Caching Dynamic Content on the Web

By

Wenzhong Chen

A thesis submitted to the

School of Computing

in conformity with the requirements

for the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

June 2003

0-612-81052-6

Abstract

Caching dynamic content is an important issue due to the growing number of non-cachable dynamic Web documents. Previous research has focused on server-side dynamic caching techniques, which are beneficial in reducing server resource demands. Proxy-side caching of dynamic content, on the other hand, reduces bandwidth consumption and download latency and increases hit ratio.

This thesis proposes a scheme to efficiently cache dynamic content at the proxy side. The scheme identifies two kinds of dynamic pages, called eager-update pages and lazy-update pages, and uses different strategies to deal with each type. For eager-update pages, the web server always "pushes" the newest data to the proxy after updates to the dynamic page content. For lazy-update pages, proxies always "pull" the newest data from the web server when clients request it. We decrease the data transferred on the web by using class-based delta-encoding to transfer data from the web server to the proxy.

A extensive simulation model has been developed to study the performance of our proposed scheme. We compare our scheme with delta-encoding and LRU. We have tested a variety of traffic conditions and measurement parameter settings in the simulation. Our results show that our scheme can achieve a higher hit ratio and improve the network latency.

Acknowledgements

I would like to express my gratitude to my supervisors Professor Patrick Martin and Professor Hossam Hassanein, for their excellent guidance, great advices and support.

I would like to thank to Wendy Powley, Wenhu Tian, Xilin Cui and all the other members in the Database System Laboratory at Queen's University for their suggestions, assistance, comments and friendship.

Finally, I would also like to thank the School of Computing at Queen's University for offering me such an invaluable opportunity to pursue my Master's Degree. The financial support provided by Communications and Information Technology Ontario (CITO) is appreciated.

Table of Contents

1. Introduction 1

2. Related Work 7
   2.1 Static content caching models and schemes 7
   2.2 Dynamic content caching techniques 11
   2.3 Dynamic content caching schemes 18
   2.4 Summary 21

3. Eager-update Page Dynamic Caching scheme 23
   3.1 Dynamic page classification 25
   3.2 HTTP extension 29
      3.2.1 Eager-update page communication 31
      3.2.2 Lazy-update page communication 34
   3.3 Class-based delta-encoding technique 37
      3.3.1 Grouping Documents into Classes 39
      3.3.2 Choosing a Base-file 41
   3.4 Extended LRU Algorithm 41
   3.5 Summary 44

4. Performance Evaluation 45
   4.1 Simulation Model 45
      4.1.1 Network Model 45
      4.1.2 Workload generation 47
      4.1.3 Simulation Parameter Settings 50
   4.2 Simulation Results 51
      4.2.1 HTTP request rate and response time 51
      4.2.2 Network link delay and response time 54
      4.2.3 Changing workload and Response Time 57
      4.2.4 Eager Page HIT and Response Time 60
      4.2.5 Hit Ratio 63
      4.2.6 Web server cost analysis 64
   4.3 Summary 65
5. Conclusion 68

6. References 72

Appendix A. Delta-encoding Algorithms 79

Appendix B. HTTP Extensions 89

Appendix C. The Flow Charts, Simulation Software Structure and Pseudo-Code of EPDC 93

Appendix D. Confidence Intervals 103

Vita 105

List of Figures

Figure2. 1 Hierarchical Architecture for Web Caching 8

Figure2. 2 Distributed Architecture for Web Caching 9

Figure2. 3 Transparent Proxy Web Caching with Redirection 10

Figure2. 4 A simple dynamic page 12

Figure2. 5 A dynamic page with a loop 13

Figure2. 6 Template/bindings encoding for the page from Figure 2.4 16

Figure2. 7 Template and bindings encoding for a search-engine response 17

Figure3. 1 network model 24

Figure3. 2 HTTP IMS request for the not modified page 27

Figure3. 3 Proxy side algorithm to process client requests 28

Figure3. 4 Eager-update page HTTP request from proxy to Web server for classification 32

Figure3. 5 Eager-update page HTTP request from Web client to Web server 34

Figure3. 6 lazy-update page HTTP request from Web client to Web server 36

Figure3. 7 Using delta-encoding for Web-documents 38

Figure3. 8 Add page function of the Extended LRU algorithm 43

Figure4. 1 Network Model 46

Figure4. 2 Zipf-like Distribution[39] 48

Figure4. 3 Average response time at link delay = 90 ms 53

Figure4. 4 Average response time at link delay = 180 ms 53

Figure4. 5 Average response time at link delay = 360 ms 54

Figure4. 6 Page arrival rate = 30 messages/min at proxy 56

Figure4. 7 Workload changing point 58

Figure4. 8 Response Time Difference 58

Figure4. 9 Workload Changing Point and Response Time Difference 59

Figure4. 10 Eager Page HIT Response Time 61

Figure4. 11 Eager Page HIT Response Time 62

Figure4. 12 Hit Ratio 63

Figure C. 1 HTTP GET request in EPDC scheme 94

Figure C. 2 HTTP GET request in Delta-encoding scheme 96

Figure C. 3 HTTP GET request in LRU scheme 97

Figure C. 4 The algorithm for proxy receiving a message from Webserver 100

Figure C. 5 The algorithm for Web server receiving an HTTP-message 101

Figure C. 6 The algorithm for Delta server receiving an HTTP message 102

List of Tables

Table 3. 1 Notation and Parameters 26

Table 3. 2 URL-parts for differently organized Web-sites 40

Table 4. 1 Simulation parameters 51

Table 4. 2: Average response time for different link delays 55

List of Acronyms

CARP	Cache Array Routing Protocol
CGI	Common Gateway Interface
CSS	Content SmartSwitch
DUP	Data Update Propagation
EPDC	Eager-update Page Dynamic Caching
HTTP	Hypertext Transfer Protocol
ICP	Internet Cache Protocol
IMS	If Modified Since
LB-L5	Load Balance Layer 5
LRU	Least Recently Used
MRT	Minimum Response Time
QoE	Quality of Experience
RCS	Revision Control System
RTT	Round Trip Time
SCCS	Source Code Control System
SSH	Secure Shell
TCP	Transmission Control Protocol
TTL	Time to Live
URL	Uniform Resource Locator
WWW	World Wide Web

Chapter 1

Introduction

The World Wide Web has evolved into the most widespread infrastructure for Internet-based information systems in the last decade. As with traditional information systems, the response time for individual interactions becomes critical in providing the right Quality of Experience (QoE) for a Web user. Many WWW clients are faced with congested networks and long propagation delays when they try to access content. Techniques have been developed that reduce the bandwidth consumption and improve the latency for this environment.

Web servers provide two types of data: static data from files stored at a server and dynamic data, which are constructed by programs that execute at the time a request is made. Responses from search engines or stock-quote sites provide typical examples of dynamic data. The use of Web-caches for reducing bandwidth consumption and download latency has been quite successful in the past. However, traditional Web caching is applicable only to static documents, or to documents that change in large timescales. Since the proportion of dynamic versus static documents is increasing day by day, current caching solutions have reached a point where they cannot offer significant performance improvements unless they incorporate a mechanism to "cache" dynamic documents. In particular, no matter what the replacement algorithms are, the cache size and the user population serviced by the cache, proxy-cache hit rates are usually around 40% [38]. However, if proxy-caches are equipped with mechanisms that exploit redundancy from all documents, static and dynamic, hit rates could be up to 80% [38].

Previous research on caching dynamic content mostly concerns server-side caching techniques. Caching dynamic pages at server sites can reduce server computing resource demands and simplify page invalidation. A major problem with caching dynamic content is to ensure that strong consistency is maintained between cached pages and underlying data objects. Vahdat et al. [36] and Holmedhl et al. [41] have proposed invalidation techniques using file operation interception or TTL for a class of applications. Iyengar et al [24], [30] propose a fine-grain dependence-based approach that allows applications to explicitly issue invalidation messages to a cache. Zhu at al [40] propose a complementary solution for applications that require coarse-grain cache management. They design and implement a caching system that integrates the above techniques, with standard Web servers, which adds dynamic page caching capability to the Web server. Thus, it can offload servers from computing dynamic documents on the fly and reduce user latency as well.

Proxy-side caching of dynamic content reduces bandwidth consumption, and download latency and increases hit ratio. The Active Cache scheme [16], by Cao et al, allows servers to supply cache applets to be attached with documents, and requires proxies to invoke cache applets upon cache hits to furnish the necessary processing without contacting the server. Even though this approach is efficient for tasks like rotating advertisements, it is not efficient as a general mechanism that fully exploits temporal correlation in dynamic traffic. It does not address invalidation of dynamic pages and the cost is high because it requires starting up a new Java process for every request.

A variety of methods to exploit the redundancy of dynamic traffic have been proposed recently. Delta-encoding was introduced by Banga et al. [28], and independently in a more restricted context by Housel and Lindquist [16]. The idea is that both ends of a slow link store the same snapshot of a Web-document, called a base-file, upon a request for that document. The server end gets the current snapshot of the document from the Web-server, computes the difference between the current and the stored snapshot, called the delta, and then sends the delta over the slow link. When the client receives the delta it reconstructs the current snapshot by combining the delta and the stored snapshot and it sends the response to the client.

Mogul et al. [37] use dynamic traces and show that delta-encoding can provide remarkable improvements in response size and response delay for an important subset of HTTP content types. Delta-encoding can provide an efficient solution to the problem of exploiting temporal correlation in dynamic traffic; however, it cannot solve the scalability problems at the server side. The storage requirements at the server-side grow enormously due to the increasing number of dynamic documents.

This problem becomes even more intense due to personalized Web-documents, since delta-encoding stores each personalized version of a dynamic document. Psounis et al. [12] introduce class-based delta-encoding, a scalable scheme to perform delta-encoding on dynamic Web traffic. The idea is to group documents into classes, and store one document per class on the server-side. The proposed scheme exploits both temporal correlation in a dynamically evolving document, and spatial correlation among different documents.

With caching, a dynamic page must be generated by the Web server for each client request. The dynamic page generation on the Web server is a time consuming process. Most existing dynamic content caching schemes are concerned with offloading the Web server cost. In this thesis we propose a scheme called Eager-update Page Dynamic Caching (EPDC) that efficiently caches dynamic content on the Web. EPDC not only offloads the Web server cost but also decreases the client response time. The scheme identifies two kinds of dynamic pages, called eager-update pages and lazy-update pages, and uses different strategies to deal with each type. An eager-update page is defined as a page whose request rate is faster than its update rate. A lazy-update page is defined as a page whose request rate is slower than its update rate. By identifying these two kinds of pages, we can decrease the number of times an eager-update page must be generated, which reduces the Web server cost and reduces the download response time. We decrease the data transferred on the Web by using delta-encoding to transfer data from the Web server to the proxy.

For eager-update pages, the Web server always “pushes” the newest data to the proxy after updates to the dynamic page content. The proxy always has the newest snapshot of an eager-update page as the base-file. The Web server only sends the delta when the dynamic page content is updated.

For lazy-update pages, proxies always “pull” the newest data from the Web server when clients request it. The proxy adopts the class-based delta-encoding that has a base file that groups different pages with the same URL pattern so that several dynamic pages share the same base file. Each time the proxy passes the request to the Web server, the Web server sends the delta, which differs between the page and the sharing base file, to the proxy.

The EPDC scheme has advantages of both server side and proxy side dynamic caching. Not only does EPDC improve the user response time, cache hit ratio, and reduce network bandwidth and network latency, but it also offloads the Web server processing time.

In this thesis, we propose and study the performance of EPDC. First, we identify two kinds of pages by their request and update rate (Eager and Lazy pages). Second, we extend the HTTP protocol to efficiently exchange page class information between proxies and Web servers. Third, we extend the LRU replacement algorithm to handle our page classes. Finally, we design and implement a simulation that integrates the above techniques and evaluates EPDC performance. The results show our scheme outperforms the delta-encoding and LRU schemes.

The rest of the thesis is organized as follows. In Chapter 2 we provide a review of related static and dynamic content caching technologies. Chapter 3 presents an overview of the proposed Eager-Page Dynamic Caching scheme followed by a discussion of the design decisions. A performance evaluation of EPDC is presented in Chapter 4. The adopted simulation model is described. Simulation experiments are conducted in order to study the effect of network link delay and HTTP request rate on the performance of EPDC. Results show that our scheme can achieve higher hit ratios and improves network latency in comparison with the delta-encoding and the LRU schemes. Chapter 5 concludes the thesis, and provides some suggestions for further research.

Chapter 2

Related Work

In this chapter we provide an overview of various Web caching techniques. Section 2.1 briefly introduces static content caching models and algorithms. Section 2.2 introduces two dynamic content caching techniques: based-instance caching and template caching. Section 2.3 introduces server side and proxy side schemes for caching dynamic content. Section 2.4 gives a summary of this chapter.

2.1 Static content caching models and schemes

Three popular caching schemes used for Web caching are hierarchical, distributed and switch-based transparent Web caching systems. Traditional hierarchical cache server architectures such as Squid [12] have two levels. Leaf proxies represent organizational proxies. Second-level proxies are connected to a backbone. A leaf may form sibling relations with some other leaves and child-parent relations with multiple roots, as shown in Figure 2.1. When a child cache does not satisfy a request, the request is redirected to a sibling cache. If the document is not found at the siblings, the request is then forwarded to the parent level. This process goes upwards until the data is located or the root cache server fetches the data from the Web server. The cache servers then send the data down the hierarchy and each cache along the path stores the data. Hierarchical Web caching was first proposed in the Harvest project [36]. Other examples of hierarchical caching include Adaptive Web caching [24] and

Access Driven cache [26]. The hierarchical architecture can save bandwidth, but its multiple levels introduce additional delays and redundant copies of documents.

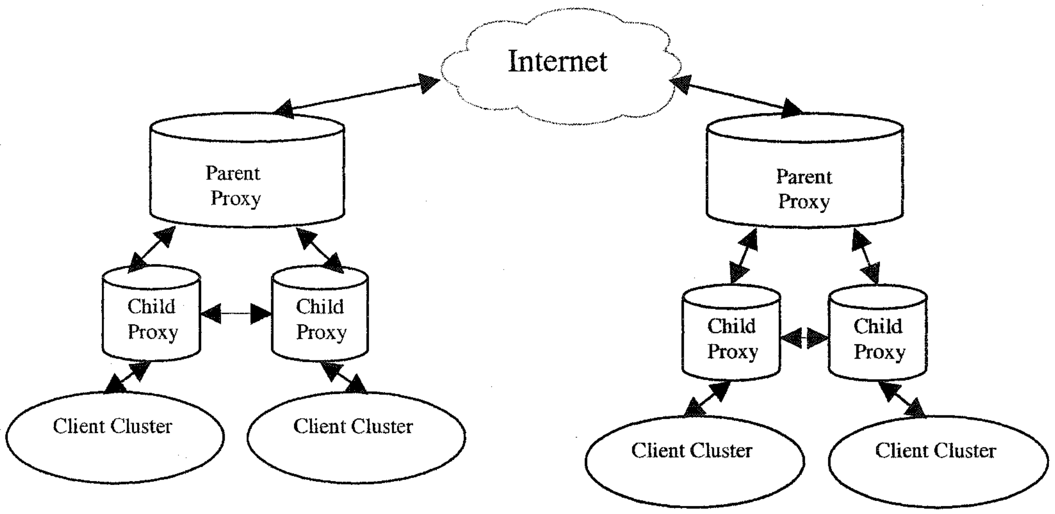


Figure2. 1 Hierarchical Architecture for Web Caching

In distributed Web caching systems, no intermediate caches are set up and there are no parent-child relationships between the proxies. There is only a group of cache servers cooperating with each other, serving each other's misses. In order to decide from which cache server to retrieve a document, cooperating cache servers keep metadata information about the content of every other cache. The structure of the distributed caching model is shown in Figure 2.2.

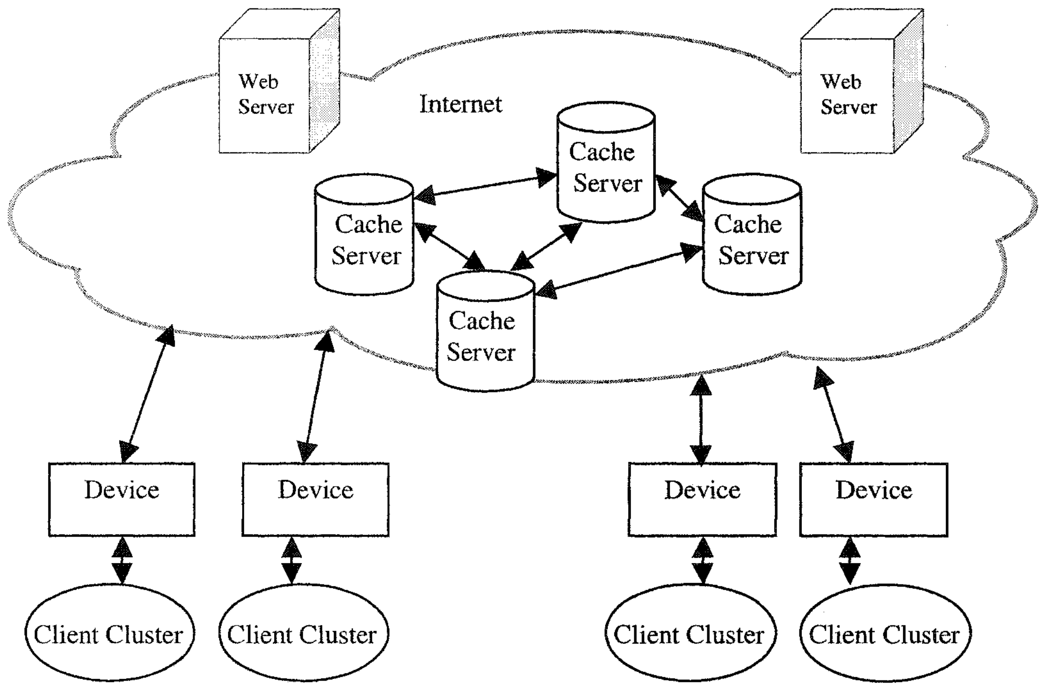


Figure2. 2 Distributed Architecture for Web Caching

There are several approaches to distributed caching. The Harvest [36] group designed the Internet Cache Protocol (ICP), which supports discovery and retrieval of documents from neighbour caches and parent caches. Another approach to distributed caching is the Cache Array Routing Protocol (CARP)[37], which divides the URL-space among an array of loosely-coupled caches and lets each cache store only the Web objects whose URLs are mapped to it. Distributed Web caching systems rely on replicated objects and services to improve performance and reliability. All cache servers are employed at the same level so distributed Web caching systems overcome the drawbacks of hierarchical Web caching systems. Moreover, they have better fault tolerance, distribution of server loads and client performance since they bring cache servers closer to Web clients.

In the switch-based transparent Web caching system, a switch sits in the data path between the Web clients and the server cluster. It intercepts the Web traffic and transparently redirects the HTTP requests to different cache servers or to the Web server. The structure of the switch-based Web caching model is shown in Figure 2.3.

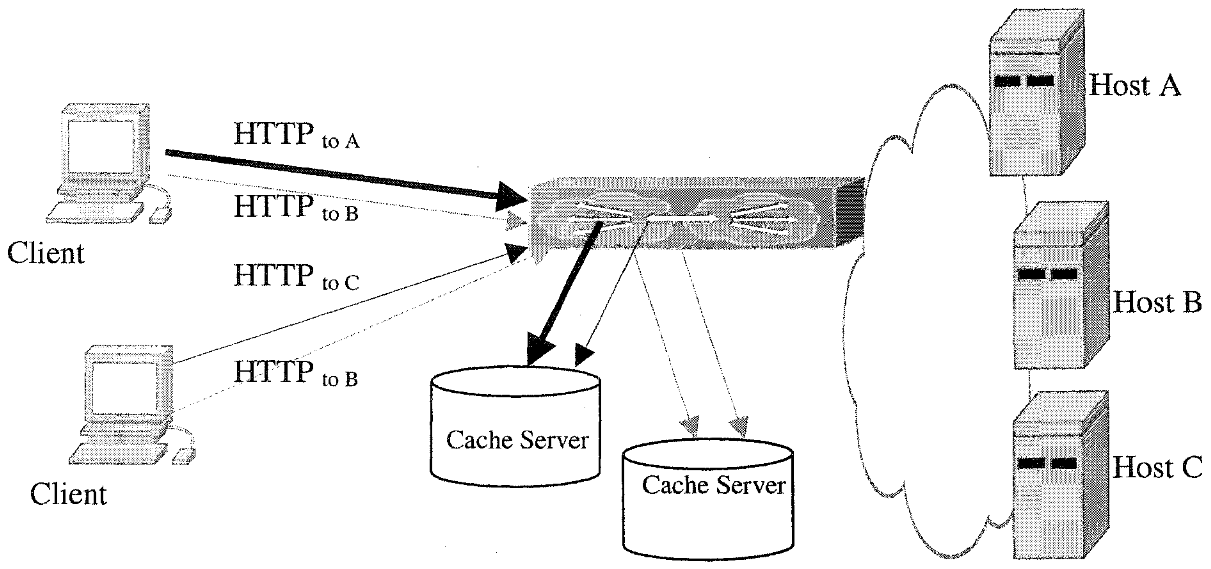


Figure2. 3 Transparent Proxy Web Caching with Redirection

Transparent Web caching makes the configuration of the caching system easier. Switches can rapidly process and forward the packets. This switching-based transparent Web caching technique can use content-aware Layer 5 switches in a distributed Web caching system with enhanced cache cooperation [1] [22]. The switches perform content checking based on Layer 5 header information of the HTTP request packets. A HTTP request is redirected by switches to the cache server that can best service the request. One example is the Arrowpoint Content SmartSwitch

(CSS) [3]. On the client side CSS can be configured to redirect static HTTP requests to a cache server cluster since it can distinguish among different "higher-level" protocols, like HTTP [20] and The Secure SHell (SSH) remote login protocol [32], and divert them to the appropriate server or group of servers that service the type of requested content.

Liang [22] proposed a fully distributed Web caching scheme that extends the capabilities of Layer 5 switching to improve the response time and balance cache server workload. In LB_L5, a Layer 5 switch selects the best server based on cache content, cache server workload, network load and the HTTP header information. LB_L5 does not guarantee the minimum response time so Zhou [39] proposed the Minimum Response Time (MRT) switching-based Web caching scheme to reduce the HTTP request response time and balance the workload among the caches based on a combination of request content, cache server content, network latency and server workload.

2.2 Dynamic content caching techniques

Dynamic objects are generated by programs that run on servers every time the corresponding objects are accessed. Responses from search engines or stock-quote sites provide typical examples of dynamic objects. It turns out that not all information in a dynamic page is different for each access. Frequently, a large portion of the page is devoted to formatting specifications, headers and footers, and other information that does not change across accesses. A cache-friendly encapsulating implementation creates embedded objects containing the dynamic variables so that the dynamic page itself is cacheable and only the embedded objects have to be downloaded on every access. Although encapsulating dynamic portions of a page into embedded objects is a useful technique in many situations, it falls short for two kinds of pages. The first includes pages with multiple bits and pieces of dynamic data scattered around the page. A typical example is shown in Figure 2.4.

```
<HEAD>
<TITLE>Michael Chang<TITLE>
<BODY>
......
This page has been active for 35 days. It has been accessed 44 times. Avg. times
accessed per day: 1.27
</BODY>
```

Figure2. 4 A simple dynamic page

This page contains three dynamic items. Including all dynamic items into a single embedded object would make this object encompass most of the page, defeating the purpose of the technique. On the other hand, encapsulating each dynamic item into a separate object entails multiple downloads of small objects from the server, which increases the server load.

The second kind of page that is unsuitable for encapsulation is represented by responses from search engines. Figure 2.5 shows the structure of a typical search engine response.

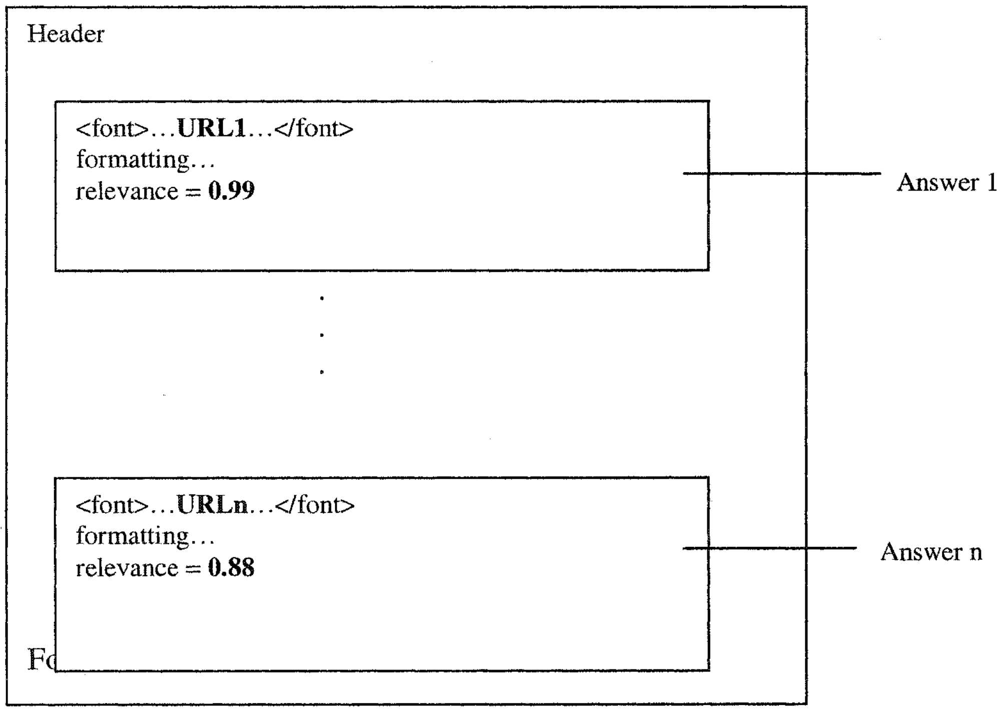


Figure2. 5 A dynamic page with a loop

In this figure, the dynamic page starts with a header that typically identifies the search engine, then lists a number of answers and concludes with a footer. Each answer includes the URL of a page that the search engine believes is relevant to the query and a measure of the page relevance. Dynamic answers typically take up most of the content on these dynamic pages. The encapsulation technique would require including all answers in a dynamic object, which would cover most of the information in the page, which defeats the purpose of encapsulation.

From the above examples we find that page fragments corresponding to each dynamic variable contain, to a large extent, static information. There are two main techniques allowing caching of static page portions: base-instance caching and template caching. Consider a Web server sending a dynamic page to a client. If both share the same instance of the page, the server could use delta encoding [1]**[36]** to reduce the amount of data it needs to send. Assume that the client previously performed the query "tall green trees" on a search engine. If it now wants to perform the query "caching dynamic objects," it can send the engine an indication that it already has the result of the "tall green trees" query. The search engine could now send just the delta between the first and second query results to the client.

Unfortunately, delta-encoding requires the server to store instances of the pages it previously sent to each client. Given the number of clients accessing popular servers and the load on these servers, this approach is clearly impractical. Base-Instance caching means the server can designate a certain base instance of the dynamic page and arrange that every client has the same base page instance. The server then needs to store only one base instance to be able to delta-encode future responses.

The Web Express product from IBM **[16]** implements base-instance caching. For example, let the server designate an instance of the dynamic page to be base instance

B that is the result of the query “tall green trees”. When a client sends its first request Q1 to the server, which is the query of “caching dynamic objects”, the server sends it three pieces of information:

- the instance B of the dynamic page, together with its Etag (Entity tag, a unique identifier for a particular instance of an object);
- the indication that B is the base instance; and
- the delta of the page instance generated for request Q1 over the base instance.

When the client receives this response, it reconstructs the page instance Q1 by applying the delta to base instance B and caches the base instance. For subsequent accesses, the client includes the identity of its cached base instance. If the server still has B as the base instance, it will send only the deltas back. If the base instance at the server has changed, the server will have to send both the new base version and the delta again. Consider the examples in Figures 2.4 and 2.5. In both examples, if the client already caches the base file, it obtains a small delta of the current over the base version on each access, and it does so in a single access to the server. For the example in Figure 2.5, if the current version of the page has a greater number of answers, the delta will contain the extra answers in their entirety. In practice, some delta-encoding algorithms produce deltas that are compressed **[24]**.

Template caching [23] separates dynamic and static portions of the page explicitly. The static portion is augmented with macro-instructions for inserting dynamic information. The static portion together with these instructions is called the page template. Macro-instructions use macro-variables to refer to dynamic information.

The dynamic portion contains the bindings of macro-variables to strings specific to the given access. Before rendering the page, the client expands the template according to the macro-instructions and using the bindings that are downloaded from the server for each access. The rationale behind template caching is that the client caches the template and downloads only the bindings for every access instead of the entire page. Let us consider the two examples in Figures 2.4 and 2.5 again. The template and bindings encoding for these pages are shown in Figures 2.6 and 2.7, respectively:

```
<HEAD>
<TITLE>Michael Chang</TITLE>
<BODY>
...
This page has been active
For <VAR daysactive> days.
It has been accessed <VAR count>
Times.
Avg. times accessed per day:
<VAR count/daysactive>
</BODY>
```

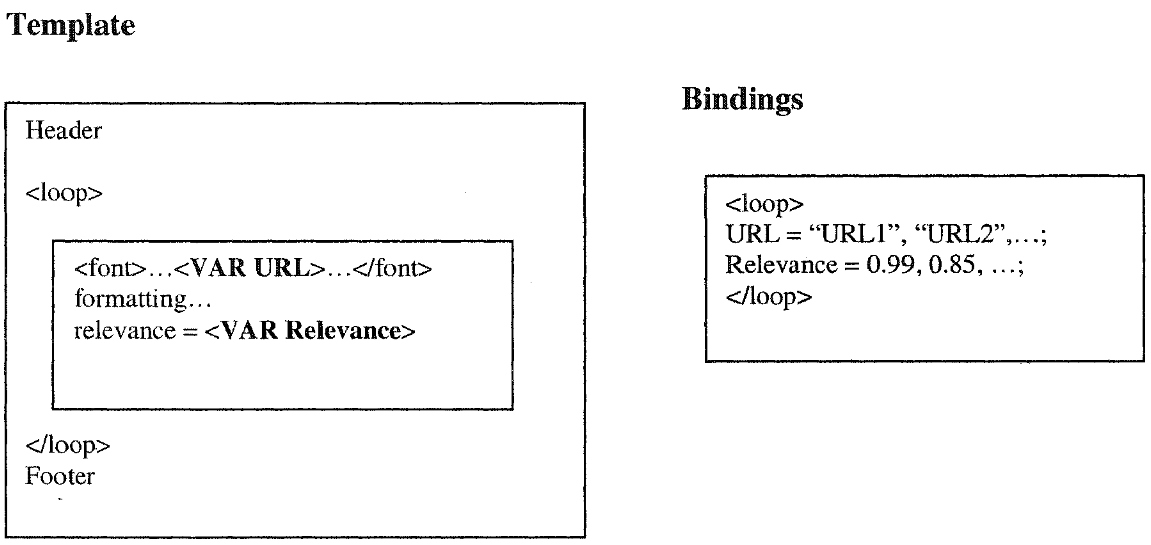
(a) The template

```
        <HTML>
<TEMPLATE HREF="query.hpp">
        <BODY>
daysactive = 35;

        count = 44;
</BODY>
        </HTML>
```

(b) The bindings

Figure2. 6 Template/bindings encoding for the page from Figure 2.4



Template

```
Header

<loop>

    <font>...<VAR URL>...</font>
    formatting...
    relevance = <VAR Relevance>

</loop>
Footer
```

Bindings

```
<loop>
URL = "URL1", "URL2",...;
Relevance = 0.99, 0.85, ...;
</loop>
```

Figure2. 7 Template and bindings encoding for a search-engine response

The binding encoding groups all dynamic items in a single object that allows the client to obtain all dynamic items in a single download. The dynamic items are scattered throughout the page according to template macro-instructions (such as VAR tags in Figure 2.6a).

The template in Figure 2.7 uses a single loop macro-instruction to describe all answers in the response. The corresponding loop construct in the bindings contains a list of values for every macro-variable in the loop, one value per answer. When the actual page is constructed, the macro preprocessor expands the loop body as many times as the number of values assigned to macro-variables.

Template caching is similar to various mechanisms used on the server side such as Java Server Page or Active Server Page, which provide complex languages for expanding the page template prior to sending it to the client. The differences are that template caching expands templates at the client rather than the server, and it obtains all dynamic information in one download to keep the number of interactions with the Web server to a minimum. Several companies now implement template caching at surrogate servers, so that surrogates reconstruct the actual pages. Akamai refers to template caching by surrogates as edge-side includes [23] and leads a group of companies in defining a new template language for this purpose.

The big advantage of base-instance caching is that it is transparent to Web developers who create content. Only underlying server and client platforms must be extended. The benefit of template caching is that the template need not be generated at the time of access. Instead, it can be pre-computed and possibly pre-compressed once. Both base-instance and template caching techniques are based on delta encoding. Base-instance caching adds the overhead of computing the delta to the critical path of the request processing. The delta-encoding algorithms ask the server to compute the page instance, then compute the delta. This prevents the server from pipelining the page to the client as it is being produced, which may increase latency.

2.3 Dynamic content caching schemes

Server side dynamic caching schemes are concerned with the situation where the bottleneck is the server's CPU. The most popular scheme is to set up a cache server at the server side. It can cache the newest snapshots of dynamic pages so the major problem is how to ensure that consistency be maintained between cached pages and underlying data objects. Challenger et al [30] developed and implemented an algorithm called Data Update Propagation (DUP) that maintains data dependence information between cached objects and the underlying data in the form of a graph. When the system becomes aware of a change to underlying data, graph traversal algorithms are applied to determine what cached objects are affected by the change. Cached objects that are found to be obsolete are either invalidated or updated. They used this technique at the official Web site for the 1998 Olympic Winter Games and achieved great performance and high availability. This approach has two drawbacks. First, it requires that the server application be rewritten to take advantage of the cache, and this can be a nontrivial task. Second, for every dynamic request, the Web server still must start the application, even if only to return a cache hit. The operating system overhead for this call is significant.

Holmdahl et al [41] developed a distributed Web server called Swala, in which the nodes cooperatively cache the results of CGI requests. They use a two-level cache table consistency protocol and a replicated global cache directory to maximize the system performance and minimize overhead in responding to dynamic Web requests.

Iyengar et al [24][30] propose a fine-grained dependency-based approach that allows applications to explicitly issue invalidation messages to a cache. It needs to rely on maintaining a fine-grain graph that specifies dependence among individual Web pages and underlying data sets. This kind of graph will grow enormously and will be hard to maintain when the sites host arbitrarily large numbers of dynamic pages.

Zhu et al [40] propose a complementary solution for applications that require coarse-grained cache management. The key idea is to partition dynamic pages into classes based on URL patterns so that an application can specify page identification and data dependence, and invoke invalidation for a class of dynamic pages. To make this scheme time-efficient with respect to space requirements, lazy invalidation is used to minimize slow disk accesses when Ids of dynamic pages are stored in memory with a digest format. Selective precomputing is further proposed to refresh stale pages and smooth load peaks. A trie data structure is developed for efficient URL class searching during lazy or eager invalidation. This paper presents the design and implementation of a caching system called Cachuma that integrates the above techniques, runs in tandem with standard Web servers, and allows Web sites to add dynamic page caching capability with minimal changes.

Proxy-side caching dynamic content reduces bandwidth consumption, network latency and increases the hit ratio. Smith at al [30] propose a new protocol to allow individual content-generating applications to exploit query semantics and specify how their results should be cached and delivered. Applications may declare a dynamic request to be identical, equivalent, or partially equivalent. In the first two cases cached results are up to date, while in the third case content can be immediately delivered as an approximate solution while actual content is generated and delivered. This is not a complete solution to the problem of exploiting temporal correlation in dynamic traffic in that there exist dynamic documents that share data, yet their redundancy cannot be exploited with this simple scheme.

The Active Cache scheme [16] allows servers to supply cache applets to be attached with documents, and requires proxies to invoke cache applets upon cache hits to furnish the necessary processing without contacting the server. Even though this approach is efficient for tasks like rotating advertisements, it is not efficient as a general mechanism that fully exploits temporal correlation in dynamic traffic. It does not specifically address invalidation of dynamic pages and the cost is high since it requires starting up a new Java process for every request.

Douglis et al. [23] separate the static and dynamic portions of a document. They cache static parts as usual, while dynamic parts are obtained on each access from the server. They extend HTML to support this scheme. From their simulations, the size of network transfers is typically 2 to 8 times smaller than the original sizes. Compared to delta-encoding, this idea is simpler but less adaptable.

2.4 Summary

In this chapter, we reviewed static and dynamic Web caching techniques. In Section 2.1, we described the hierarchical, distributed and switch-based Web caching models and their advantages and disadvantages. In Section 2.2, we described two basic dynamic caching techniques: base-instance and template caching. Base-instance caching allows dynamic content to benefit from caching when instances of the content do not differ drastically from one another. Template caching also addresses the cacheability of dynamic content and allows a content provider to explicitly separate truly dynamic data from static page templates. In Section 2.3, we introduce various dynamic content caching schemes at server side and proxy side. The server side dynamic caching schemes are concerned with the situation in which the bottleneck is the server's CPU. They do not care about the network latency and bandwidth. The proxy side dynamic caching schemes reduce bandwidth consumption, network latency and increase hit ratio. A major problem with dynamic caching is to ensure that strong consistency is maintained between cached pages and underlying data objects. How to deal with consistency at the proxy side is an interesting topic and we propose our Eager Page Dynamic Content caching scheme in the following chapter.

Chapter 3

Eager-update Page Dynamic Caching scheme

A dynamic page is generated by a Web server for each client request. The dynamic page generation on the Web server is a time consuming process. For decreasing the number of times a dynamic page is generated, Eager-update Page Dynamic Caching (EPDC) scheme dynamically identifies two kinds of pages: eager-update pages and lazy-update pages. It keeps strong consistency of eager-update pages between cached pages and underlying data objects. For eager-update pages, the Web server always "pushes" the newest data to the proxy after updates to the dynamic page content. For lazy-update pages, proxies always "pull" the newest data from the Web server when clients request it. EPDC extends HTTP, the most popular Web protocol, to support communication between cache servers and Web servers. EPDC also uses class-based delta-encoding [26] to reduce network bandwidth and latency.

The EPDC scheme has advantages over both server side and proxy side dynamic caching. Not only does EDPC improve the user response time, cache hit ratio, and reduce network bandwidth and latency, but it also offloads the Web server processing time. The network model for EPDC is shown in Figure 3.1.

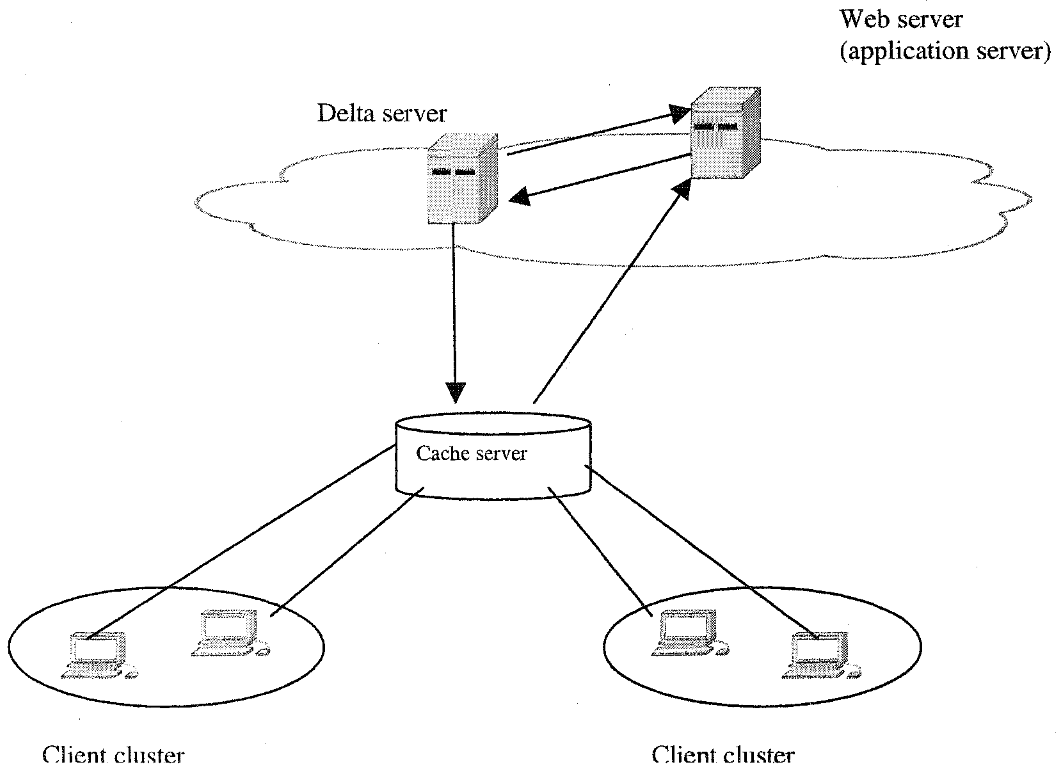


Figure3. 1 network model

Proxy cache servers accept HTTP requests from a cluster of clients and send HTTP requests to the Web server. The Web server (application server) generates new pages and passes them to the delta server to generate deltas then returns responses to proxies.

We first introduce dynamic page classification, which identifies two kinds of pages: eager-update pages and lazy-update pages. We next describe our HTTP extensions and explain how to use the extended HTTP to communicate between cache servers and the Web server. We then introduce class-based delta-encoding technique used by

EPDC. Finally, we describe an extended LRU algorithm used with EPDC and give a summary of this chapter.

3.1 Dynamic page classification

To classify eager-update and lazy-update pages, the proxy cache server counts two numbers for each page in the cache: the number of requests and the number of updates. When the first client request arrives, the proxy passes the request to the Web server and gets the object back. The proxy then sends the object to the client and saves a cache copy of the object. When the client's request for the same page comes again, the proxy passes the request with IMS (**IF_MODIFIED_SINCE**) header to the Web server. If the page has been updated then the Web server sends the new object back, otherwise it sends **"304 Not Modified"** message back. Figure 3.2 describes the HTTP IMS request for the not modified page and gives an equation for the total time of a client request getting the response from the Web server. All notations used in this chapter are defined in Table 3.1.

Parameter	Meaning
Connect_{pc_ws}	The elapsed time from where a proxy cache sends TCP_SYN to a Web server to the proxy receiving TCP_SYN_ACK from the Web server
Connect_{wc_pc}	The elapsed time from where a Web client sends TCP_SYN to a proxy cache server to the client receiving TCP_SYN_ACK from the cache server
DiskAccess_{pc}	The time it takes a proxy cache server to retrieve a cache object from disk to memory
Processing_{ws}	The time between a Web server receiving a dynamic page request and returning the first byte of the requested object
$\text{Processing}_{ws\_TimeOut}$	The time it takes a proxy server to abort an outgoing HTTP connection setup request for a Web server
Relay_{pc}	The time it takes a proxy cache server to relay a response to the request party
Reply_{pc}	The time it takes a proxy cache server to reply an object in memory to the request party
Reply_{pc_304}	The time it takes a proxy cache server to reply a "304:not modified" message
Reply_{ws}	The time it takes a Web server to reply an object in memory to the requesting party
RTT_{pc_ws}	The round trip time between a proxy cache server a Web server
RTT_{wc_pc}	The round trip time between a Web client and a proxy cache server
Search pc	The time it takes a proxy cache server to search for an object in its cache

Table 3. 1 Notation and Parameters

$Total\ response\ time\ for\ HTTP\ IMS\ request\ = Connect_{wc\_pc} + RTT_{wc\_pc/2}*2 + Connect_{pc\_ws}*2 + RTT_{pc\_ws/2}*2 + Processing_{ws} + Processing_{pc}$

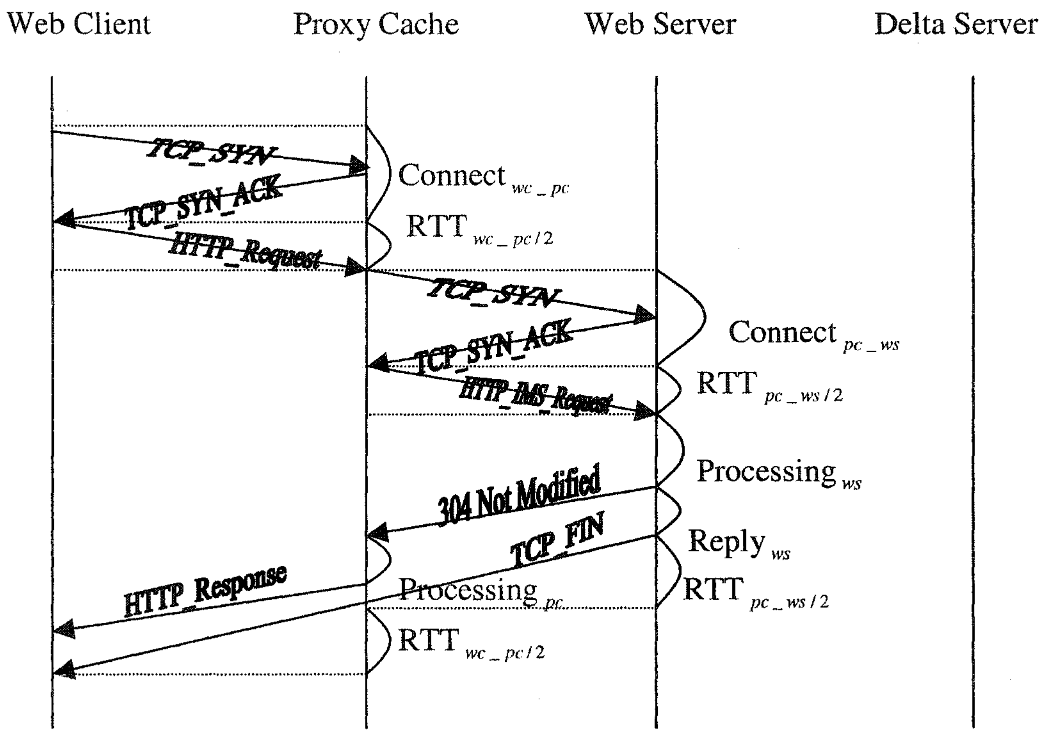


Figure3. 2 HTTP IMS request for the not modified page

The proxy increments the request count for a page for each client request and increments the update count for each response containing a new updated page. Here we use α as the ratio of $\frac{pageupdatecount}{pagerequestcount}$. If $\alpha < 1$, then the page update rate is slower than the page request rate, and the page is classified as an eager-update page; otherwise, the page is classified as a lazy-update page. This process is performed for each request so the sets of eager and lazy pages always change based on the above ratio. We choose threshold 1 here to evaluate the page request rate and page update rate. If the $\alpha = 1$, that means page update rate is equal to page request rate. The reason we choose 1 is that we hope to generate the eager-update page only once when update occurs at the Web server back-end system. The dynamic page generation is a time consuming process and we need to extensively decrease the Web server processing time. For example, if we choose 2 as the threshold which means the page update rate is faster than page request rate, even if no client request for that page, we still need to generate the page when update occurs. Thus wastes the Web server processing time. The proxy side algorithm to process a client request is shown in the Figure 3.3.

```
Process ReceiveMessageFromClient(msg: HTTPMessage)

//To check if the request URL is in local cache
   if (msg.requestURL.getFromCache())
      cspage = CachedPage;
      break;
   endif

   switch (msg.OPCode)

      case HTTP_GET_REQUEST:
           if (cspage.isEagerpage)
              add the page request number;
              LRUCache.findPage(msg.requestURL);
              Send that page's cache copy to the client;
          else
              add the page request number;
              send IMS request to the Web server;
          endif
   endswitch
end
```

Figure3. 3 Proxy side algorithm to process client requests

3.2 HTTP extension

To support the interaction between proxies and the Web server in EPDC, the HTTP Cache-Control header field can be extended through the use of one or more cache-extension tokens, each with an optional assigned value (the HTTP extension specification is given in Appendix B). Informational extensions (those that do not require a change in cache behavior) may be added without changing the semantics of other directives. Behavioral extensions are designed to work by acting as modifiers to the existing base of cache directives. Both the new directives and the standard directive are supplied such that applications that do not understand the new directive default to the behavior specified by the standard directive, and those that understand the new directive recognize it as modifying the requirements associated with the standard directive. In this way, extensions to the Cache-Control directives can be made without requiring changes to the basic protocol.

We can therefore use the cache-extension mechanism to transfer page class information. Our extension is as follows:

```
cache-extension = class
class =
    "eager"
  | "base"
  | "delta"
  | "from cache"
```

The Option Codes of the cache control header are as follows (new option codes are marked):

HTTP_GET_REQUEST: used by clients to send a GET request to the Web server.

HTTP_GET_IMM_REQUEST (new): used by proxies to send eager-update class information in the header of GET request to the Web server. It will add "class = eager" to the cache control header of the HTTP GET request.

HTTP_GET_IMS_REQUEST: used by proxies to send if-modified-since information in the GET request header to the Web server.

HTTP_FROM_CACHE_REQUEST (new): used by delta cache server to send a GET request to the Web server. It will add "class = from cache" to the cache control header of the HTTP GET request.

HTTP_200_RESPONSE: used by any server to respond a HTTP 200 OK to a GET request indicating that the request was successfully received.

HTTP_304_RESPONSE: used by any server to respond a HTTP 304 Not Modified to a GET request indicating that the document has not been modified.

HTTP_BASE_RESPONSE (new): used by the delta server. When the proxy sends the eager-update page information to the Web server, the delta server adds "class = base" in cache control header of a HTTP response to indicate that this is base file.

HTTP_PUT_BASE_REQUEST (new): used by the delta server to broadcast the base file to all registered proxies. It adds "class = base" in the cache control header of a HTTP PUT request.

HTTP_PUT_DELTA_REQUEST (new): used by the delta server to broadcast the delta file to all registered proxies. It adds "class = delta" in the cache control header of HTTP PUT request.

3.2.1 Eager-update page communication

When a client initiates a request for a page, if the proxy identifies it as an eager-update page, the proxy passes that **GET HTTP** request to the Web server. In the general header field it has:

Cache-Control: class = "eager"

which means that the proxy classifies the page as an "eager-update" page. For each eager-update page, the Web server keeps a list of proxies. When the Web server receives this information it knows the page's class and registers this proxy to the client list. It then asks the application server to construct the dynamic page. The application server sends the page to the delta server. The delta server classifies it as an eager–update page, saves it as a base file and broadcasts it to all proxies in the page list with the cache-control extension of " class = base". The proxy sends it to the client and saves a copy as a base file (Figure 3.4).

Total response time for HTTP request for eager page classification = $Connect_{wc\_pc}$ + $RTT_{wc\_pc/2}*2$ + $Connect_{pc\_ws}$ + $RTT_{pc\_ws/2}*2$ + $Processing_{ws}$ + $Processing_{ds}$ + $Reply_{ds}$ + $Processing_{pc}$

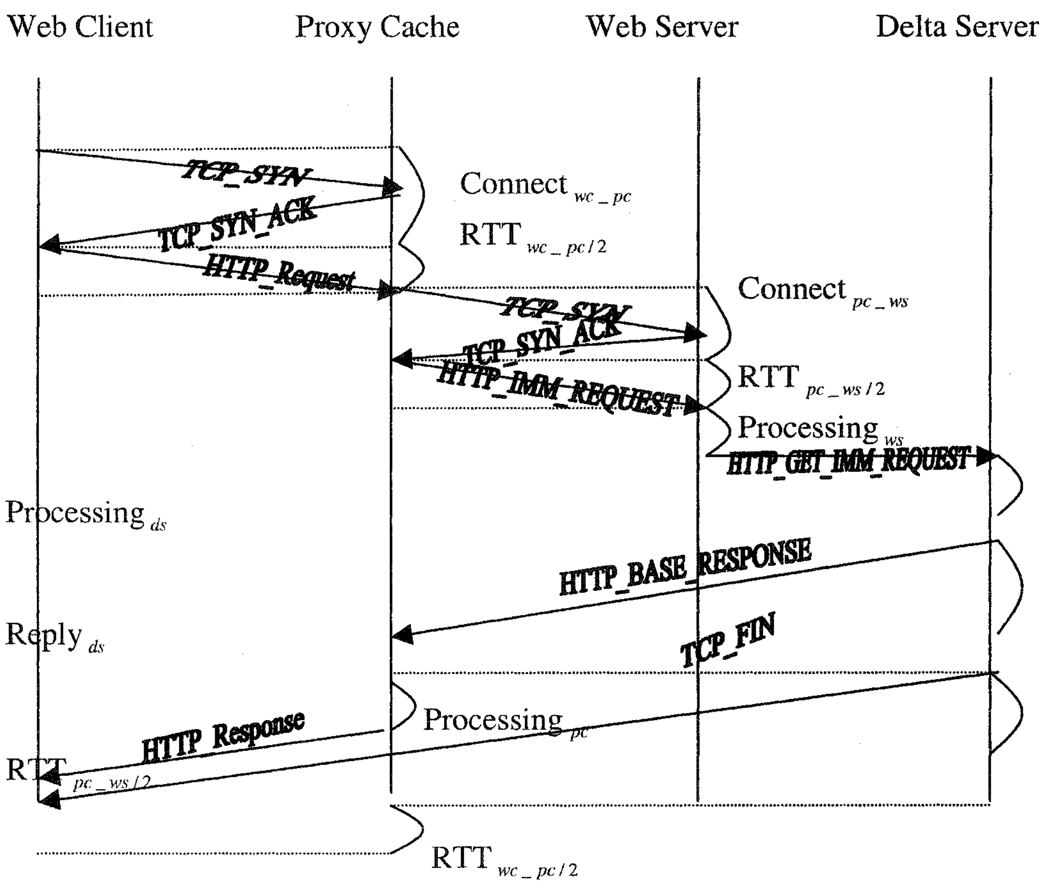


Figure3. 4 Eager-update page HTTP request from proxy to Web server for classification

Each time an eager page is updated, the Web server automatically notifies the delta server to calculate the delta between the new page and the base file. In our performance model, we have a delta server to send a HTTP GET request to the Web server if the page is stale. The delta server then adds Cache-Control: class = "from cache" to the general header and sends the delta using a **HTTP PUT** request to all proxies that are in the page's proxy list. It then saves the new update page as the base file.

When the proxy server receives **HTTP PUT** request with cache control extension of "class = delta", it recognizes the request URL matches it with a cached base file and combines the base file with the delta to construct a new page. It then saves the new page as the base file and sends a 200(OK) response to the Web server. When a new client request for that page arrives, the proxy directly returns the cached page to the client without validating the page (Figure 3.5).

Total response time for HTTP request for eager page $= Connect_{wc\_pc} + RTT_{wc\_pc/2} * 2 + Processing_{pc} + Reply_{pc}$

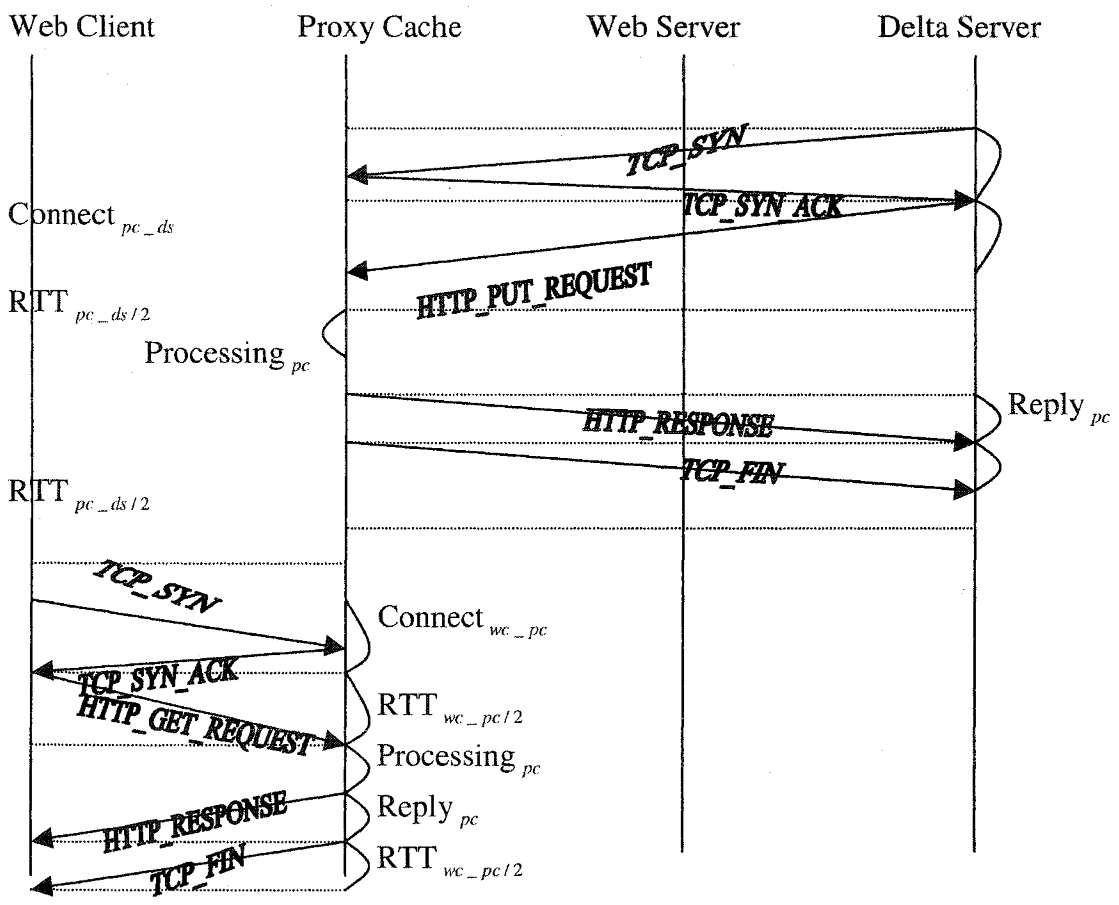


Figure3. 5 Eager-update page HTTP request from Web client to Web server

3.2.2 Lazy-update page communication

For Lazy-update pages, a proxy sends a GET HTTP request to the Web server. In the general header field with cache control extension of " if-modified-since", which means that the proxy classifies this page as a "lazy-update" page. When the Web server receives this information, it registers the proxy to the proxies list. If the page is not stale, the Web server sends "**304 Not Modified**" response back to the proxy, otherwise, the Web server sends the request to the application server to generate the page. The page is then sent to the delta server, which saves the page and sends it to the proxy cache server. The delta server uses class-based delta-encoding [26] for lazy-update pages.

When the delta server chooses the base file, it broadcasts the base file to all registered proxy caches using a **HTTP PUT** request with the cache control extension of " class = base". The proxy cache saves the base file and responds with 200 "OK". When the proxy cache server receives the next request from the client, if there is a cached copy, the proxy will still send a HTTP GET request with IMS to the Web server. If the Web server finds the page is still valid, it responds with 304 "Not Modified" to the proxy cache. If an update has occurred, the application server generates the page and sends it to the delta server. The delta server recognizes the page's group and calculates the delta between the base file and the page. It then sends the delta to the proxy cache server. The proxy combines the base file and delta to generate the page, which it sends to the client (Figure 3.6).

Total response time for HTTP request for lazy page = $Connect_{wc\_pc} + RTT_{wc\_pc/2} * 2 + Processing_{pc} + Connect_{pc\_ws} + RTT_{pc\_ws/2} * 2 + Processing_{ws} + Processing_{ds} + Reply_{ds} + Processing_{pc} + Relay_{pc}$

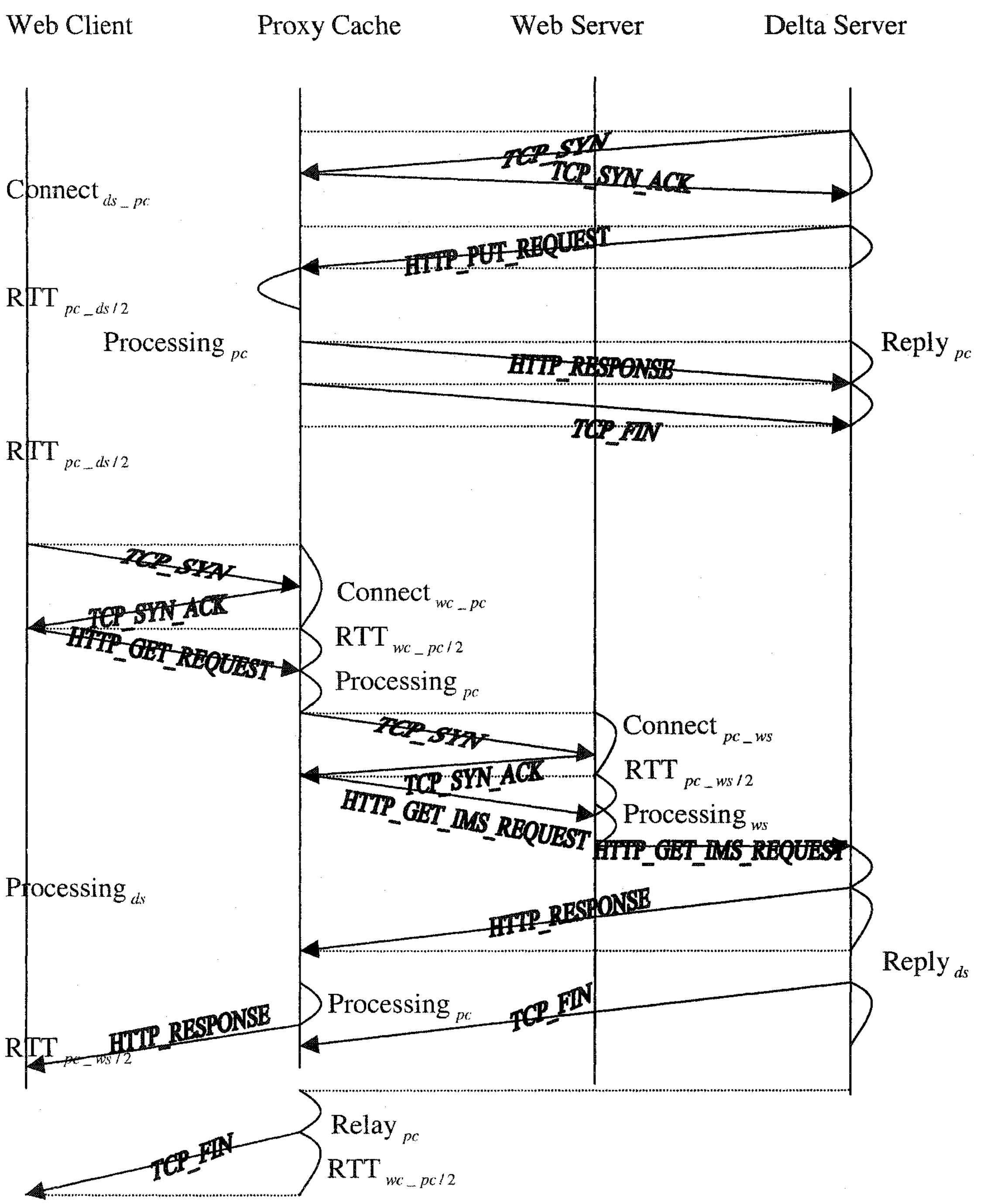


Figure3. 6 lazy-update page HTTP request from Web client to Web server

3.3 Class-based delta-encoding technique

Delta-encoding is the process of generating a difference file, called a delta, between two files with the following two properties: (1) the combination of the delta and one of the files, called the base-file, suffices to reproduce the other file, and (2) the size of the delta is as small as possible. A detailed discussion of delta-encoding is given in Appendix A.

In the context of HTTP, delta-encoding can be used to exploit temporal correlations between consecutive snapshots of a dynamic Web-document. As shown in Figure 3.7, the client and the server share a common base-file, which is a snapshot of the dynamic document at some point in time. Whenever the client requests the document from the server, the server computes the delta between the current snapshot of the document and the base-file, and sends the delta to the client. Upon receipt of the delta, the client computes the current snapshot of the document by combining the delta and the locally stored base-file.

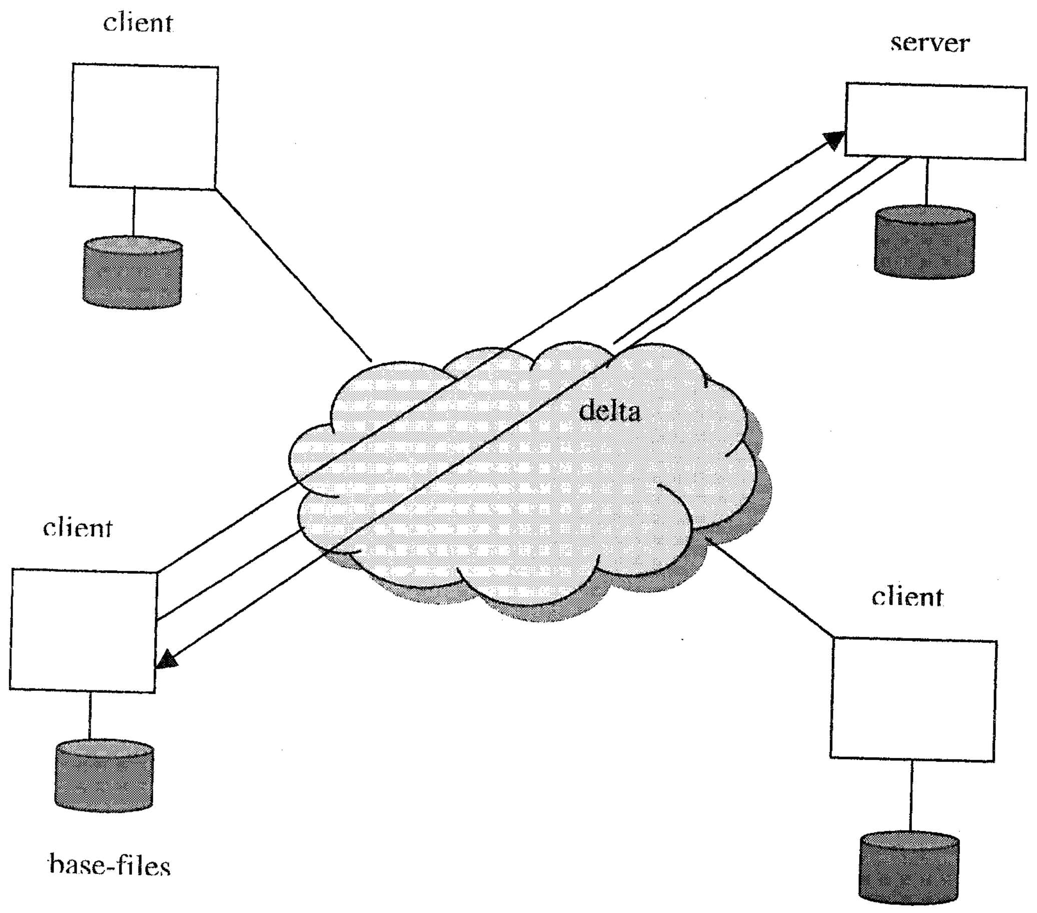


Figure3. 7 Using delta-encoding for Web-documents

In the following two sections we introduce the class-based delta-encoding mechanism [26] used in EPDC. EPDC adopts class-based delta-encoding mechanism to extensively decrease network traffic and reduce response time.

3.3.1 Grouping Documents into Classes

The mechanism aims to quickly identify a good class for each lazy-update page. A class is good if the size, in bytes, of the delta between the base-file of the class and the document is small.

All requests are processed by the delta-server after they are forwarded by the Web-server. Initially, there are no classes formed in the delta-server. Whenever an ungrouped URL-request arrives at the delta-server, the scheme groups it into an existing class, or creates a new class.

We partition the URLs in three segments (parts), the server-part, the hint-part, and the rest. The server-part is the string from the beginning of the URL till the first slash, as usual. The portion of the URL that is used as the hint-part differs among Web servers and depends on how the Web server organizes its content. For example, let **www.computersell.com** be a Web site that sells computers, laptops and desktops. Assume that documents corresponding to laptops are similar, while they differ from documents corresponding to desktops. Table 3.2 shows the URL-partitioning for three different cases. Depending on the Web site, the administrator describes to the grouping mechanism how to partition URLs into parts using regular expressions. Then, the mechanism uses these parts to expedite the grouping process.

URL	Hint-part	rest
www.computersell.com/laptops?id=100	laptops	id=100
www.computersell.com/?dept=laptops&id=100	dept=laptops	id=100
www.computersell.com/laptops/100	laptops	100

Table 3. 2 URL-parts for differently organized Web-sites

We now describe the grouping mechanism. A match occurs if the delta between the requested document and the base-file of the class is smaller than a threshold. A new class is created in case there are no classes with members whose server-part is the same with the request's server-part. Else, some heuristics are used to tradeoff between search-time and matching-quality:

- If some classes have members whose hint-parts are the same with the request's hint-part, the mechanism only considers those as potential classes to group the request.
- The mechanism never considers more than N existing classes as potential classes

  to group a request. If no match is found after N tries, a new class is created.
- The mechanism first attempts to group the request into classes with many members, and then into less popular classes. In particular, the first *a*N* tries consist of the most popular classes, and the last *(1-a)*N* consist of random selections among the rest of the eligible classes.

3.3.2 Choosing a Base-file

Once a class is formed, it is necessary to identify a good base-file. The following scheme is proposed:

a) Sample each request with probability *p,* i.e., consider the corresponding document as a base-file candidate and store it in memory.

b) Use as a base-file the best of the stored documents, i.e. the document that minimizes the sum of deltas between itself and all other stored documents.

c) Store up to *K* documents. Thus, after acquiring *K* samples, whenever a new sample is drawn, evict the document that maximizes the sum of deltas.

By design, the algorithm stores good base-file candidates and uses the best out of those as a base-file. Let a *rebase* be the process of changing a base-file. After a rebase, the new base-file should be distributed to all proxies before they can benefit from delta-encoding. To control the number of rebases, a rebase takes place if both a better base-file candidate exists, and a rebase-timeout, since the previous rebase, has expired.

3.4 Extended LRU Algorithm

The Least Recently Used (LRU) algorithm [29] is the most popular replacement algorithm in Web Caching systems. It sorts cached documents by the latest access time. When a cache hit occurs the access time of the requested document is updated and it is moved to the head of the list. The least recently used document (located at the tail of the list) is the next to be replaced.

Since LRU drops the page that has not been accessed for the longest time when a new cache space is needed, it limits itself to only considering the time of last reference. Specifically, LRU does not discriminate well between frequently and infrequently referenced pages until the system has possibly wasted a lot of resources keeping infrequently referenced pages in the cache for an extended period. Many extended LRU algorithms, such as LRU-K **[41]**, have been proposed to improve the LRU algorithm.

EDPC uses LRU but always gives priority to eager-update pages. When a request for an eager-update page is received, the cache server sends a request for that page to the Web server and gets the newest object back. If the cache server has no space left, the victim chosen is the least recently used lazy-update page in the cache server. When a request for a lazy-update page comes, it can only replace the least recently used lazy-update page in the cache server too. On the other hand, if the proxy cache server classifies an eager-update page turning into a lazy-update page when receiving the request for that page, that eager page is marked as lazy-update page then added to the header of the lazy-update pages queue in the cache. Finally it is replaced by the newest requested lazy-update page from the cache if it is not requested for a sufficiently long time. The detailed algorithm is described in the Figure 3.8.

```
Process LRUaddToCache(URL: le)
private long MaxCacheSize;
private long CacheUsed;
private CacheDB eager_cacheDB;
private CacheDB lazy_cacheDB;

//if it is in cacheDB, can't be added
if (eager_cacheDB!=null or lazy_cacheDB!=null)
        return false;
endif
//check cache size
if (le.size>MaxCacheSize)
        return false;
endif

CacheEntry ceLazyTailer = lazy_cacheDB.getTailerEntry();

 While ((le.size + CacheUsed) > MaxCacheSize)

      if(ceLazyTailer == null)
           return false;
      discard(ceLazyTailer);
 endwhile

//add the new entry to the header of lazy_cacheDB
      CacheEntry ceNew = new CacheEntry();
         ceNew.setPrev(0);

       if(lazy_cacheDB.getHeaderEntry()!=null)
                CacheEntry ceHeader = lazy_cacheDB.getHeaderEntry();
                ceNew.setNext(ceHeader);
                ceHeader.setPrev(ceNew);
                lazy_cacheDB.updateEntry(ceHeader);
       endif

lazy_cacheDB.addEntry(ceNew);
CacheUsed += ceNew.size;
return true;
end
```

Figure3. 8 Add page function of the Extended LRU algorithm

3.5 Summary

In this chapter, we introduced the Eager-update Page Dynamic Caching scheme. EPDC dynamically identifies two kinds of pages: eager-update pages and lazy-update pages. It keeps strong consistency of eager-pages between cached copies and the underlying data objects. Using an instant pre-computing strategy, EPDC only computes eager-update pages once per update in order to save the Web server cost. EPDC also computes lazy-update pages only when requests come, so it also reduces computing resource contention.

For eager-update pages, the Web server always "pushes" the newest data to the proxy after updates to the dynamic page content. EPDC also extends HTTP to support communication between cache servers and Web servers. It not only reduces the critical response time but also improves the hit ratio by extending the LRU algorithm to give priority to eager-update pages in the cache. EPDC also uses class-based delta encoding to reduce network bandwidth and latency.

Chapter 4

Performance Evaluation

In this chapter, we evaluate the performance of our proposed EPDC scheme. The results are compared with those of Delta Encoding and LRU. Section 4.1 explains the simulation model produced in this research, which includes a network model and a workload model. The effects of network link delay and HTTP request arrival rate on the performance of the Web dynamic caching schemes are reported in Section 4.2. Section 4.3 analyses the Web server cost and Section 4.4 provides a summary of the results obtained in the simulation study.

4.1 Simulation Model

We first describe the simulation model, including the network model and the Zipf-like distribution used to generate HTTP request traffic. The simulation of the EPDC schemes is then described, followed by the necessary parameter settings and the simulation software structure.

4.1.1 Network Model

To evaluate the performance of the EPDC scheme, we developed a simulator that models the behavior of a proxy cache server and allows us to observe and measure the performance of EPDC, Delta-encoding and LRU schemes under a variety of conditions. We compare EPDC with Delta-encoding because Delta-encoding is a popular dynamic caching technique used in Akamai [24] and SpiderCache [31]. We compare EPDC with LRU because it is used mostly in general Web caching environments.

In this study, a dynamic caching architecture is simulated. The network model for the three simulated dynamic caching schemes is shown in Figure 4.1.

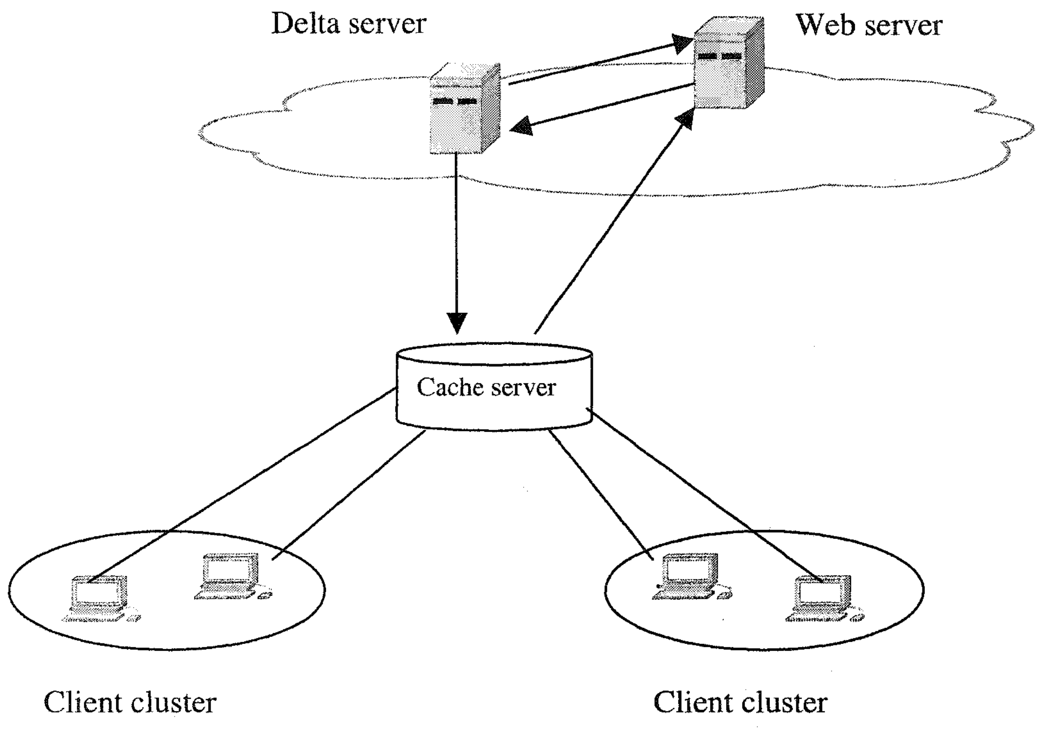


Figure4. 1 Network Model

In EPDC, the cache server accepts HTTP requests from a cluster of clients and sends HTTP requests to the Web server. The Web server generates new pages and the Delta server generates deltas. In Delta-Encoding, the only difference from EPDC scheme is that every page request needs to be sent to the Web server and then a delta is returned back if any update has occurred for that page. In LRU, there is no delta server on the

Web server side. Each page request needs to be sent to the Web server for the newest page.

4.1.2 Workload generation

We cannot use publicly available proxy traces as input to our simulation due to privacy concerns, which force the deletion of data necessary for classification for these traces. For example, most dynamic pages contain private information such as credit card numbers or passwords. Therefore, a Zipf-like distribution [38][5] has been used to accurately model Web access patterns and is used here. There have been many studies on page request distribution, that is, the relative frequency with which Web pages are requested [29], which indicate that a Zipf-like distribution is a good approximation for Web accesses. Zipf's law predicts that the relative frequency of access for an object is a function of its popularity, that is, the *i*-th most popular object will be accessed with a frequency proportional to $1/i^{\alpha}$ [38], where the exponent varies from trace to trace and the concentration of Web accesses to "hot" documents depends on α.

Previous work has shown that a simple model of independent requests with a Zipf-like distribution of object popularity can be used to accurately model access skews observed at Web proxies [38]. Figure 4.2 depicts the cumulative access probability of objects as a function of the fraction of objects accessed for a Zipf-like distribution with different values of θ, where $\theta = 1-\alpha$. It can be seen that, for the access skew with $\theta = 0$, about 70% of the accesses are restricted to 20% of the objects, which is a 70-20 skew. As θ decreases, popular documents receive a greater fraction of requests, so a Zipf-like distribution yields high access skews at low values of θ, and tends to become more uniform at larger values of θ.

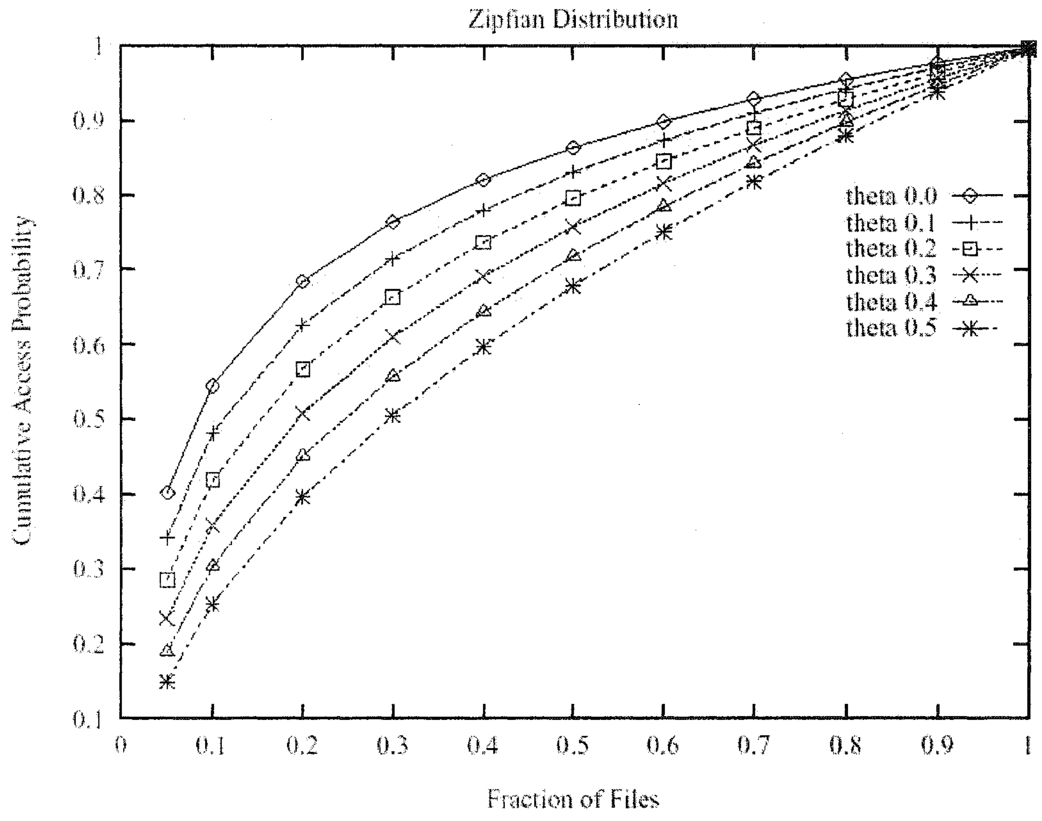


Figure4. 2 Zipf-like Distribution[39]

We create a workload of page references based on a Zipf-like distribution of reference probabilities. For each distinct object in the set of objects used by clients, we can compute the access probability for the object given its popularity rank, the total number of objects in the set, and the Zipf parameter α. We assume that the least popular gets one access. Since the number of accesses to the *i*-th most popular object *n(i) = P(i) * n,* where *P(i)* is the access probability of the object and *n* is the total number of accesses to all objects, we can further deduce *n* from the access probability of the least popular object, and the number of accesses to each of the other objects can be easily calculated.

Given the list of all the distinct pages ranked in terms of their popularity order and the number of accesses to each page, we can proceed to generate the input workload to our simulation model. The input is implemented as a text file, where each entry in the file stores all the attributes that identify a particular request. We identify four attributes for each entry:

Page ID: a unique number representing the URL of the requested object.

Client ID: a number representing the client IP address.

Timestamp: the time at which the client socket is closed. The format is "Unix time"(seconds since January 1, 1970) with millisecond resolution.

Size: the number of bytes transferred from the proxy to the client.

In our experiments, we use a Poisson distribution to model the request arrivals input workload. We assume there are 1000 dynamic pages available at the Web server. There are 10 news groups and each group has 100 pages. We also assume that dynamic pages are invalidated at 30-second intervals. We assume four different request rates (30,75,150,200 per minute) and generate four 24 hours Web traffic log files as input workload for our experiments. The ratio of eager-update to lazy-update pages is a parameter that is varied in the simulations.

4.1.3 Simulation Parameter Settings

The parameters used in the simulation are based on data measured by Chiang [11] and Rousskov [28]. These parameters are summarized in Table 4.1.

Parameter	Meaning	Nominal Value	Source
Connect $_{pc\_ws}$	The elapsed time from where a proxy cache sends TCP_SYN to a Web server to the proxy receiving TCP_SYN_ACK from the Web server	350ms	[28,11]
Connect $_{wc\_pc}$	The elapsed time from where a Web client sends TCP_SYN to a proxy cache server to the client receiving TCP_SYN_ACK from the cache server	0~30ms	[28,11]
DiskAccess $_{pc}$	The time it takes a proxy cache server to retrieve a cache object from disk to memory	100ms	[28,11]
Processing $_{ws}$	The time between a Web server receiving a dynamic page request and returning the first byte of the requested object	500ms	**[40]**
Processing $_{ws\_TimeOut}$	The time it takes a proxy server to abort an outgoing HTTP connection setup request for a Web server	15000ms	[11]
Processing $_{ws\_GIMS}$	The time between a Web server receiving a GIMS request and returning the first byte of the requested object	250ms	[11]
Relay $_{pc}$	The time it takes a proxy cache server to relay a response to the request party	50ms	[11]
Reply $_{pc}$	The time it takes a proxy cache server to reply an object in memory to the request party	150ms	[11]
Reply $_{pc\_304}$	The time it takes a proxy cache server to reply a "304:not modified" message	50ms	[28,11]
Reply $_{ws}$	The time it takes a Web server to reply an object in memory to the requesting party	150ms	[11]
RTT $_{pc\_ws}$	The round trip time between a proxy cache server a Web server	300ms	[11]

RTT $_{wc\_pc}$	The round trip time between a Web client and a proxy cache server	0~20ms	[11]
Search *pc*	The time it takes a proxy cache server to search for an object in its cache	250ms	[11]

Table 4. 1 Simulation parameters

4.2 Simulation Results

The details of the simulation software are presented in Appendix C. In this section, we describe and analyze our simulation experiments. In the experiments, network link delay (including propagation delay, packet transmission delay and network access delay) and the page arrival rate at the proxy are two main variable parameters used to evaluate the dynamic caching schemes. We also examine how the number of hits on eager pages per unit time (**HIT**) and changing workload influence the client response time.

4.2.1 HTTP request rate and response time

The number of HTTP requests received by a proxy determines the workload of the proxy. The response time of a proxy cache server should follow the HTTP request rate. So with a Poisson distribution to generate request inter-arrival time we expect that schemes should have flat response time curves.

The average response times under different link delays and page arrival rates are shown in Figure 4.3–4.5. EPDC outperforms the other two schemes in all cases. Under a specific link delay (90 milliseconds), as shown in Figure 4.3, when the arrival rate is low (30/min), EPDC outperforms Delta-encoding by 4.5% on average, and LRU by 11.7% on average. When the arrival rate increases to 75, 150, 200 and 500 messages per minute, EPDC outperforms the Delta-encoding by 7.3%, 10.2%, 11.9% and 18.5%, respectively. It also outperforms LRU by 14.3%, 16.6%, 17.4% and 22%, respectively. There are more requests for the eager pages when the arrival rate increases so EPDC saves the connection time between proxy cache servers and Web servers.

The interesting thing we can find from Figure 4.3 is that under the same link delay, when arrival rate is big enough such as 200 messages per minute we can see the response time goes up. The reason is that once the number of arrival messages goes up, they are congesting in the waiting queue at the proxy cache server. We can see when arrival rate = 150 per minute we can achieve the best response time. We also can see EPDC outperforms Delta-encoding and LRU schemes more when arrival rate is bigger than 150 messages per minute. The response time in EPDC does not increase as much as that of Delta-encoding and LRU when the arrival rate increases (arrival rate > 150 messages). The reason is that when the arrival rate is higher, EPDC save more connection time between proxy cache servers and Web servers.

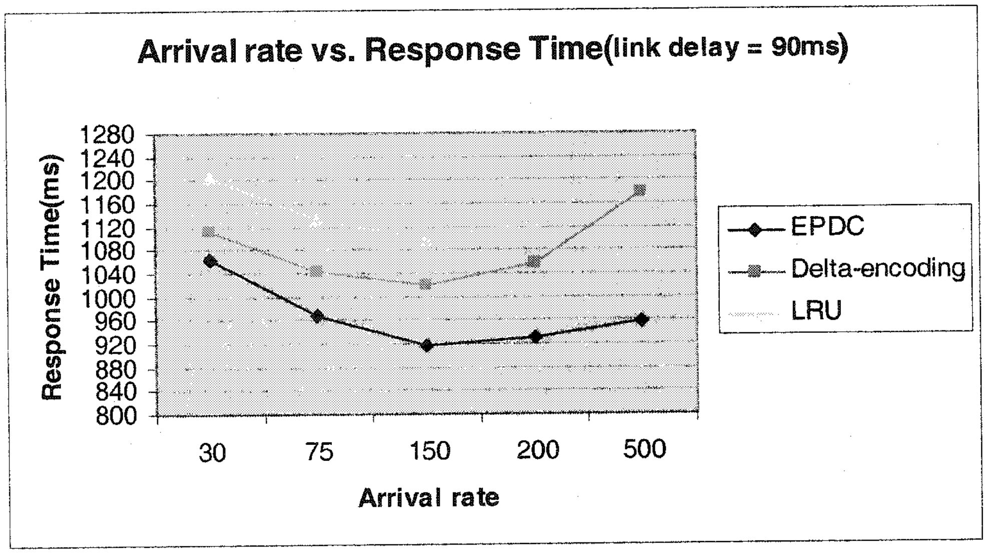


Figure4. 3 Average response time at link delay = 90 ms

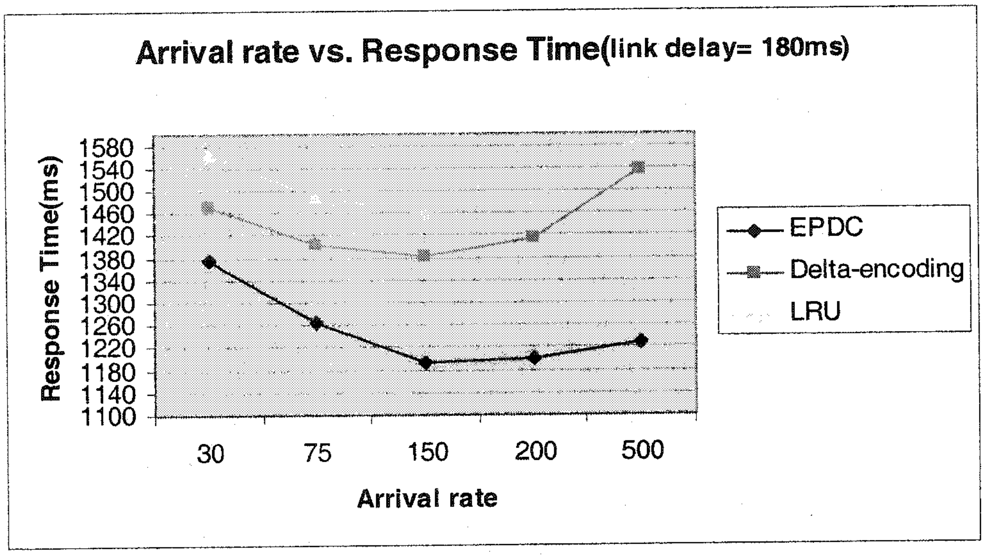


Figure4. 4 Average response time at link delay = 180 ms

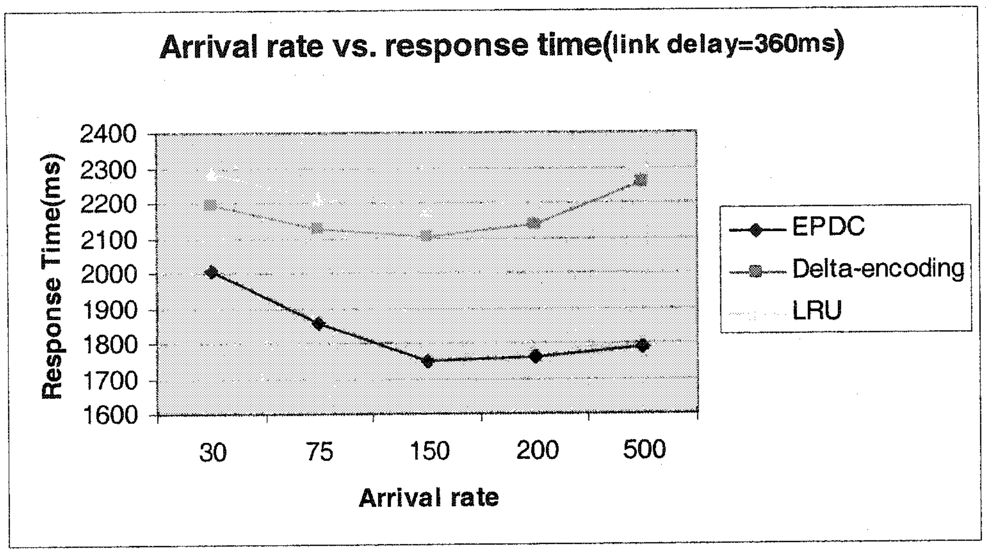


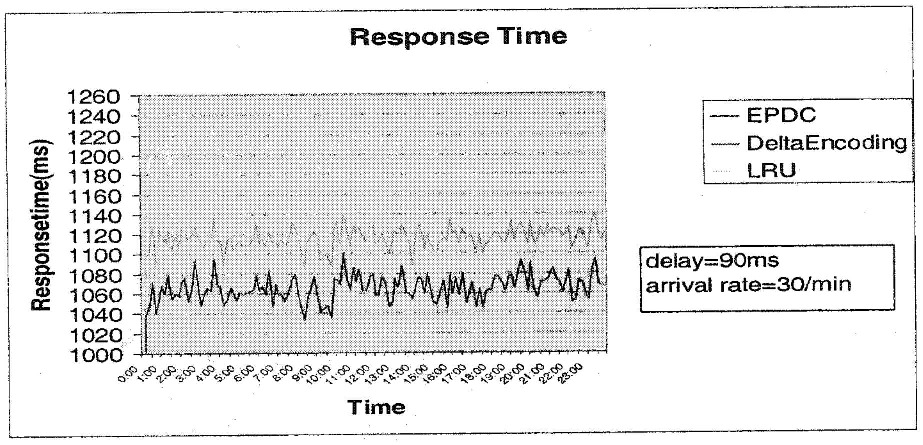
Figure4. 5 Average response time at link delay = 360 ms

4.2.2 Network link delay and response time

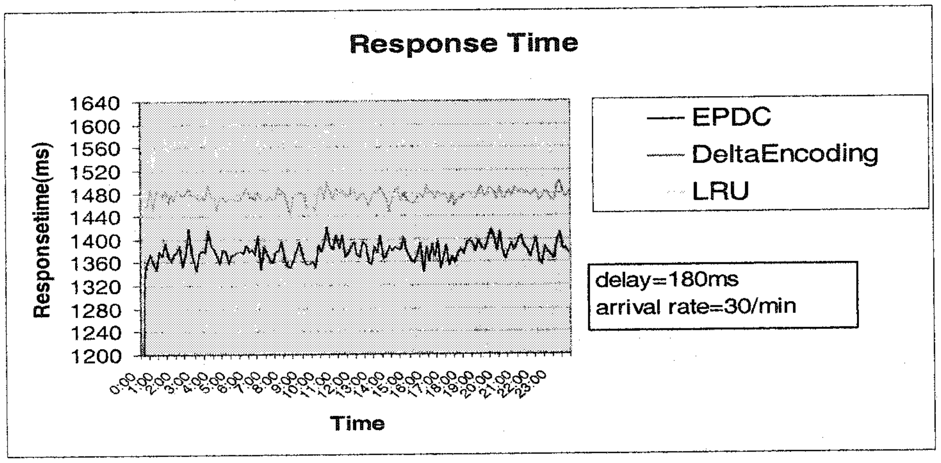
Figure 4.6 shows that under different network link delays, EPDC effectively outperforms the Delta-encoding and LRU schemes. Under large link delays and the same arrival rate, EPDC's performance is even better. As shown in Figure 4.6, EPDC outperforms Delta-encoding by 4.5%, 6.6% and 8.7% when link delays are 90ms, 180ms and 360ms, respectively. Similarly, EPDC outperforms LRU by 11.7%, 12% and 12.3% on average. Table 4.2 shows that when arrival rates are 30, 75, 150, 200 and 500 messages per minute, the larger the network link delay, the more EPDC outperforms Delta-encoding and LRU schemes. This is because EPDC decreases the communications between the proxy cache server and the Web server.

Scheme	Arrival Rate / Link Delay	30/min	75/min	150/min	200/min	500/min
EPDC	90ms	*1063ms*	*969ms*	*917ms*	*930ms*	*958ms*
	180ms	*1376ms*	*1266ms*	*1193ms*	*1201ms*	*1229ms*
	360ms	*2003ms*	*1860ms*	*1746ms*	*1762ms*	*1789ms*
Delta-encoding	90ms	*1113ms*	*1045ms*	*1021ms*	*1056ms*	*1176ms*
	180ms	*1473ms*	*1405ms*	*1381ms*	*1416ms*	*1536ms*
	360ms	*2193ms*	*2125ms*	*2101ms*	*2136ms*	*2256ms*
LRU	90ms	*1204ms*	*1131ms*	*1099ms*	*1126ms*	*1228ms*
	180ms	*1564ms*	*1491ms*	*1459ms*	*1486ms*	*1588ms*
	360ms	*2284ms*	*2211ms*	*2179ms*	*2205ms*	*2308ms*

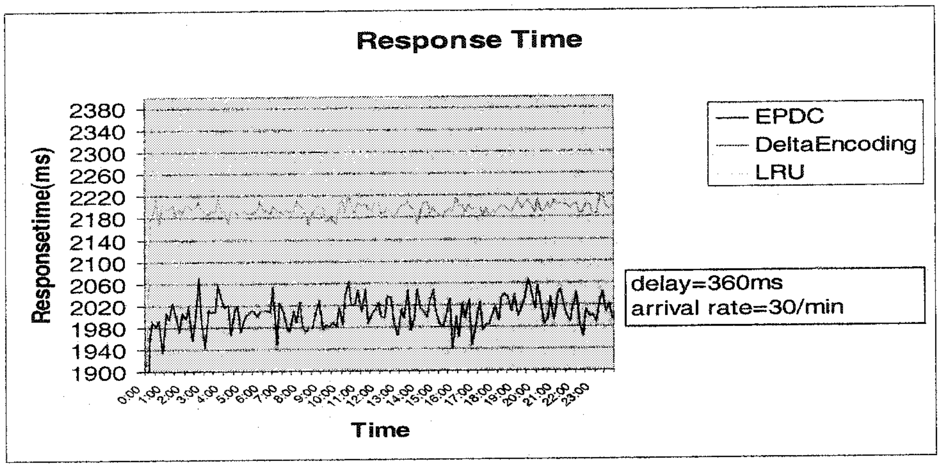
Table 4. 2: Average response time for different link delays



(a)Average response time at link delay = 90ms



(b) Average response time at link delay = 180ms



(c) Average response time at link delay = 360ms

Figure4. 6 Page arrival rate = 30 messages/min at proxy

4.2.3 Changing workload and Response Time

To test the EPDC scheme's adaptability, we alter the pattern of page accesses by changing the page rankings. We partition the 24-hour access trace into 6 parts. We conduct successive four-hour experiments continuously based on different Zipf-like distributions to examine how EPDC reacts to the workload changes. In Figure 4.7, we examine the 40 minutes surrounding the changing workload point time = 4am (3:40 to 4:20). From the X-axis of Figure 4.6, we find from *x = 21 (4:01)*, when the access changes, the response time changes gradually from *920ms* up to *1160ms*. This indicates that the cached eager pages are no longer useful. The response time begins to decrease at *x = 36 (4:16)*, which means that EPDC has loaded the cache with the new popular pages that are now marked as eager.

Figure 4.8 shows response time difference between EPDC and Delta-encoding schemes for the whole 24-hour period. We can see at each access change point, the response time difference goes down a little then goes up rapidly. This reflects the quick recovery of the EPDC response time regaining the response time difference between EPDC and Delta-encoding. The main reason is that when page accessibility changes, EPDC identifies more eager-update pages in very short time period. Then after dynamic pages' updates occur a lot lately, more eager-update pages turn into lazy-update pages so that the response time goes up again. Such fluctuations only last a few minutes and are hence indicative of the adaptability of EPDC to changes in page request patterns. Figure 4.9 shows the workload pattern changing point and response time difference between EPDC and Delta-encoding for different arrival rates.

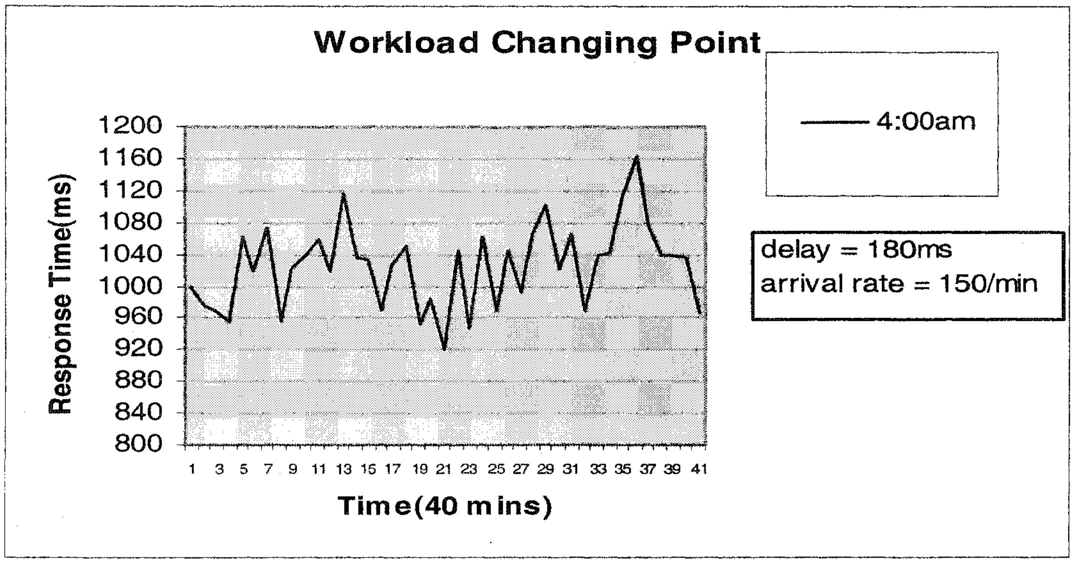


Figure4. 7 Workload changing point

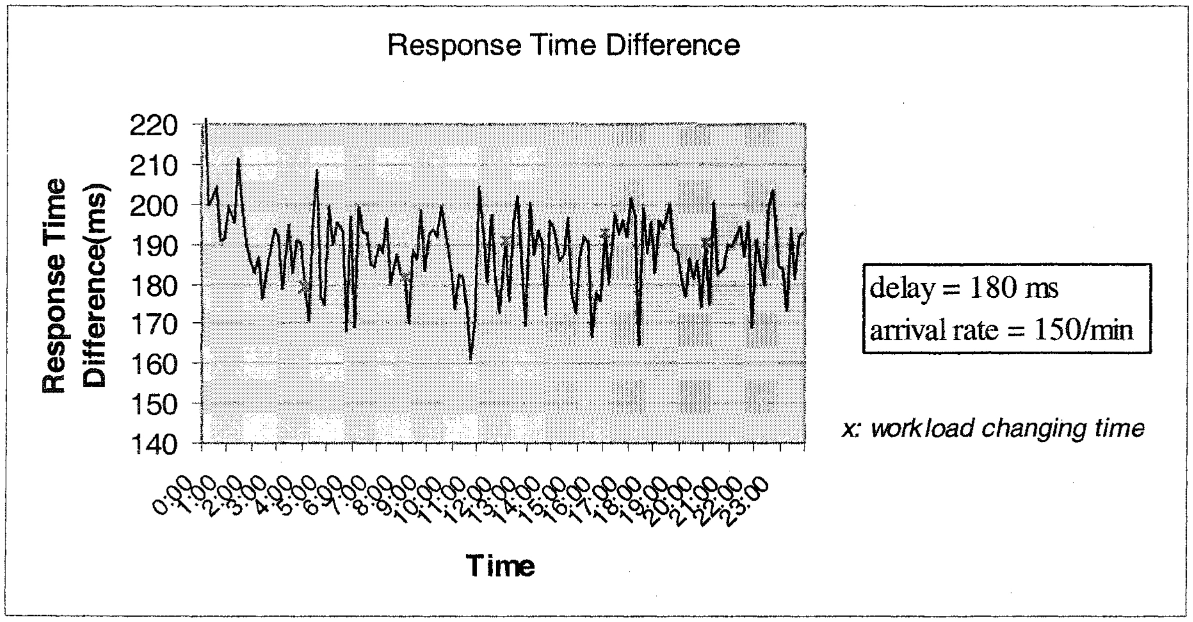
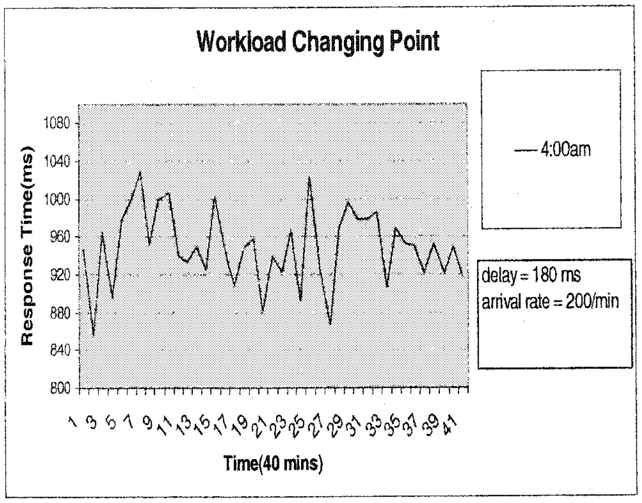
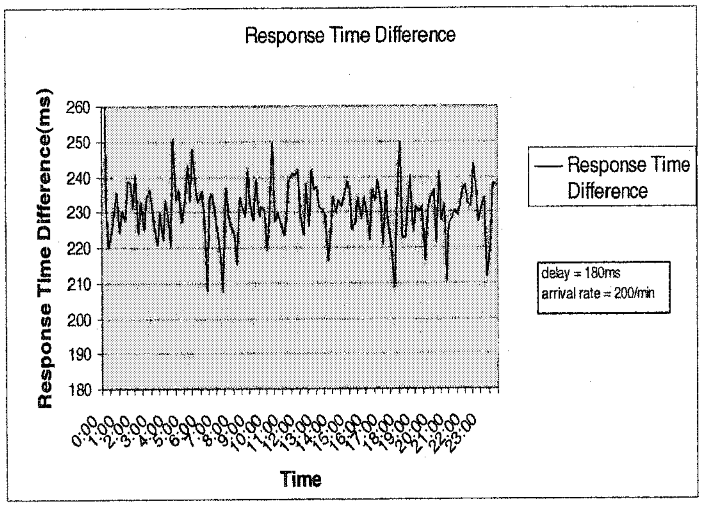


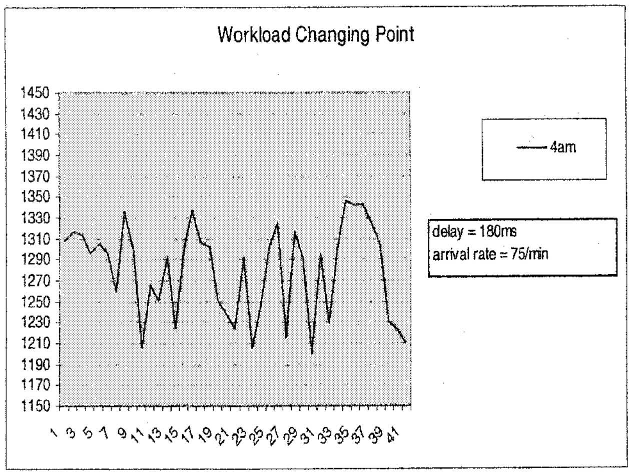
Figure4. 8 Response Time Difference



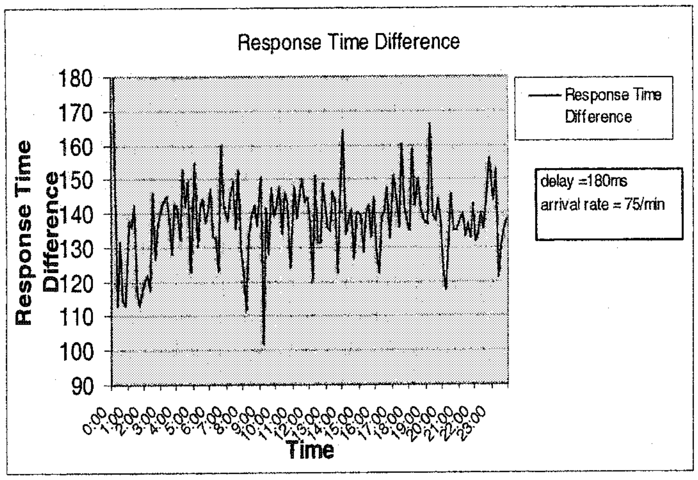
(a.1)



(a. 2)



(b.1)

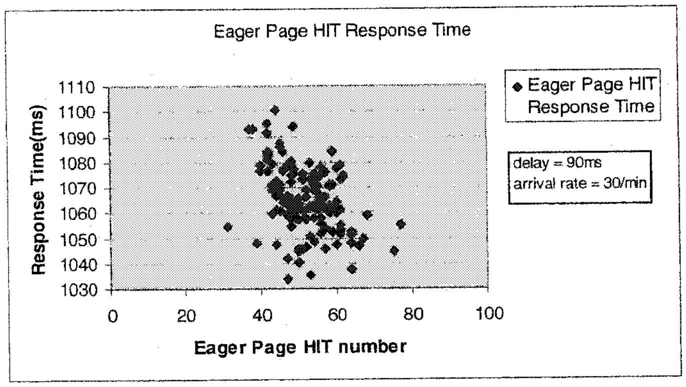


(b. 2)

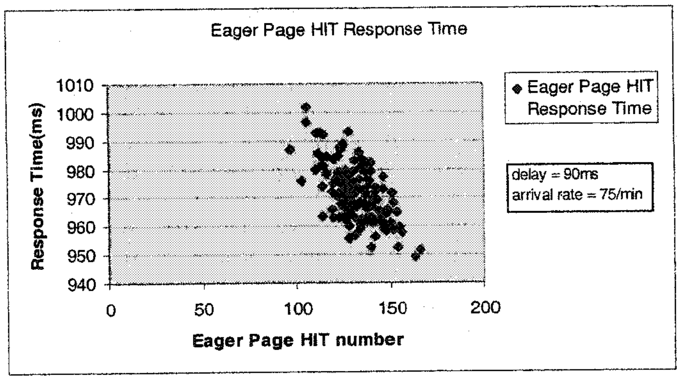
Figure4. 9 Workload Changing Point and Response Time Difference

4.2.4 Eager Page HIT and Response Time

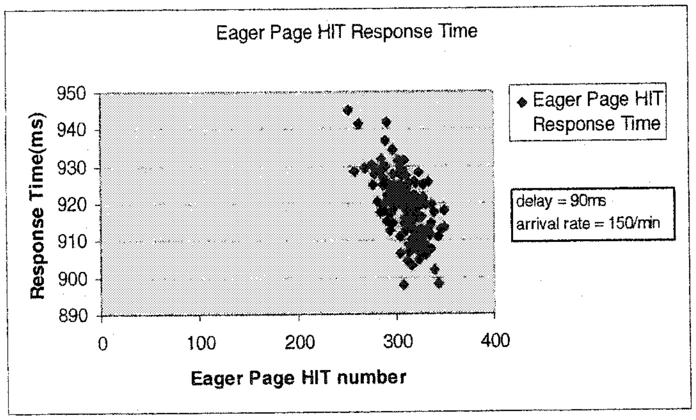
We collect the average number of cache hits during a 10-minute period for each eager-update page (**HIT**). We also collect the response time for each period. We plot the average response time versus HIT in Figure 4.10 and see that when **HIT** increases the response time drops accordingly. We also can see that under the same link delay, the higher the page arrival rate is, the lower the response time. Figure 4.11 shows the relation between **HIT** and the response time under different link delays and page arrival rates. The eager-update pages have the cache priority in the proxy cache server. When the client request for an eager-update page arrives, it always gets the newest cache copy from the proxy cache server without going to the Web server. So it reduces the response time significantly. The greater the number of requests for eager-update pages, the greater the savings in client response times.



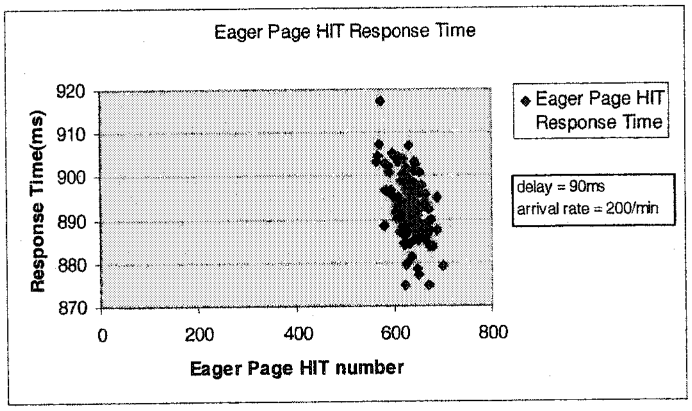
(a) page request rate = 30/min



(b) page request rate = 75/min

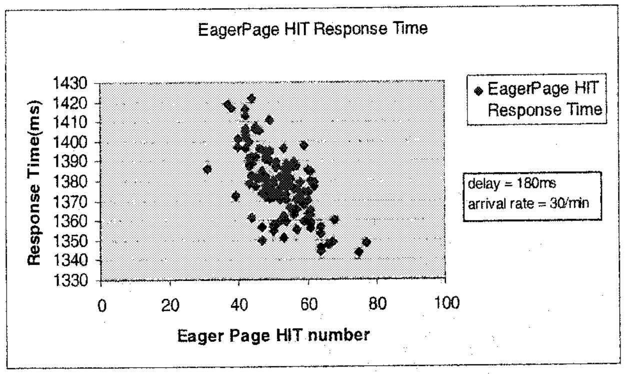


(c) page request rate = 150/min

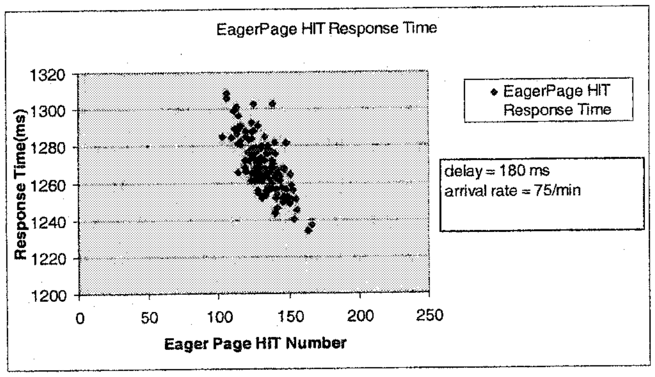


(d) page request rate = 200/min

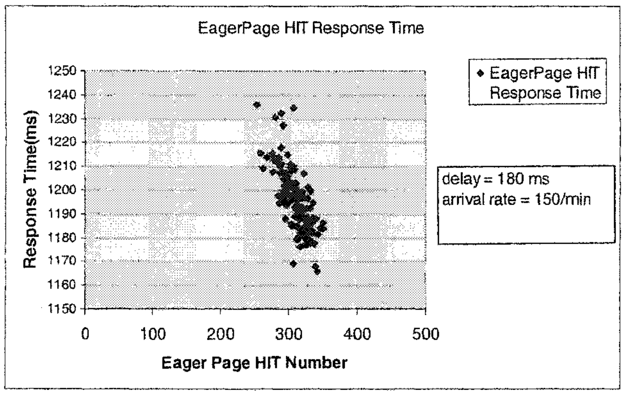
Figure4. 10 Eager Page HIT Response Time



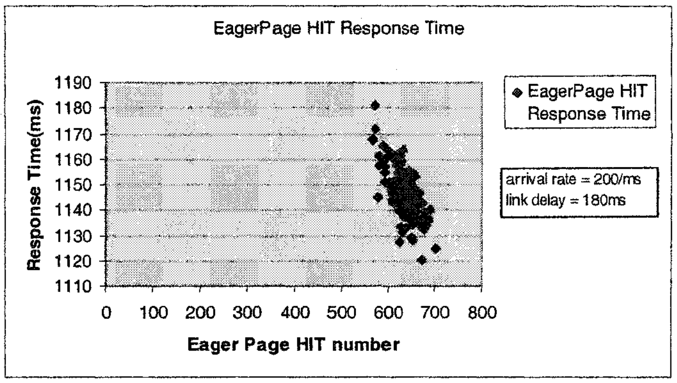
(a.1)



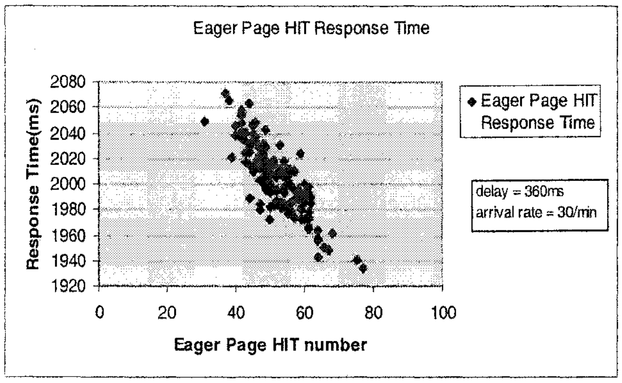
(a.2)



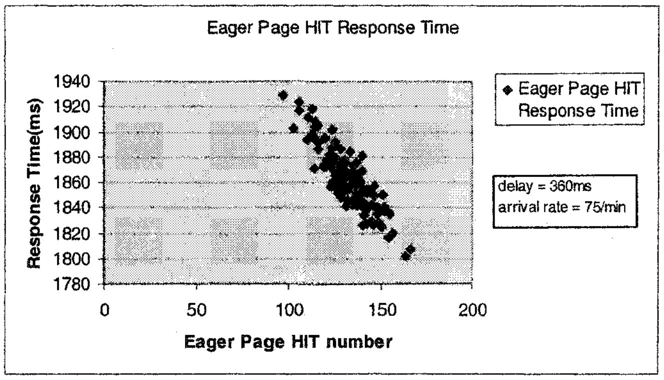
(a.3)



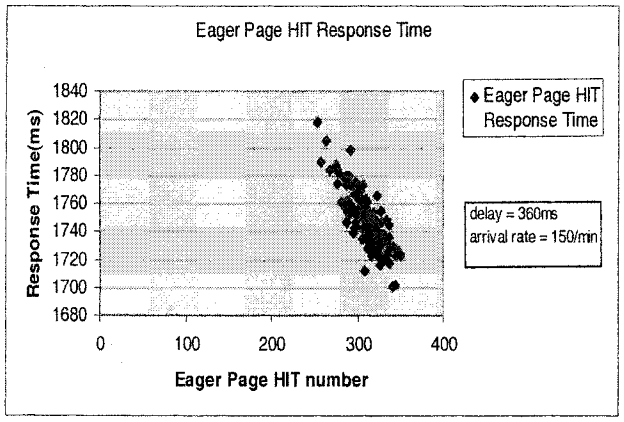
(a.4)



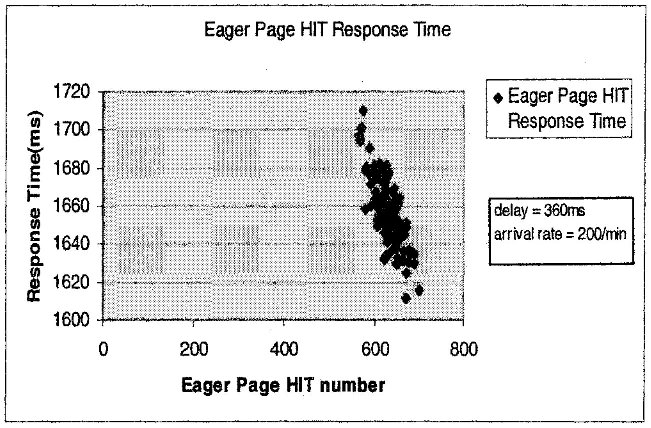
(b.1)



(b.2)



(b.3)



(b.4)

Figure4. 11 Eager Page HIT Response Time

4.2.5 Hit Ratio

The cache hit rates of the two schemes under different page request rates are shown in Figure 4.12. The LRU scheme adopts the same replacement algorithm as the Delta-encoding scheme so they have the same hit ratio. Comparing the EPDC to the Delta-encoding scheme, we find EPDC achieves a higher hit rate. The average hit rates for EPDC are 2% to 3% higher than Delta-encoding. The reason is that we extend the LRU replacement algorithm for EPDC. In EPDC, eager-update pages have priority to lazy-update pages in the proxy cache server. Every time client requests for eager-update pages always get hit in the cache server. The Delta-encoding scheme adopts the normal LRU replacement algorithm that indiscriminately treats eager-update pages and lazy-update pages. The client requests for eager-update pages sometimes could not get the hit in the cache server because those cached copies are already deleted from the cache server. Comparing with the hit rate of non-dynamic content caching (40%), the hit rate of EPDC is very significant.

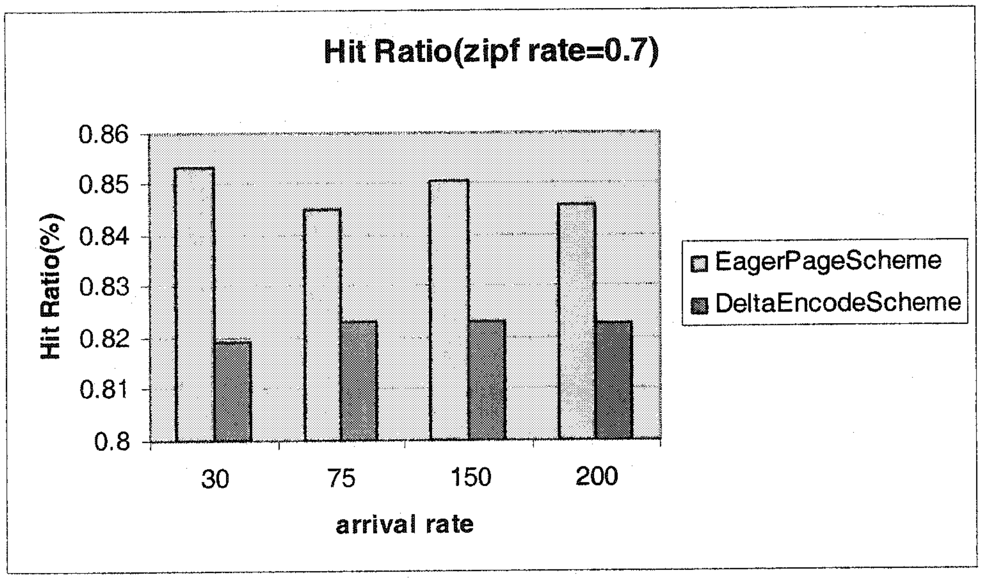


Figure4. 12 Hit Ratio

4.2.6 Web server cost analysis

Pre-computing **[30]** a dynamic page occurs whenever the page becomes stale. This is called instant pre-computing. Instant pre-computing is effective for a Web site in which each dynamic page is frequently accessed. In EPDC, we adopt instant pre-computing for eager-update pages. If a Web server knows that a page is an eager-update page, it pre-computes the new page when an update occurs and has the delta server calculate the delta between new and old versions of the page and send the delta to the proxy cache. Lazy-update pages, which are a large portion of dynamic pages of the Web server, are not accessed frequently and so demand on the server computing resource is limited. Delaying pre-computing of these pages can reduce resource contention.

By using this strategy, EPDC reduces the Web server cost. Let d be the average CPU time to compute a dynamic page on a machine, r be the total number of page requests and T be the total CPU time to compute all requested pages. The Web server cost without EPDC is:

$$T = r * d \qquad (1)$$

Under EPDC, let e be the total number of updates to eager-update pages at the Web server, er be the total number of requests for eager-update pages, lr be the total number of requests for lazy-update pages and $T1$ be the total CPU time to compute all requested pages with EPDC. We have:

$$T1 = e * d + lr * d \qquad (2)$$

Let $T2$ be the total CPU time saved by computing all request pages with EPDC. We have:

$$T2 = er * d - e * d \qquad (3)$$

From (2) and (3) we get:

$$T1 + T2 = e * d + lr * d + er * d - e * d = lr * d + er * d = (lr + er) * d = T$$

The rationale behind the above equation is that for frequently accessed Web pages, the Web server only generates the newest snapshot of the page once no matter how many requests for these pages come to the proxy cache. It also reduces resource contention while the lazy-update pages only are pre-computed only when clients' requests come. The space cost for the eager-update pages is proportional to the number of cached eager-update pages in the delta server. The space cost for the lazy-update pages is proportional to the number of URL lazy-update base files. Their space costs are small.

4.3 Summary

In this chapter we evaluated the performance of the proposed EPDC scheme. The performance of EPDC was compared to that of Delta-encoding and LRU via simulation. In addition, the adopted network model, simulation experiment settings, and simulation software implementation were described. The event-driven simulation experiment was conducted to investigate the effects of network link delay, HTTP request rate and changing workload on the performance of the Web dynamic caching schemes.

Simulation experiments show that EPDC outperforms the Delta-encoding and LRU schemes, with respect to HTTP request response time under various page request rates. Regardless of the network link delay, the response time of EPDC decreases faster than that of Delta-encoding and LRU schemes when the page request rate increases because the number of the more requests for eager-update pages also increases.

The results obtained from the experiments conducted on the effect of the network link delays showed that EPDC outperforms Delta-encoding and LRU for all values of link delay. When the link delay is large, EPDC saves more response time than Delta-encoding and LRU because there are more requests for the eager-update pages, which can be serviced without communication between the proxy cache server and the Web server.

EPDC can also to adapt to a workload in which the access probabilities to individual pages changes. EPDC dynamically identifies eager-update pages and lazy-update pages and it can stabilize the average response time automatically. Adopting the extended LRU replacement algorithm, EPDC also outperforms the Delta-encoding and LRU schemes in hit rates due to the requests for eager-update pages always get hit in the proxy cache server.

EPDC offloads the Web server's cost by using the pre-compute technique. For frequently accessed Web pages, the Web server only generates the newest snapshot of the page once no matter how many requests for these pages come to the proxy cache. It also reduces resource contention while the lazy-update pages are pre-computed only when clients' requests come.

Chapter 5

Conclusion

Traditional Web caching techniques are applicable only to static documents, or to documents that change in large timescales. Current caching solutions have reached a point where their performance cannot be significantly improved unless they incorporate a mechanism to "cache" dynamic documents. Dynamic caching schemes such as Delta-encoding can save network bandwidth and latency.

Our proposed Eager-update Page Dynamic Caching (EPDC) scheme combines advantages of both proxy side and server side caching. It not only improves critical response time and hit rate but also reduces the Web server processing time. EPDC uses the class-based delta-encoding technique to reduce network bandwidth and latency. Class-based delta-encoding means the server can designate a certain base instance of the dynamic page and arrange that every client have the same base page instance. The server needs to store only one base instance to be able to delta-encode future responses. The big advantage of base-instance caching is that it is transparent to Web developers who create content.

EPDC extends the HTTP protocol and the LRU replacement algorithm. Extended HTTP transfers eager-update and lazy-update pages class information between the

Web server and the proxy server. Although it adds the overhead to the HTTP header, it does not influence the communication between the proxy and the client. The extended LRU algorithm gives eager-update pages caching priority and improves the hit rate. Combining both techniques EPDC not only saves response time but also offloads the Web server processing time.

As a proxy side dynamic content caching scheme, EPDC provides the following benefits:

1. **Reduced Network bandwidth and latency.** In EPDC, the cache server saves base files and only retrieves deltas from the Web server.
2. **Reduced response time.** When the client sends a request for an eager page, the cache server always returns a cached copy to the client without connecting with the Web server.
3. **Keep consistency between cached pages and underlying data objects.** For eager pages, when the underlying data objects are updated in the back-end system, the Web server broadcasts the change to all registered cache servers using delta-encoding.
4. **Decreased Web server cost.** The Web server only needs to generate eager pages once when the underlying data objects change. This saves Web server processing time.
5. **Improved cache hit ratio.** In cache servers, eager pages always have priority so requests for eager pages have a higher cache hit.

A detailed simulation model was developed to study the performance of EPDC. EPDC was compared with two Web caching schemes, namely, Delta-encoding and LRU. Simulation results show that EPDC outperforms Delta-encoding and LRU with respect to HTTP request response time under various page request rates. Regardless of the network link delay, the response time of EPDC decreases faster than that of Delta-encoding and LRU when the page request rate increases. Regardless of the arrival rate, EPDC saves more response time than Delta-encoding and LRU when the link delay is large.

EPDC can also to adapt to a workload in which the access probabilities to individual pages changes and stabilize the average response time automatically. Adopting the extended LRU replacement algorithm, EPDC also outperforms the Delta-encoding and LRU schemes with respect to hit rates.

EPDC offloads the Web server's cost by using the pre-compute technique. For frequently accessed Web pages, the Web server only generates the newest snapshot of the page once no matter how many requests for these pages come to the proxy cache. It also reduces resource contention while the lazy-update pages are pre-computed only when clients' requests come.

Although EPDC outperforms existing Delta-encoding and LRU schemes, it still has some disadvantages. EPDC adds overhead to the HTTP header to communicate between proxies and Web servers. It also needs the Web server to maintain a list of proxy cache servers in order to transfer deltas. With EPDC, the proxy server returns a cached copy of eager-update pages to the client directly without contacting the Web server, which can create stale pages when race contention occurs. Under the CDN (content delivery network) system, there are many Web servers and Cache servers. If some pages are eager-update pages in some cache servers and lazy-update pages in other cache servers, we need to adjust the EPDC scheme to deal with it.

References

[1] ACE director "Load Balancing - Technical Specifications". Available at: http://www.verio.com/products/dedicated/infocenter/managed/specs.cfm

[2] Akamai Company, "Turbo-Charging Dynamic Web Sites with Akamai EdgeSuite", http://www.akamai.com/en/resources/pdf/Turbocharging_WP.pdf

[3] ArrowPoint Communications, "Content Smart Cache Switching", http://www.westcon.com/prodinfo/vertical/verticalesp/arrowpoint/appnotes/smart_caching.html

[4] G. Banga, F. Douglis and M. Rabinovich, "Optimistic Deltas for WWW Latency Reduction", *in Proceedings of USENIX Technical Conference*, '97.

[5] P. Barford and M. E. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," *in Proceedings of ACM SIGMETRICS'98*, 1998.

[6] C. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz, "The Harvest information discovery and access system", *in Proceedings of the Second International World Wide Web Conference,* October 1994.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "On the Implications of Zipf's Law for Web Caching", *in Technical Report 1371, University of Wisconsin*, April 1998.

[8] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf- like Distributions: Evidence and Implications", *in Proceedings of IEEE Infocom'99*, Mar. 1999.

[9] P. Cao, J. Zhang, and K. Beach, "Active Cache: Caching Dynamic Contents on the Web," *in Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, Mar. 1998.

[10] J. Challenger, A. Iyengar, and P. Dantzig, "A Scalable system for ConsistentlyCaching Dynamic Web Data," *in Proceedings of the IEEE INFOCOM'99*, Mar. 1999.

[11] C. Chiang, M. Ueno, M. Liu and M. Muller, "Modeling Web Caching Hierarchy Schemes", Technical Report, Ohio State University, OSU-CISRC-6/99-TR17, 1999.

[12] K. Claffy and D.Wessels, "ICP and the Squid Web Cache", *in IEEE Journal on Selected Areas in Communication*, April 1998.

[13] F. Douglis, A. Haro, and M. Rabinovich, "HPP:HTML macropreprocessing to support dynamic document caching", *in Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pp. 83-94.

[14] Edge(2001). Edge Side Includes-*http://www.esi.org/.*

[15] V. Holmedahl, B. Smith, and T. Yang, "Cooperative Caching of Dynamic Content on a Distributed Web Server," *in Proceedings of the Severth IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[16] B. Housel and D. Lindquist, "WebExpress: A System for Optimizing Web Brows-ing in a Wireless Environment", *in Proceedings of the ACM/IEEE 2nd Annual International Conference on Mobile computing and networking, MOBICOM*, November '96.

[17] J. Hunt, K. Vo and W. Tichy, "Delta Algorithms: An Empirical Analysis", *ACM Transactions on Software Engineering and Methodology*, April '98.

[18] J. W. Hunt and M. D. Mcillroy. "An algorithm for differential file comparison", *Technical Report Computing Science Technical Report 41*, Bell Laboratories, June 1976.

[19] J. W. Hunt and T. G. Szymanski. “A fast algorithm for computing longest common subsequences”, *Communications of the ACM*, 20(5):350-353, May 1977.

[20] Hypertext Transfer Protocol -- HTTP/1.1. Available at: http://www.cis.ohio-state. edu/htbin/rfc/rfc2068.html

[21] A. Iyengar and J. Challenger, “Improving Web Server Performance by Caching Dynamic Data”, *in Proceedings of USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[22] Z. Liang, H.Hassanein and P.Martin “Transparent Distributed Web Caching”, *in Proceedings of the IEEE Local Computer Network Conference*, Nov 2001, pp.225-233.

[23] E. McCreight. “A space economical suffix tree construction algorithm”, *Journal of the ACM*, 1976.

[24] J. C. Mogul, F. Douglis, A. Feldmann and B. Krishnamurthy, “Potential benefits of delta encoding and data compression for HTTP”, *in Proceedings of SIGCOMM*, 1997.

[25] E. O'Neil, P. O'Neil, G. Weikum. "The LRU-K Page Replacement Algorithm For Database Disk Buffering", *in Proceedings of ACM SIGMOD International Conference on Management of Data*, New York, 1993.

[26] K. Psounis, "Class-based Delta-encoding: A Scalable Scheme for Caching Dynamic Web Content," *in IEEE International Conference on Distributed Computing Systems Workshops*, Vienna, Austria, July 2002.

[27] M. J. Rochkind. "The source code control system", *IEEE Transactions on Software Engineering*, December 1975.

[28] A. Rousskov and V. Soloviev, "A performance study of the squid proxy on http/1.0", *in World Wide Web,* January 1999.

[29] A. Silberschataz, J. Peterson, and P. Galvin, "Operating System Concepts", Addison Wesley, 1992.

[30] B. Smith, A. Acharya, T. Yang, and H. Zhu, "Exploiting Result Equivalence in Caching Dynamic Web Content", *in Proceedings of Second USENIX Symposium on Internet Technologies and Systems(USITS99)*, Oct. 1999.

[31] SpiderCache Company, "SpiderCache Enterprise 2.0: Dynamic Content Delivered Faster", http://www.spidersoftware.com/Documents/WhitePaperSpiderCache20Ev1.pdf

[32] SSH Protocol. Available at: http://www.snailbook.com/protocols.html

[33] R. Tewari, H. Vin, A. Dan, and D. Sitaram, "Resource-Based Caching for Web Servers", *in Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking*, San Jose, CA, Jan. 1998, pp. 191-204.

[34] W. F. Tichy. "RCS-a system for version control", *Software-Practice and Experience*, 15(7):637-654, July 1985.

[35] W. F. Tichy. "The string-to-string correction problem with block moves", *ACM Transactions on Computer Systems*, 2(4):309-321, November 1984.

[36] A. Vahdat and T. Anderson, "Transparent Result Caching", *in Proceedings of 1998 USENIX Technical Conference*, 1998.

[37] V. Valloppillil and K. Ross, "Cache Array Routing Protocol v1.0". Internet Draft, draft-vinod-carp-v1-03.txt , February 1998.

[38] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin and H. Leny, "On the scale and performance of cooperative Web proxy caching", *in Proceedings of ACM Symposium on Operating Systems Principles*, '99.

[39] Q. Zhou, H. Hassanein and P.Martin "Transparent Web Caching with Minimum Response Time", *in Proceedings of the 22nd International Performance, Computing and Communications Conference(IPCCC 2003)*, Phoenix, May 2003.

[40] H. Zhu and T. Yang, "Class-based Cache Management for Dynamic Web Content", *in Proceedings of IEEE INFOCOM, '01.*

[41] J. Ziv and A. Lempel. "Compression of individual sequences via variable-rate coding", *IEEE Trans. on Information Theory*, IT-24(5):5306, September 1978.

Appendix A

Delta-encoding Algorithms

The following introduction of Delta-encoding algorithms is directly taken from J. Hunt, K. Vo and W. Tichys' "Delta Algorithms: An empirical analysis"[17].

Delta algorithms, i.e. algorithms that compute differences between two files or strings, have a number of uses when multiple versions of data objects must be stored, transmitted, or processed. The major early application of delta algorithms occurred in revision control systems such as SCCS and RCS [27,34]. By storing deltas relative to a base version, these systems save substantial amounts of disk space compared with storing every revision in its entirety. Much less information needs to be stored for each revision because changes from one revision to the next are typically small. Other well known applications are the display of differences between files and the merging together of the changes in two different files relative to a common base.

The classic program for generating deltas is Unix *diff* [18,19]. Both SCCS and RCS use it for storage and display of differences; RCS also uses it for merging. Since *diff* is limited to text files, so are SCCS and RCS. However, users wish to place binary code under revision control as well, not just source text. A simple technique is to map the binary code into text and then apply *diff*. While this works reliably and is widely used in practice, the deltas produced are typically larger than the originals! Newer algorithms such as *bdiff*[35] and *vdelta*[35] do not exhibit this problem. Unlike *diff*, these algorithms are applicable to any byte stream. They exploit reordering of blocks to produce short differences.

Bdiff and *vdelta* offer additional compression on the resultant delta. For this reason, *diff* coupled with *gzip* post processing is included in the study as well. All algorithms run enough faster than a byte-level LCS computation to have practical applications.

Both *bdiff* and *vdelta* are comparable in utility to *diff*. Not only do they produce deltas suitable for compression and 3-way file-merging, but their output can also be used to display differences. Since *bdiff* and *vdelta* break lines apart, minor postprocessing of the deltas is needed to produce output identical to *diff's*. Other human-readable output that takes advantage of the finer granularity of the output of these algorithms can be produced using color coding techniques. In addition, both *bdiff* and *vdelta* can compress a single file with itself. We introduce both *bdiff* and *vdelta* as follows:

A. *BDIFF*

Bdiff is a modification of W. F. Tichy's block-move algorithm[35]. It uses a two stage approach. First it computes the difference between the two files. Then it uses a second step to compress the resulting difference description. These two parts run concurrently in that the first stage calls the second each time it generates output.

In the first phase, *bdiff* builds an index, called a suffix tree, for the first file. This tree is used to look up blocks, i.e. every possible match is examined to ensure that the longest possible match is found. The output from this phase is a sequence of copy blocks and character insertions that encode the second file in terms of the first. It can be shown that the algorithm produces the smallest number of blocks and runs in linear time. It also discovers crossing blocks, i.e. blocks whose order were permuted in the second file.

The second phase efficiently encodes the output of the first. A block is represented as a length and an offset into the first file. Characters and block lengths are encoded in the same space by adding 253(256 minus the three unused lengths) to lengths before encoding. Blocks of lengths less than four bytes are converted to character insertions. Characters and lengths are then encoded using a common splay tree. The splay tree is used to generate a character encoding that ensures that frequently encoded characters are shorter than uncommon characters. Splay trees dynamically adapt to the statistics of the source without requiring an extra pass. A separate splay tree encodes the offsets.

Bdiff actually uses a sliding window of 64 Kbytes on the first file, moving it in 16 Kbytes increments. This means that the first phase actually builds four suffix trees that index 16 Kbytes each of the first file. The window is shifted forward whenever the encoding of the second file crosses a 16 Kbytes boundary, but in such a fashion that the top window position in the first file is always at least 16 Kbytes ahead of the current encoding position in the second file. Whenever the window is shifted, the oldest of the four suffix trees is discarded and a new one built in its space. The decoder has to track the window shifts, but does not need to build the suffix trees. Position information is given as an offset from the beginning of the window.

B. *VDELTA*

Vdelta is a new technique that combines both data compression and data differencing. It is a refinement of W. F. Tichy's block-move algorithm[35], in that, instead of a suffix tree, *vdelta* uses a hash table approach inspired by the data parsing scheme in the 1978 Ziv-Lempel compression technique**[41]**. Like block-move, the Ziv-Lempel technique is also based on a greedy approach in which the input string is parsed by longest matches to previously seen data. Both Ziv-Lempel and block-move techniques have linear-time implementations**[23]**. However, implementations of both of these algorithms can be memory intensive and, without careful consideration, they can also be slow because the work required at each iteration is large. *Vdelta* generalizes Ziv-Lempel and block-move by allowing for string matching to be done both within the target data and between a source data and a target data. For efficiency, *vdelta* relaxes the greedy parsing fule so that matching prefixes are not always maximally long. This modification allows the construction of a simple string matching technique that runs efficiently and requires minimal main memory.

B.1 Building Difference

For encoding, data differencing can be thought of as compression, where the compression algorithm is run over both sequences but output is only generated for the second sequence. The idea is to construct a hash table with enough indexes into the four bytes starting at that position. In order to break a sequence into fragments and construct the necessary hash table, the sequence is processed from start to end; at each step the hash table is searched to find a match. Processing continues at each step as follows:

(1) if there is no match,

   (a) insert an index for the current position into the hash table,

   (b) move the current position forward by 1, and

   (c) generate an insert when in output mode; or

(2) if there is a match,

   (a) insert into the hash table indexes for the last 3 positions of the matched portion,

   (b) move the current position forward by the length of the match, and

   (c) generate a copy block when in output mode.

Each comparison is done by looking at the last three bytes of the current match plus one unmatched byte and checking to see if there is an index in the hash table that corresponds to a match. The new match candidate is checked backward to make sure that it is a real match before matching forward to extend the matched sequence. If there is no current match, i.e. just starting a new match, use the 4 bytes starting at the current position.

As an example, assume the sequence below with the beginning state as indicated (the $\Downarrow$ indicates the current position):

```
⇓
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
b  c  d  e  a  b  c  d  a  b  c  d  a  b  c  d  e  f  g  h
```

The algorithm starts at position 0. At this point the rest of the sequence is the entire sequence so there is no possible match to the left. Case 1 requires position 0 to be entered into the hash table (indicated with a * under it) then to advance the current position by 1.

```
   ⇓
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
b  c  d  e  a  b  c  d  a  b  c  d  a  b  c  d  e  f  g  h
*
```

This process continues until position 8 is reached. At that time, we have this configuration:

```
                        ⇓
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
b  c  d  e  a  b  c  d  a  b  c  d  a  b  c  d  e  f  g  h
*  *  *  *  *  *  *  *
```

Now the rest of the sequence is “abcdabcdedfg”. The longest possible match to some part previously processed is “abcdabcd” which starts at location 4. Case 2 dictates entering the last 3 positions of the match (i.e. 13, 14, 15) into the hash table, then moving the current position forward by the length of the match. Thus the current position becomes 16 in this example.

```
                                                ⇓
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	c	d	e	a	b	c	d	a	b	c	d	a	b	c	d	e	f	g	h
*	*	*	*	*	*	*	*						*	*	*				

The final step is to match "efgh" and that fails so the last mark is on position 16. The current position moves to position 17 which now does not have enough data left for the next hash code so the algorithm stops after outputting the last three characters.

																	⇓		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	c	d	e	a	b	c	d	a	b	c	d	a	b	c	d	e	f	g	h
*	*	*	*	*	*	*	*						*	*	*	*			

Note that the above matching algorithm will actually find the longest match if indexes are kept for every location in the string. The skip in step 2b prevents the algorithm from being able to always find the longest prefix; however, this rule saves considerable processing time and memory space. In fact, it is easy to see from the above construction rules that the space requirement is directly proportional to the output. The more compressible a target data set is, the faster it is to compress it.

B.2 Difference Encoding

In order to minimize the output genereated, the block-move list generated above must be encoded. The output of *vdelta* consists of two types of instructions: **add** and **copy**. The **add** instruction has the length of the data followed by the data itself. The **copy** instruction has the size of the data followed by its address. Two caches are maintained as references to minimize the space required to store this address information.

Each instruction is coded starting with a control byte. Eight bits of the control byte are divided into two parts. The first 4 bits represent numbers from 0 to 15, each of which defines a type of instruction and a coding of some auxiliary information. Below is an enumeration of the first 10 values of the first 4 bits:

0: an **add** instruction,

1,2,3: a **copy** instruction with position in the **QUICK** cache,

4: a **copy** instruction with position coded as an absolute offset from the beginning of the file,

5: a **copy** instruction with position coded as an offset from current location, and

6,7,8,9: a **copy** instruction with position in the **RECENT** cache.

For the **add** instruction and the **copy** instructions above, the second 4 bits of the control byte, if not zero, code the size of the data involved. If these bits are 0, the respective size is coded as a subsequent sequence of bytes.

The above mentioned caches-**QUICK** and **RECENT**-enable more compact coding of file positions. The **QUICK** cache is an array of size 768 (3* 256). Each index of this array contains the value *p* of the position of a recent copy instruction such that *p* modulo 768 is the array index. This cache is updated after each copy instruction is output (during coding) or processed (during decoding). A copy instruction of type 1, 2, or 3 will be immediately followed by a byte whose value is from 0 to 255 that must be added to 0, 256 or 512 respectively to compute the array index where the actual position is stored. The **RECENT** cache is an array with 4 indices storing the most recent 4 copying positions. Whenever a **copy** instruction is output (during coding) or processed (during decoding), its copying position replaces the oldest position in the cache. A **copy** instruction of type 6,7,8, or 9 corresponds to cache index 1, 2, 3, or 4 respectively. Its copying position is guaranteed to be larger than the position stored in the corresponding cache index and only the difference is coded.

It is a result of this encoding method that an **add** instruction is never followed by another **add** instruction. Frequently, an **add** instruction has data size less than or equal to 4 and the following **copy** instruction is also small. In such cases, it is advantageous to merge the two instructions into a single control byte. The values from 10 to 15 of the first 4 bits code such merged pairs of instructions. In such a case, the first 2 bits of the second 4 bits in the control byte code the size of the **add** instruction and the remaining 2 bits code the size of the **copy** instruction. Below is an enumeration of the values from 10 to 15 of the first 4 bits:

10: a merged **add/copy** instruction with copy position coded as itself,

11: a merged **add/copy** instruction with copy position coded as difference from the current position.

12,13,14,15: a merged **add/copy** instruction with copy position coded from a **RECENT** cache.

In order to elucidate the overall encoding scheme, consider the following files:

Version1: a b c d a b c d a b c d e f g h

Version2: a b c d x y x y x y x y b c d e f

The block-move output would be

1. **copy**	4	0		01000100 0
2. **add**	2	"xy"	which encodes to	00000010 "xy"
3. **copy**	6	20	(instruction in binary)	01000110 20
4. **copy**	5	9		01000101 9

Note that the third instruction copies from Version2. The address 20 for this instruction is 16 + 4 where 16 is the length of Version1. Note also that the data to be copied is also being reconstructed. That is, *vdelta* knows about periodic sequences.

This output encoding is independent of the way the block-move lists are calculated, thus *bdiff* could be modified to use this encoding and *vdelta* could be modified to use splay coding.

Appendix B

HTTP Extensions

After we classify the pages, we need to send the page class information to the Web server. We can extend HTTP to do this job. In HTTP/1.1, the general-header field is as follows:

```
General-header = Cache-Control
                 | Connection
                 | Date
                 | Pragma
                 | Transfer-Encoding
                 | Upgrade
                 | Via
```

The Cache-Control general-header field is used to specify directives that must be obeyed by all caching mechanisms along the request/response chain. Cache directives are unidirectional in that the presence of a directive in a request does not imply that the same directive should be given in the response. The detail of Cache-Control header is as follows:

```
Cache-Control = "Cache-Control" ":" 1#cache-directive

Cache-directive= cache-request-directive

                  | cache-response-directive

cache-request-directive =

                    "no-cache" [ "=" <"> 1#field-name<"> ]

                  | "no-store"

                  | "max-age" "=" delta-seconds

                  | "max-stale" [ "=" delta-seconds ]

                  | "min-fresh" "=" delta-seconds

                  | "only-if-cached"

                  | cache-extension

cache-response-directive =

                    "public"

                  | "private" [ "=" <"> 1#field-name <"> ]

                  | "no-cache" [ "=" <"> 1#field-name<"> ]

                  | "no-store"

                  | "no-transform"

                  | "must-revalidate"

                  | "proxy-revalidate"

                  | "max-age" "=" delta-seconds

                  | cache-extension

cache-extension = token [ "=" ( token | quoted-string ) ]
```

The Cache-Control header field can be extended through the use of one or more cache-extension tokens, each with an optional assigned value. Informational extensions (those which do not require a change in cache behavior) may be added without changing the semantics of other directives. Behavioral extensions are designed to work by acting as modifiers to the existing base of cache directives. Both the new directive and the standard directive are supplied, such that applications which do not understand the new directive will default to the behavior specified by the standard directive, and those that understand the new directive will recognize it as modifying the requirements associated with the standard directive. In this way, extensions to the Cache-Control directives can be made without requiring changes to the basic protocol.

So we can use the cache-extension mechanism to transfer page class information. Our extension is as follows:

```
cache-extension = class
class =
    "eager"
  | "base"
  | "delta"
  | "from cache"
```

For an eager-update page, a proxy sends a **GET HTTP** request to the Web server. In the general header field it has:

Cache-Control: class = "eager"

This means after page classifying the proxy identifies this page as an "eager-update" page. The delta server needs to save a base file and broadcast it to all proxies in the page list with the following message in the general header:

Cache-Control: class = "base"

The delta server also needs to generate deltas between base file and the new version of the page then send deltas to the proxy with the following message in the general header:

Cache-Control: class = "base"

In the implementation, the delta server sends request to the Web server asking for new version of the eager-update pages. To differentiate the request for eager-update pages from proxies, the following message is inserted in the general header:

Cache-Control: class = "from cache"

The cache-extension mechanism helps us to only modify a little in HTTP protocol header field to communicate between proxies and Web servers.

Appendix C

The Flow Charts, Simulation Software Structure and Pseudo-Code of EPDC

In this study, we evaluate the performance of EPDC, Delta-Encoding and LRU dynamic Web caching schemes. The HTTP request processing flow charts of the three schemes are shown in Figure C.1 to Figure C.3. These flow charts are drawn according to the scheme descriptions in Chapter 3.

In the EPDC scheme, when a proxy cache server receives a HTTP GET request, it searches its cache for the requested object. If the object is not a hit, the proxy forwards it to the Web server to get a new page back. If the requested object is found in the cache, the proxy cache server checks if it is an eager-update page or a lazy-update page. If the page is an eager-update page, the proxy returns the object to the requesting client. Otherwise, the proxy cache server forwards that lazy-update page to the Web server to check if it is stale, if not, the Web server sends a "304 Not-modified" message back to the proxy. If the page is stale, the Web server asks the delta-server to generate the delta and returns it back to the proxy (Figure C.1).

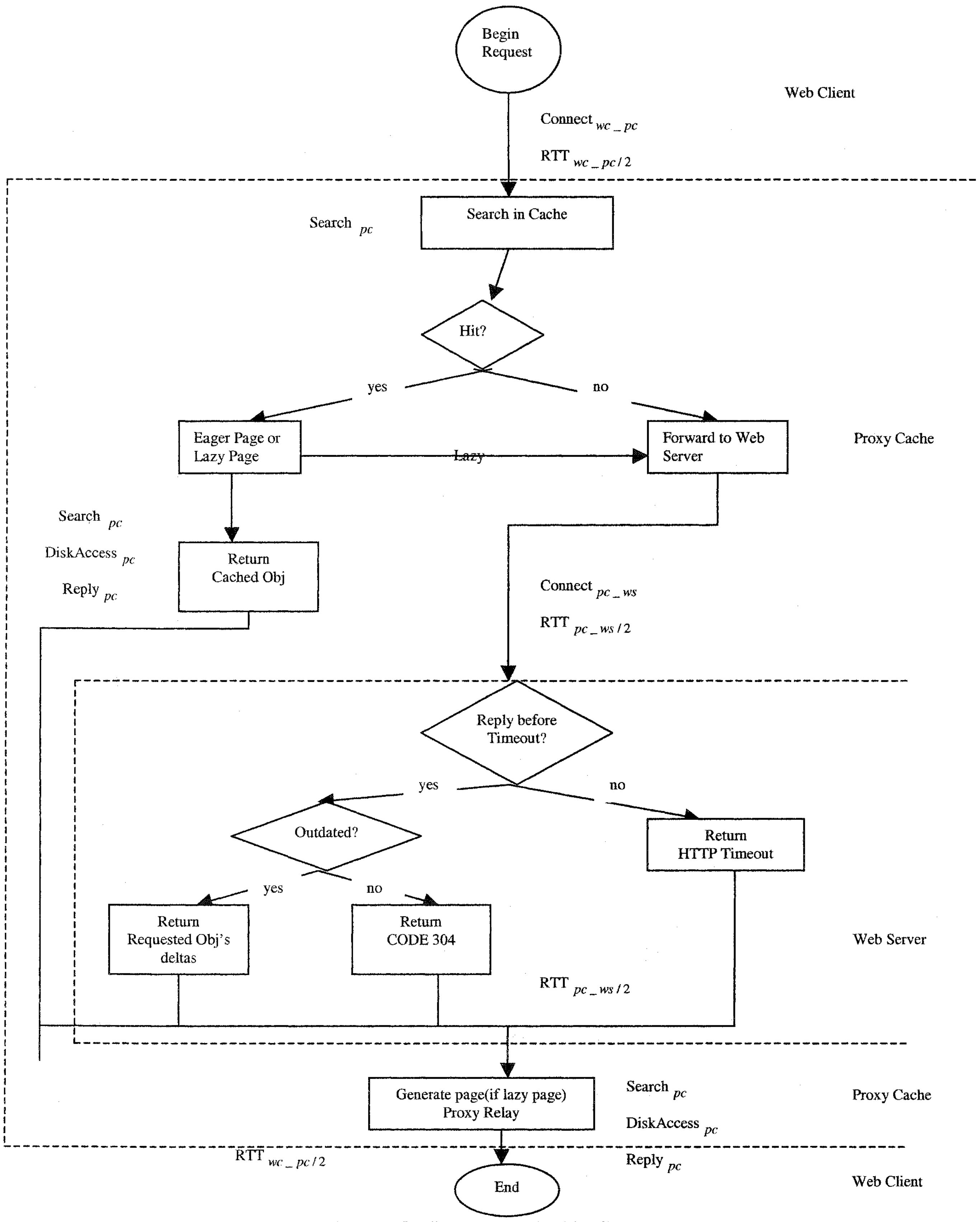


Figure C. 1 HTTP GET request in EPDC scheme

In the Delta-encoding scheme, when a proxy cache server receives a HTTP GET request, it searches its cache for the requested object. If the object is not hit, the proxy forwards it to the Web server to get a new page base file and deltas back. If the requested object is found in the cache, the proxy cache server still needs to forwards that page to the Web server to check if it is stale, if not, the Web server sends a "304 Not-modified" message back to the proxy. If the page is stale, the Web server asks the delta-server to generate the delta and returns it back to the proxy (Figure C.2).

In the LRU scheme, when a proxy cache server receives a HTTP GET request, it searches its cache for the requested object. No matter if the object is hit or not, the proxy forwards it to the Web server to check if it is stale, if not, the Web server sends a "304 Not-modified" message back to the proxy. If the page is stale, the Web server returns a new page back to the proxy (Figure C.3).

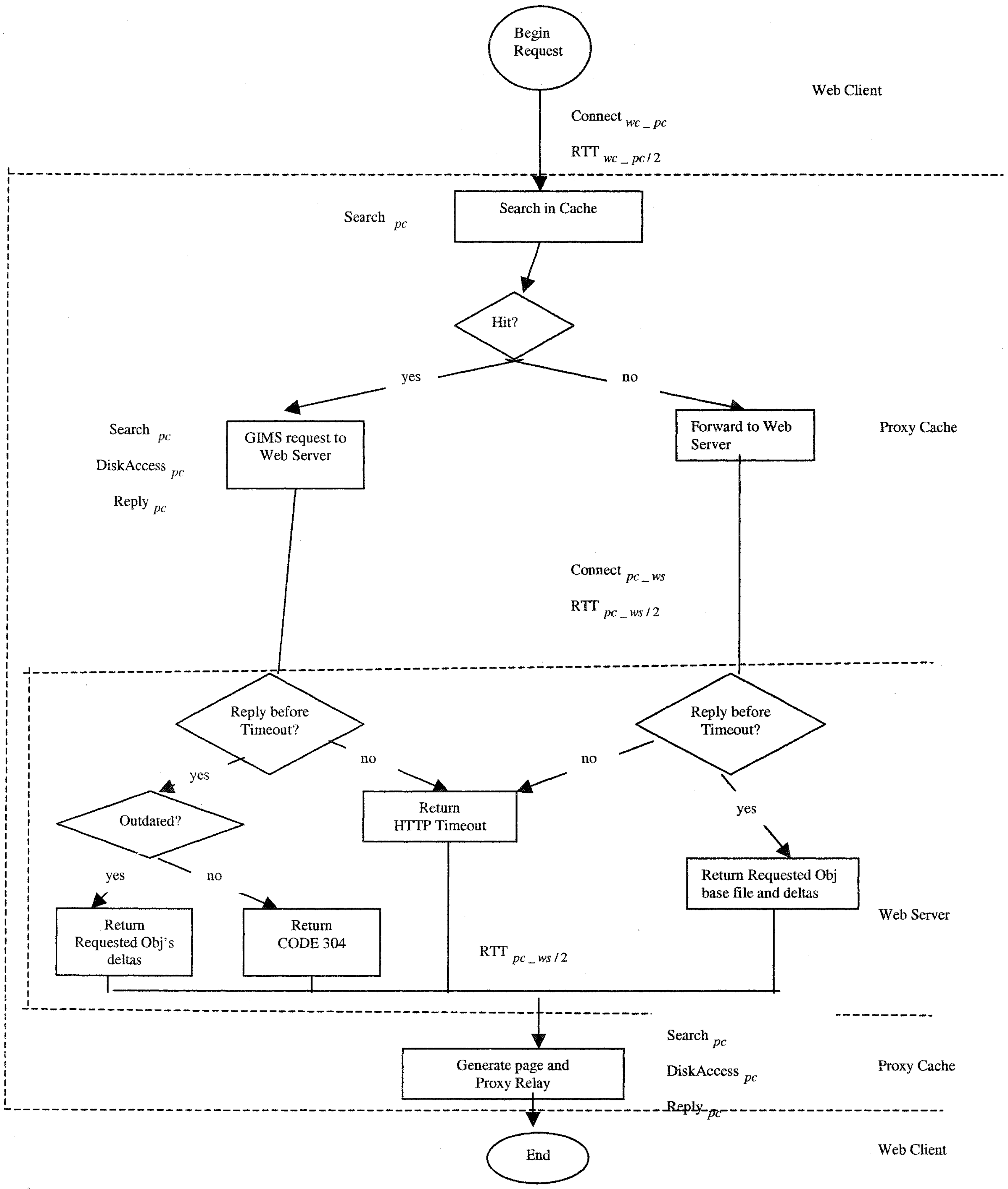


Figure C. 2 HTTP GET request in Delta-encoding scheme

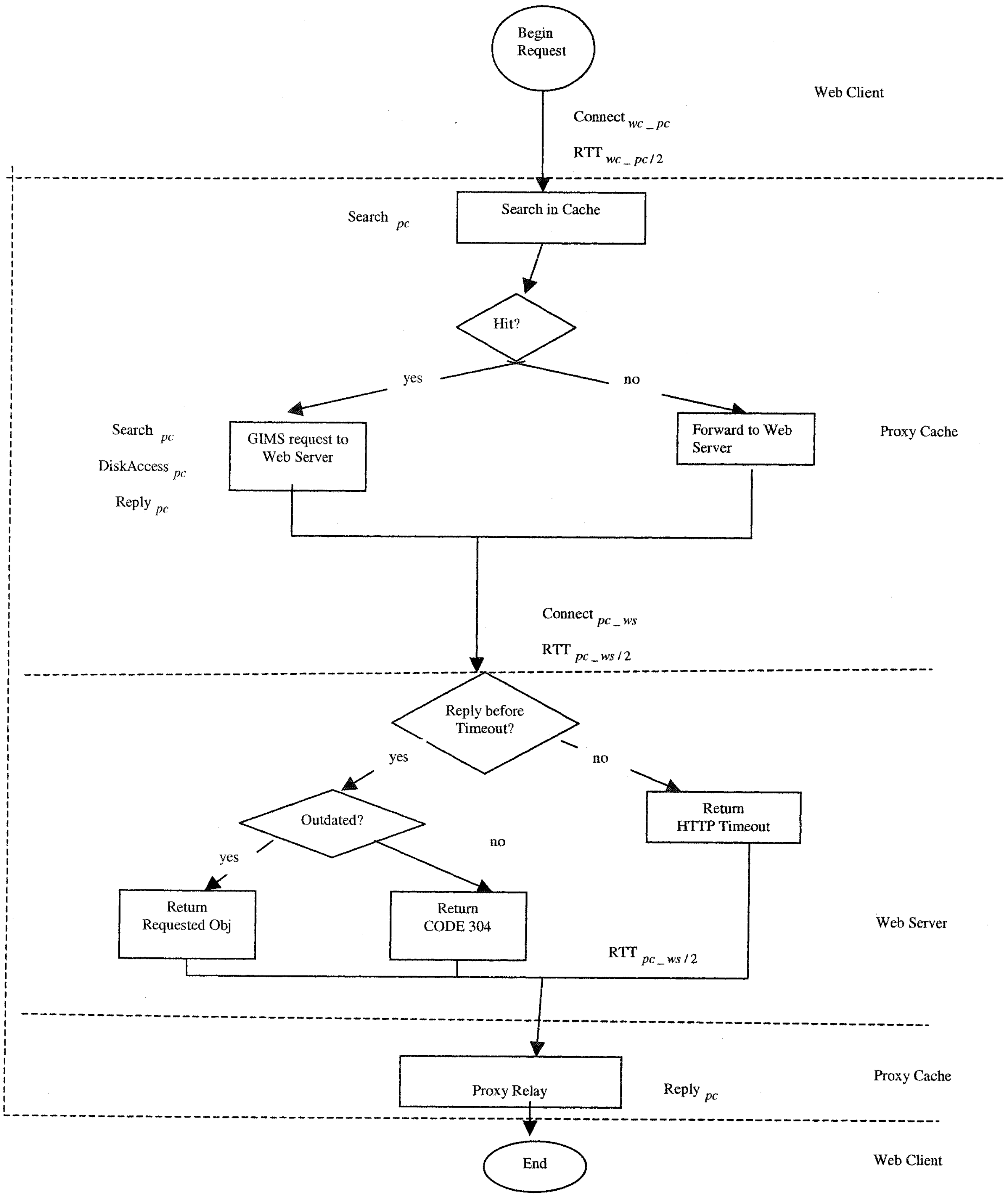


Figure C. 3 HTTP GET request in LRU scheme

The simulator used in this thesis conducts discrete event driven simulation. It is developed using the Java programming language. The simulation software consists of the following five major components:

- Client Cluster: responsible for simulating a cluster of clients. It generates HTTP request traffic using the request logs from workload text files.
- Dynamic Proxy Cache Server: responsible for simulating a proxy cache server. It has the following functions:
    - Pass the client's HTTP request to Web server. Calculate the client request number and page update number. Identity the eager-update page and lazy-update page.
    - Send HTTP GET request with page class information to the Web server. Save the base-file and combine it with the delta to construct a dynamic page. Send HTTP response to the client.
- Web Server: responsible for simulating a Web server. It accepts HTTP requests, and then sends back base file and delta to the proxy cache server through the Delta Server.
- Delta Server: responsible for simulating a delta server. It is integrated with the Web server. When it receives the dynamic page from the Web server, it has the following functions:
    - For eager-update page, it saves it as base file and broadcast it to the proxy cache servers registered in the page list. The next time the same

new page comes from the Web server, it will save a new page as base file and broadcast the delta.

  - o For lazy-update page, it saves it and then sends it back to the proxy cache server. The delta server will choose optimal base file uses class-based delta encoding mechanism and broadcast it to all registered proxy cache server. So when the request comes again, it will calculate the delta between the base file and the newest dynamic page and then sends it back to the proxy cache server.

- Event Manager: responsible for simulation event queuing and dispatching. All simulation events are handled by the event manager.

The pseudo-code of algorithms of message-receiving at the proxy, the Web server and the delta-server are described from the Figure C.4 to Figure C.6.

```
Process ReceiveMessageFromWebserver(msg: HTTPMessage)

//To check if the request URL is in local cache
for i = 1 to NumOfCachedPage do
   if (CachedPageArray[i].URL == msg.requestURL)
      cspage = CachedPageArray[i];
      break;
   endif
endfor



//Receive message from Webserver
    switch(msg.OPCode)
       //if the request is PUT, there are two conditions
       case HTTP_PUT_BASE_REQUEST:
            //if the message class is base file, save the file and response 200 OK
            LRUCache.addToCache(msg.data);
            sendMessage(WebserverURL, HTTP_200_RESPONSE);

       //if the message class is delta, construct the page and add to cache
       case HTTP_PUT_DELTA_REQUEST:
            consturctPage(msg.senderURL);
            sendMessage(WebserverURL, HTTP_200_RESPONSE);
            LRUCache.addToCache(msg.data);

       //if the reponse is not modified message, send cached copy to client
       case HTTP_304_RESPONSE:
            sendMessage(clientId, HTTP_200_RESPONSE);

       //if the reponse is base, add base file to cache
       case HTTP_BASE_RESPONSE:
            LRUCache.addToCache(msg.data);

       //if the response is new object, save it and add page update no.
       //then send client new document
       case HTTP_200_RESPONSE:
            cspage.page_update_num++;
            LRUCache.addToCache(msg.data);
            sendMessage(clinetId, HTTP_200_RESPONSE);
    endswitch
end
```

Figure C. 4 The algorithm for proxy receiving a message from Webserver

```
Process OnReceiveMsgAtWebserver(msg: HTTPMessage)

//Register the proxy
registerProxy(proxyId);

  switch (msg.OPCode)
    //The GET immediate page request from proxy
    case HTTP_GET_IMM_REQUEST:
      //For each immediate page, register a proxy client list
      registerPageClientList(pageURL,proxyId);
      generatePage(pageURL);
      sendToDeltaServer(page.data);
    //The GET lazy page request from proxy
    case HTTP_GET_IMS_REQUEST:
      if (page.isNotUpdate())
        sendMessage(proxyId, HTTP_304_RESPONSE);
      else
        genaratePage(pageURL);
        sentToDeltaServer(page.data);
      endif
    //The GET immediate page request from delta server
    case HTTP_GET_FROMCACHE_REQUEST:
      generatePage(pageURL);
      sendToDeltaServer(HTTP_200_RESPONSE);
  endswitch
endif
end
```

Figure C. 5 The algorithm for Web server receiving an HTTP-message

```
Process DeltaProcessing(message: pageData)
for i=1 to NumOfCachedPage do
    if (CachedPageArray[i].URL==msg.requestURL)
        cspage = CachedPageArray[i];
        break;
    endif
endfor

swich (OpCode)
     case HTTP_GET_IMM_REQUEST:
          saveBase(pageData);
          sendMessage(proxyId, HTTP_BASE_RESPONSE);
          broadcastMessage(pageClientList, HTTP_PUT_BASE_REQUEST);

     case HTTP_200_RESPONSE:
          saveBase(pageData);
          calculateDelta(baseFile, pageData);
          broadcastMessage(pageClientList, HTTP_PUT_DELTA_REQUEST);

     case HTTP_GET_IMS_REQUEST:
          savePage(pageData);
          sendMessage(proxyId, HTTP_200_RESPONSE);
          calculateLazyBase();
endswitch
end
```

Figure C. 6 The algorithm for Delta server receiving an HTTP message

Appendix D

Confidence Intervals

Normally, confidence intervals placed on the mean values of simulation results can be used to describe the accuracy of the simulation results. Consider the results of N statistically independent simulation runs for the same experiment: $X_1, X_2, \ldots, X_N$. The sample mean, $\overline{X}$ is given as:

$$\overline{X} = \frac{\sum_{i=1}^{N} X_i}{N}$$

The variance of the distribution of the sample values, S_x^2 is:

$$S_x^2 = \frac{\sum_{i=1}^{N} (X_i - \overline{X})^2}{N-1}$$

The standard derivation of the sample mean is given by: $\frac{S_x}{\sqrt{N}}$.

Under the assumption of independence and normality, the sample mean is distributed in accordance to the Normal-Distribution, which means the sample mean of the simulation runs fall in the interval $\pm\varepsilon$ within the actual mean with a certain probability drawn from the T-Distribution.

$$\varepsilon = \frac{S_x t_{\alpha/2, N-1}}{\sqrt{N}}$$

where $t_{\alpha/2,N-1}$ is the value of the T-distribution with N-1 degrees of freedom with probability $\alpha/2$.

The upper and lower limits of the confidence interval regarding the simulation results are:

$$\text{Lower Limit} = \overline{X} - \frac{S_x t_{\alpha/2,N-1}}{\sqrt{N}}$$

$$\text{Upper Limit} = \overline{X} + \frac{S_x t_{\alpha/2,N-1}}{\sqrt{N}}$$

Where $t_{\alpha/2,N-1}$ is the upper $\alpha/2$ percentile of the t-distribution with N-1 degrees of freedom.

The simulation experiments in this thesis were run with a 90% confidence level with 10% confidence intervals for each data point. The number of simulation runs has been chosen big enough to ensure stability and tight confidence intervals.