231-1020509159
CCC: cmd=INV;
     objs="main.css^1020508627"
```

Figure 5: Server response with CCC for container *CO* and invalidation for embedded object

and retrieves a new copy of it. The client reuses cached copies of those RSt objects that the server did not explicitly invalidate: *EO2* and *EO3*. If the client does not indicate to the server that it had previously seen the *CO* object, the server can either provide no invalidation information at all, as is done currently, or inform the client about the current versions of the RSt objects embedded in *CO*.

3.3 Design with Page Components

The scenario that we just discussed is fairly simple in that it has a single frequently changing container object embedding a few rarely changing objects. Now that major Web browsers provide better and more consistent support for the W3C's Document Object Model specification [7] and with the introduction of page assembly services at the edge of the network [8], Web sites may choose to expose constituent page components to clients. Under such a scenario our example in Figure 1 may look to clients as shown in Figure 6.

Those parts of the original BoA container *CO* that change frequently or on every access are now captured in separate components: *CMP1* and *CMP2* are BoA, and are the only objects in our modified page that must be retrieved from the server on every access; *CMP4* and *CMP5* are RDyn and can be cached until invalidated by the server. We relocated cacheable and deterministically manageable content—a daily opinion poll on a controversial question—into the Periodic component *CMP6*. We also moved RSt content that is shared between many pages at the site into components *CMP3* and *CMP7*. This content, as well as the new RSt container *CO'*, can now be cached.

The transition of the container object from BoA to RSt fundamentally affects how we manage this set of objects. Since *CO'* is now RSt, the server selects a different object that is changing frequently to assist caches in managing the container *CO'* and other non-deterministically changing objects on the page. The CCC that the server associates with *CO'* is shown in Figure 7. That CCC instructs caches that they may cache the container but should always satisfy the provided *precondition*—retrieval of *CMP1* in this case—before reusing the cached copy of the container.

```
HTTP/1.1 200 OK
Date: Sat, 04 May 2002 10:45:58 GMT
Server: Apache/1.3.19 (Unix)
CCC: cmd=C; pre=username.cmp;
     version=1008104231;
     vname=index.html;
     vversion=1008104231-1020509159
```

Figure 7: Server response with CCC for container *CO'*. CCC includes precondition.

Upon receiving a request for *CMP1*, referred from the RSt container *CO'*, the server attempts to invalidate the container and nondeterministically changing objects embedded in it.

4 MONARCH Prototype Implementation

We have designed and built a prototype system implementing the MONARCH approach to object management. The system consists of three components: the MONARCH Proxy Server (MPS), MONARCH Web Server (MWS) and MONARCH Volume Manager (MVM). We

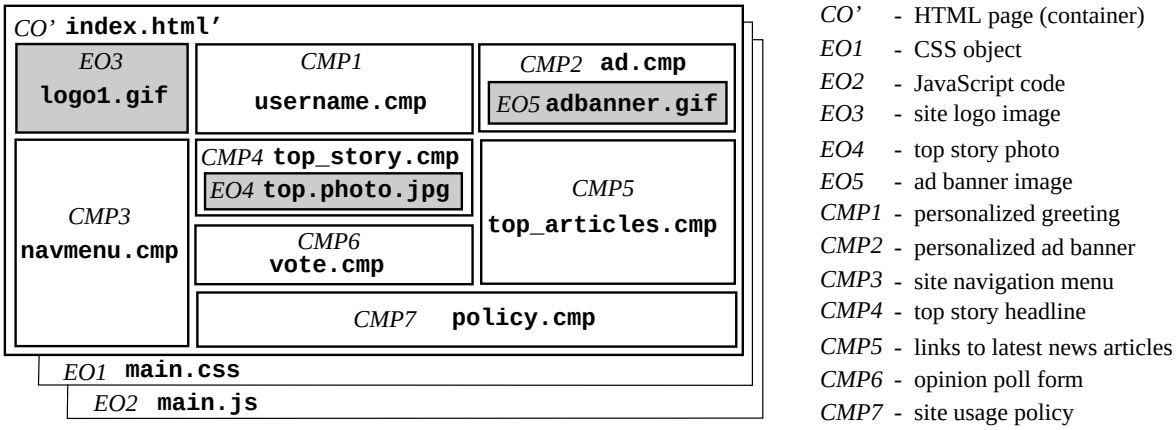


Figure 6: Objects and Components Composing Home Page of a Popular News Site

describe each of the components next.

4.1 MONARCH Volume Manager

The MONARCH Volume Manager is responsible for monitoring content available at a Web site and keeping track of object changes and volume membership changes. When an object update takes place, the volume manager determines which volumes are affected and updates volume membership information, which is stored in a relational database (MySQL). MVM updates volume version only when volume membership changes. Depending on a particular mix of objects at a site, the volume manager can run periodically, or be invoked by trigger mechanism upon object updates. For example, the volume manager can run when updated objects are uploaded to a production server from a staging server. The Web server can also invoke MVM to process dynamically generated objects.

4.2 MONARCH Web Server

We implemented the MONARCH Server as a plug-in for the Apache Web server. Apache can be configured to hand off all or certain requests to MONARCH plug-in. The main responsibility of the MONARCH server is to communicate with the volume manager and obtain object and volume version and volume invalidation information that is included in the server response. The MONARCH server provides its clients with targeted invalidations that are likely to be immediately useful. MWS maintains no per-client state and relies on its clients to provide volume information on subsequent visits. When a previously retrieved volume name and version is present in the request, MWS asks the volume manager to determine which objects need to be invalidated. If client has no previous volume information, or chooses not to use it, MWS cannot perform volume invalidation.

The MONARCH server is capable of assembling page components into a monolithic page for clients that either cannot or are not authorized to perform assembly themselves. MWS also adjusts HTTP response headers associated with the assembled pages.

4.3 MONARCH Proxy Server

The MONARCH Proxy Server is also implemented as a plug-in for the Apache Web server. Upon receiving a client request, the proxy attempts to find the requested object, either retrieving it from its cache or fetching it from the origin server. MPS always manages objects using the MONARCH object management policy, but falls back to heuristic policy if the server does not provide CCC commands.

Whenever MPS contacts the origin server to validate a cached object, it includes the cached object version identifier and the cached volume version identifier in its request. Upon receiving the server response, MPS examines the attached CCC commands and removes those objects that the server invalidates. If MPS receives a request for a cached object that the server associated a precondition with, MPS always satisfies the precondition first, by fetching the precondition object from the server.

5 Web Object Management Simulator

In this section we describe the Web Object Management Simulator (WOMS) that we developed in order to evaluate the performance of MONARCH and compare it to that of other object management policies on a range of scenarios.

WOMS is a discrete event-based simulator that handles request arrival and object update events. WOMS generates requests using an exponential distribution with a given mean interarrival time. It generates object update events based on object change characteristics. WOMS can run in two distinct modes. While running in *object composition* mode, WOMS exposes all objects to client requests. While running in *monolithic* mode, WOMS never generates client requests for components that constitute the top-most container object. WOMS still takes components into account in that the size of the resulting container is the sum of sizes of all constituent objects, and the change characteristic of the resulting container is the most dominant of all constituent objects. In both modes, each object, including private components, is updated individually to guarantee that the same scenario simulated under the two modes always has exactly the same sequence of object updates. It also makes it easy to simulate breaking a large container page into components.

WOMS contains a Web content simulation component that generates Web objects based on provided descriptions of object change characteristics and relationships. WOMS can clone an object, creating new objects with the same change characteristic as the original object, but different names. WOMS uses cloned objects to simulate rotating ad banners by cycling through a pool of cloned objects in a round-robin fashion and embedding the next object in the pool into the parent object. WOMS can also simulate addition of a new, never before seen, object to the page by creating a large pool of cloned objects. WOMS can also generate and write to disk real Web pages (see Section 8). WOMS assumes an infinite cache and no cache replacement policy. Object management policies are implemented as plug-ins—as long as policies use the simulator's programming interface, new policies can be written and added to the simulator. The simulator simply feeds each occurring event to all available plug-ins.

6 Performance Evaluation

Our goal in this work is to compare the performance of MONARCH to that of the existing policies that maintain no per client state at the server. In particular, we are interested in comparing MONARCH with heuristic policies. We used the number of stale objects served to clients and the number of messages and bytes transferred between the cache and server as the performance metrics. All object management policies studied in this work faithfully obey object expiration times and do not cache objects marked as uncacheable. Currently WOMS does not interoperate with the MONARCH prototype system, therefore we implemented the MONARCH policy as a simulator plug-in. In addition to MONARCH (M) we simulated the following policies: 1. the **Optimal (Opt)** policy that has the perfect knowledge of object updates, maintains strong consistency and contacts the server only when absolutely necessary; 2. the de facto standard **Heuristic** policy, with 5% (H5) and 10% (H10) of the object's age used

as adaptive expiration time; 3. the **Always Validate (AV)** policy that validates cached nondeterministically changing objects on every access and 4. the **Never Validate (NV)** policy that never validates cached objects.

In designing scenarios for the simulation, we attempted to create realistic scenarios that cumulatively cover a wide range of pages available on the Web today. We informally analyzed pages at a set of Web sites and noted such page characteristics as object composition, object sizes, and update frequencies. We used these characteristics to create synthetic content. Even though specific Web sites motivated our scenarios, each scenario depicts a *category* of Web pages rather than a specific page. Using only synthetic content is sufficient for preliminary evaluation of our policy across a range of Web page categories. To model Web content more accurately and to validate our preliminary results we also plan to collect and use actual content from a set of Web sites.

We evaluated each scenario under different request arrival frequencies and simulation lengths. While changing these parameters affects the absolute results, the relative performance of the policies studied remains the same. For the results presented here, the mean request inter-arrival time was set at 15 minutes, and the length of the simulation was set at 3 weeks. For two scenarios these parameters are different and are described below. To ensure that objects (especially RSt ones) have more realistic creation times, each simulation run had a warm up period of 6 months during which the simulator generated and processed only object update events. We now describe each of the scenarios used and present the simulation results.

6.1 News Portal

The first simulation scenario is similar to the one discussed earlier in the paper and mimics the home page of a busy Web news portal. Types of objects composing the page in monolithic and object composition modes are shown at the top of Table 1. BoA objects change on every access and St objects do not change at all. Modification times for RSt and RDyn objects are drawn from pre-defined ranges using a uniform distribution. The range configured for the RSt container object, for example, is 1–2 days. Therefore, the time between any two modifications of the container is at least one day and not more than 2 days. Other RSt objects in this scenario have ranges on the order of weeks and months. Ranges for the RDyn objects in this scenario are on the order of tens of minutes and a couple of hours.

In monolithic mode, BoA, Per, RDyn and one RSt components are merged together to produce a (larger) container object with the BoA change characteristic. Even though the simulator continues updating the RDyn, RSt, and Per components, these changes are masked by the BoA components on the page. In both monolithic and object composition modes, the page contains rotating ad banners. In addition, from time to time a new, never before seen object, representing a topical news photo, is created and added to the page. Popular news sites, such as `boston.com` and `cnn.com` have pages that resemble the page modeled by this scenario.

Simulation results for this scenario are shown in Table 1. For each policy the table shows the average number of: requests that the caching proxy sent to the origin server, bytes that the server sent to the proxy, stale objects that the proxy served to clients, and unnecessary validation requests that the proxy sent to the server.

To better understand how to read the table, consider the results for the Optimal policy. In monolithic mode, the page consists of 13 objects, one of which changes on each access and is uncacheable. On the first request, the Optimal policy fetches all 13 objects and caches 12 of them. On subsequent requests it always fetches the BoA object and fetches other objects only when they change. Thus on each page retrieval the Optimal policy always sends at least one request to the server. On average, across multiple retrievals of the page, the Optimal policy

sends 1.65 requests to the server. All other policies, except for the Always Validate policy, send the same or a few more requests to the server. The Always Validate policy sends substantially more requests to the server than other policies in order to ensure strong consistency, although HTTP/1.1 pipelining, if supported, could reduce the impact of these requests on the response latency.

In object composition mode, the server makes more objects composing the page visible to its clients. As a result, the presence of 2 BoA objects on the page and frequent changes in the RDyn objects force the Optimal policy to send on average 3.60 request to the server per page retrieval compared to 1.65 in monolithic mode. The number of bytes received from the server, however, decreases in object composition mode since more of the content can be cached and reused.

In monolithic mode, all objects embedded on the page are RSt. As a result, the simulated policies have to deal with only a handful of object changes. Nevertheless, as the table shows, heuristic policies H5 and H10 serve 0.06 and 0.08 stale objects to the client, while still making 0.13 and 0.06 unnecessary validation requests to the server per page retrieval respectively. The MONARCH policy not only provides strong cache consistency and contacts the server the same number of times as the Optimal policy, but also does not noticeably increase number of bytes that the server sends to the cache. In object composition mode, where policies have to handle changes to more objects, invalidation traffic introduced by MONARCH becomes more noticeable, but the amount of overall traffic from the server per page retrieval is still less than under heuristic policies. We also notice a significant increase in the amount of stale content and unnecessary validation requests under heuristic policies.

Comparing the results in the top and bottom parts of each column we notice that exposing components forming the page to clients resulted in more requests reaching the server, while the number of bytes served by the server was reduced by at least 53%. In our earlier work we examined home pages of popular Web sites and also found that treating pages as composed from objects increases the amount of cacheable content by 50% [20]. To lower the number of compulsory requests for BoA objects reaching the server, servers could advertise a single *aggregate* BoA object that is used to generate all the required BoA objects at once. Once a client receives such an aggregate object, or a *bundle* [21], it recovers all individual BoA objects encapsulated in it.

6.2 Discussion/Storytelling Site

The next scenario simulates pages that contain content submitted by Web users. Page composition for this scenario is shown at the top of Table 2. The RDyn component representing user-submitted content changes every few minutes and contributes the most bytes to the page. Two more RDyn components change every few minutes and every 12 to 24 hours respectively. RSt objects include the container object, embedded images and two components. Update ranges for the RSt objects are on the order of a few months to a year. Ad banners are rotating within a separate public component: a layer or a frame. The site that motivated this scenario is `slashdot.org`. Given the popularity of `slashdot.org` and availability of the code that runs it [17], it is not surprising that many other sites are similar to it.

For this scenario we decreased the observation period from 3 weeks to 1 day in order to focus on the effect of frequently changing RDyn objects on the studied policies. We also lowered the mean request inter-arrival time from 15 to 5 minutes to generate more requests over the course of the simulation. We believe that for a busy discussion site both of these changes are realistic.

Simulation results for this scenario are shown in Table 2. We see that heuristic policies, even with unnecessary requests to the server, gener-

Table 1: News Portal Scenario

monolithic (13 objs): BoA container, 8 RSt and 4 St objects. composition (21 objs): RSt container, 2 BoA, 3 RDyn, 2 Per, 9 RSt and 4 St objs							
Mode	Metric (per page retrieval)	Policy					
		Opt	M	H5	H10	AV	NV
Monolithic	Requests to Server	1.65	1.65	1.78	1.72	9.65	1.65
	Bytes from Server	67284	67284	67315	67300	69203	67273
	Stale Objects			0.06	0.08		1.7
	Unnecessary Validations			0.13	0.06	7.99	
Object Composition	Requests to Server	3.60	3.60	5.79	5.36	15.66	2.67
	Bytes from Server	29809	29823	30316	30140	32704	6748
	Stale Objects			0.13	0.27		6.6
	Unnecessary Validations			2.20	1.77	12.06	

ated less traffic than both the Optimal policy and MONARCH, but did serve a small number of stale objects to clients (too small to show in the table). The MONARCH policy maintained strong cache consistency at the expense of sending server invalidations with virtually every server response. In object composition mode, heuristic policies generate more traffic, more unnecessary traffic, and serve more stale objects than both the Optimal policy and MONARCH. We also see that letting clients retrieve individual components forming a page results in more cacheable content than when the container is treated as monolithic. In this scenario, however, due to one RDyn component contributing the most bytes, only about 30% more content can be cached.

6.3 Corporate Web Presence

This scenario is characterized by the lack of objects that change on every access or every few minutes. Most of the objects on the page are RSt and remain unmodified for months, while a couple objects are RDyn and are updated daily. Examples of pages that this scenario models can be found at `ora.com` and `ieee.org`.

Simulation results for this scenario are shown in Table 3. In both monolithic and object composition modes, the MONARCH policy generates more unnecessary requests and induces the server to send more bytes than heuristic policies. This is due to the fact that MONARCH sends at least one request to the server on each page retrieval and this scenario has no frequently changing objects. On the other hand, this extra overhead allows MONARCH to maintain strong cache consistency. As in previous scenarios, breaking page into components allows to cache more content—at least 50% more in this case.

6.4 User Home Pages

Our last scenario models a category of pages that have no frequently changing objects at all. The container and all embedded objects change rarely (update ranges are on the order of months) and the page does not contain any components. The authors' home pages follow this model. We also consider a variation of this scenario that introduces a BoA access counter. For this scenario we increased the mean inter-arrival time from 15 minutes to 12 hours and also increased the length of the simulation from 3 weeks to 1 year. These increases produce more realistic arrival times and more object updates for the page in this scenario.

Simulation results for this scenario are shown in Table 4. For this scenario, both monolithic and

Table 2: Discussion/Storytelling Site Scenario

monolithic (9 objs): RDyn container, 1 BoA, 6 RSt and 1 St objects. composition (14 objs): RSt container, 1 BoA, 3 RDyn, 9 RSt and 1 St objects.							
Mode	Metric (per page retrieval)	Policy					
		Opt	M	H5	H10	AV	NV
Monolithic	Requests to Server	1.96	1.96	2.02	2.01	8.02	1.03
	Bytes from Server	53093	53107	53024	52938	54548	2945
	Stale Objects			0.00	0.00		1.00
	Unnecessary Validations			0.06	0.06	6.06	
Object Composition	Requests to Server	2.67	2.67	3.48	3.28	13.02	1.04
	Bytes from Server	36955	36979	37099	36989	39438	2947
	Stale Objects			0.03	0.06		2.97
	Unnecessary Validations			0.81	0.61	10.35	

Table 3: Corporate Web Presence Scenario

monolithic (10 objs): RDyn container, 3 RSt and 6 St objects. composition (13 objs): RSt container, 2 RDyn, 1 Per, 3 RSt and 6 St objects.							
Mode	Metric (per page retrieval)	Policy					
		Opt	M	H5	H10	AV	NV
Monolithic	Requests to Server	0.06	1.02	0.59	0.43	4.01	0.02
	Bytes from Server	1535	1766	1660	1561	2485	84
	Stale Objects			0.08	0.09		1.06
	Unnecessary Validations		0.96	0.54	0.38	3.96	
Object Composition	Requests to Server	0.06	2.04	0.87	0.57	6.03	0.03
	Bytes from Server	483	959	674	595	1915	138
	Stale Objects			0.13	0.25		2.15
	Unnecessary Validations		1.98	0.81	0.51	5.97	

object composition modes produce identical results since the page has no components. Results show that even though the MONARCH policy contacts the server on every page retrieval and generates substantially more traffic than the Optimal policy, it generates slightly fewer requests, unnecessary requests, and bytes than H5 policy, while maintaining strong cache consistency. With the introduction of a BoA access counter the performance of MONARCH virtually matches that of the Optimal policy.

This example, without the access counter, represents a page that does not have a mix of change characteristics and hence is not a prime candidate for our mechanism. The approach of designating one of the objects to be a manager that must be validated on each access is not unlike the idea of a volume lease [22], although we validate the manager and its volume on each access rather than combining it with server invalidation. If client caches want to avoid potentially unnecessary validations, they could use a heuristic approach in this case with some possibility of serving stale content.

7 Related Work

Previous work has examined volumes and ways of constructing them. Krishnamurthy and Wills studied site-wide volumes and volumes based on the first level prefix of object's path name [11]. They report that for sites with frequently changing resources the latter type of volume is more appropriate because site-wide volumes can generate a large number of invalidations. Cohen et al. [5] studied volumes based on access patterns and directory structure and proposed heuristics for thinning volumes. In our approach to volume construction, only objects composing a page are included in a volume, which results in tighter volumes, meaning that clients receive few invalidations that are not useful.

Prior work has also studied ways in which servers can provide strong cache consistency to client caches by sending object updates to all clients that accessed that object since its previous modification [14]. The number of invalidations that the server sends out and the number of clients that it has to keep track of can be reduced by using the concept of leases, proposed in [9] and studied by Yin et al. [22]. A subsequent study combined volume leases with server invalidation for handling dynamic Web workloads [23]. While these invalidation based approaches can maintain strong consistency, there are issues with the amount of per-client state that a server must maintain as well as with resynchronization when clients are disconnected from the server. The amount of state can be reduced through the use of volume leases for validating or invalidating a set of objects, but then clients must issue additional requests for renewing the lease. In addition, any server invalidation approach needs an infrastructure, such as multicast-based or hierarchical, to communicate the invalidations from the server to the client caches. In contrast, our request-driven approach requires no additional infrastructure; rather it simply uses other requests between the client and server to invalidate objects. Our approach also eliminates the need for servers to keep per-client state. Furthermore, in our approach, in case of network outages between clients and the server, clients receive an error condition instead of using a potentially stale object.

Challenger et al., in their work on the IBM 1998 Olympics Web site, had developed the Data Update Propagation (DUP) mechanism to automatically update cache contents at the origin severs when underlying data changes [3, 4]. DUP was intended to be used between servers and reverse proxies. Our approach is intended for the client-side caches, such as browser caches, forward proxies and CDN servers, but could also be deployed at reverse proxies.

Authors of the Cachuma caching system [24] advocate grouping dynamic pages into classes based on URL patterns and exploiting coarse-grain dependencies between the resulting

Table 4: User's Home Page Scenario

7 RSt objs.						
Metric	Policy					
(per page retrieval)	Opt	M	H5	H10	AV	NV
Requests to Server	0.04	1.02	1.03	0.56	7	0.01
Bytes from Server	144	380	382	269	1816	41
Stale Objects			0.18	0.43		4.15
Unnecessary Validations		0.98	0.99	0.52	6.96	
8 objs: 7 RSt and 1 BoA access counter						
Requests to Server	1.03	1.03	1.97	1.52	8	1.01
Bytes from Server	1321	1322	1547	1439	2993	1232
Stale Objects			0.18	0.45		4.1
Unnecessary Validations			0.94	0.49	6.97	

groups and underlying data. Servers invalidate a group of dynamic pages when underlying data changes. While the Cachuma approach may require servers to maintain less state than our finer-grained approach, we believe invalidating entire pages when only a portion of the underlying data changes is inefficient. Our approach in such situations invalidates only a single changing component, shared between all pages in a group.

8 Conclusions and On-going Work

In this paper we presented a Web object management approach that improves upon heuristic-based strategies for management of Web objects. We discussed the design and implementation of the prototype system and evaluated the performance of our approach using simulation. We show that for many categories of Web pages the performance of our approach is better than that of heuristic policies in terms of generated traffic while providing strong cache consistency. This consistency is obtained without requiring the server to maintain any per-client state. We also evaluated how existing and proposed object management policies are affected by exposing internal page components to clients. We show that although exposing constituent page components to clients tends to generate more requests that reach the server, the amount of usable cached content increases by 30%–50%.

Our on-going work is proceeding in four directions. First, we are in the process of extending WOMS to handle wider variety of situations, such as when object's change characteristic changes during the simulation, and to handle more object management policies, such as server invalidation with leases [14, 22].

Second, we are working on making WOMS drive the MONARCH prototype system. The idea is for WOMS to generate real Web content (which it can already do) that the MONARCH server can access and serve to clients and to also convert simulated requests into real HTTP requests and send them to the MONARCH Proxy. Our main interest in connecting the two systems is to study the scaleability of the entire MONARCH system, and of the volume manager in particular, over a range of categories of Web content. We are also evaluating the possibility of further thinning volumes by grouping objects shared by many pages at a site into a separate volume.

Currently our simulator uses synthetic content with characteristics based on informal analysis of a set of Web sites. To make the simulated

content more realistic, we plan to collect real content from a variety of Web sites over a period of time and then use the collected content to drive the simulator. Such an approach would allow us to evaluate the simulated object management policies on the same content that a real cache is required to handle. It would also allow us to simulate the behavior of a real cache, including the handling of the HTTP cache control directives. Using real content can also facilitate the study of multiple pages at a site, and the study of how to handle objects that are shared between multiple pages.

Finally, we are extending the basic content assembly functionality of our prototype system with capabilities for caches to dynamically create such content as rotating ad banners, and for handling content personalization. While similar systems have been proposed and even deployed [8], we are interested in evaluating how this addition may affect our approach to Web object management.

9 Acknowledgments

The authors thank the anonymous reviewers for their detailed and helpful comments on technical aspects of the paper and on the presentation. The authors also thank Janet Burge for reading draft and final versions of the paper and providing corrections and suggestions that improved the paper.

References

[1] Martin Arlitt and Tai Jin. A Workload Characterization Study of the 1998 World Cup Web Site. *IEEE Network*, May/June 2000.

[2] Vincent Cate. Alex - a Global Filesystem. In *Proceedings of the USENIX File Systems Workshop*, pages 1–12, May 1992.

[3] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of the IEEE Infocom '99 Conference*, New York, NY, March 1999. IEEE.

[4] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic web content. In *Proceedings of the IEEE Infocom 2000 Conference*, Tel Aviv, Israel, March 2000. IEEE.

[5] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. In *ACM SIGCOMM'98 Conference*, September 1998.

[6] John Dilley. The Effect of Consistency on Cache Response Time. *IEEE Network*, May/June 2000.

[7] DOM: Document Object Model. W3C Working Draft. `http://www.w3.org/DOM/`.

[8] Edge Side Includes. `http://esi.org/`.

[9] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 202–210, December 1989.

[10] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. Preliminary measurements on the effect of server adaptation for web content delivery. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, Marseille, France, November 2002.

[11] Balachander Krishnamurthy and Craig E. Wills. Piggyback server invalidation for proxy cache coherency. In *Seventh International World Wide Web Conference*, pages 185–193, Brisbane, Australia, April 1998.

[12] Balachander Krishnamurthy and Craig E. Wills. Analyzing Factors That Influence End-to-End Web Performance. In *Proceedings of the Ninth International World Wide Web Conference*, Amsterdam, Netherlands, April 2000.

[13] Dan Li, Pei Cao, and Mike Dahlin. WCIP: Web Cache Invalidation Protocol. Internet Draft. `http://www.wrec.org/Drafts/draft-danli-wrec-wcip-00.txt`.

[14] Chengjie Liu and Pei Cao. Maintaining strong cache consistency in the world-wide web. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997.

[15] Jeffrey Mogul. An alternative to explicit revocation?, January 1996. `http://www.roads.lut.ac.uk/lists/http-caching/1996/01/0002.html`.

[16] Erich M. Nahum. WWW Workload characterization work at IBM Research. In *Web Characterization Workshop*, Cambridge, MA, November 1998. World Wide Web Consortium. `http://www.w3.org/1998/11/05/WC-workshop/Papers/nahum.html`.

[17] The Slashdot Code. `http://slashcode.com/`.

[18] Duane Wessels. Squid Internet Object Cache. `http://squid.nlanr.net/Squid/`.

[19] Craig E. Wills and Mikhail Mikhailov. Towards a better understanding of Web resources and server responses for improved caching. In *Eighth International World Wide Web Conference*, Toronto, Canada, May 1999.

[20] Craig E. Wills and Mikhail Mikhailov. Studying the Impact of More Complete Server Information on Web Caching. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[21] Craig E. Wills, Mikhail Mikhailov, and Hao Shang. *N* for the Price of 1: Bundling Web Objects for More Efficient Content Delivery. In *Tenth International World Wide Web Conference*, Hong Kong, May 2001.

[22] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Using leases to support server-driven consistency in large-scale systems. In *Proceedings of the 18th International Conference on Distributed Systems*. IEEE, May 1998.

[23] Jian Yin, Mike Dahlin, Lorenzo Alvisi, Calvin Lin, and Arun Iyengar. Engineering server driven consistency for large scale dynamic web services. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001.

[24] Huican Zhu and Tao Yang. Class-Based Cache Management for Dynamic Web Content. In *Proceedings of the IEEE Infocom 2001 Conference*, Anchorage, Alaska USA, April 2001.