

# Deterministic Object Management in Large Distributed Systems

by

Mikhail S. Mikhailov

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

---

February 20, 2003

**APPROVED:**

---

Prof. Craig E. Wills  
Advisor

---

Dr. Balachander Krishnamurthy  
External Committee Member  
AT&T Labs–Research, Florham Park, NJ

---

Prof. David Finkel  
Committee Member

---

Prof. Micha Hofri  
Head of Department

---

Prof. Robert E. Kinicki  
Committee Member

# Abstract

Caching is a widely used technique to improve the scalability of distributed systems. A central issue with caching is maintaining object replicas consistent with their master copies. Large distributed systems, such as the Web, typically deploy heuristic-based consistency mechanisms, which increase delay and place extra load on the servers, while not providing guarantees that cached copies served to clients are up-to-date. Server-driven invalidation has been proposed as an approach to strong cache consistency, but it requires servers to keep track of which objects are cached by which clients.

We propose an alternative approach to strong cache consistency, called MONARCH, which does not require servers to maintain per-client state. Our approach builds on a few key observations. Large and popular sites, which attract the majority of the traffic, construct their pages from distinct components with various characteristics. Components may have different content types, change characteristics, and semantics. These components are merged together to produce a monolithic page, and the information about their uniqueness is lost. In our view, pages should serve as containers holding distinct objects with heterogeneous type and change characteristics while preserving the boundaries between these objects. Servers compile object characteristics and information about relationships between containers and embedded objects into explicit object management commands. Servers piggyback these commands onto existing request/response traffic so that client caches can use these commands to make object management decisions.

The use of explicit content control commands is a deterministic, rather than heuristic,

object management mechanism that gives content providers more control over their content. The deterministic object management with strong cache consistency offered by MONARCH allows content providers to make more of their content cacheable. Furthermore, MONARCH enables content providers to expose internal structure of their pages to clients.

We evaluated MONARCH using simulations with content collected from real Web sites. The results show that MONARCH provides strong cache consistency for all objects, even for unpredictably changing ones, and incurs smaller byte and message overhead than heuristic policies. The results also show that as the request arrival rate or the number of clients increases, the amount of server state maintained by MONARCH remains the same while the amount of server state incurred by server invalidation mechanisms grows.

# Acknowledgements

A Ph. D. dissertation is a multi-year venture. My life<sup>1</sup> and my work during these years have been influenced in various ways by numerous people, animals, and things. In what follows I attempt to acknowledge those whose influence has been important to me for various reasons. I apologize in advance if at this final stage in my work on the dissertation I am forgetting to acknowledge you. Your influence remains important nonetheless.

I thank Prof. Craig E. Wills for being my advisor, for working with me over the years, and for putting up with me. I also thank my committee members, Prof. David Finkel, Prof. Robert E. Kinicki, and Dr. Balachander Krishnamurthy for serving on my committee and for providing valuable comments on my dissertation proposal and the dissertation itself that substantially improved my work. In addition, I thank Profs. Kinicki and Finkel for being my first advisor and co-advisor respectively, before Prof. Wills.

I thank other faculty members at the Computer Science Department at WPI and members of the PEDS (Performance Evaluation of Distributed Systems) Research Group for providing a friendly, creative, and collaborative environment.

I would very much like to acknowledge all the assistance that I received from the system administration staff in the Computer Science Department at WPI. In particular, I thank Mike Voorhis and Jesse Banning, who always answered numerous questions of mine, investigated various issues at my requests, and cut me quite a bit of slack. Furthermore, they let me use the powerful `q1` box for the study described in Chapter 6.

---

<sup>1</sup>Was I supposed to have one?

My studies at WPI have been supported by different organizations, both directly and in the form of summer jobs. I would like to thank Tandem Computer, Stratus Computer, BMC Software, ArrowPoint Communications (now Cisco Systems), and the National Science Foundation (grant CCR-9988250) for providing financial support for my studies. I would like to thank folks at these companies who gave WPI research grants, and also people at WPI who served as principal investigators on these research projects. In particular, I thank Prof. David Finkel, Prof. Robert E. Kinicki, Prof. Mark Claypool, and Prof. Craig E. Wills for getting the grants. I also want to thank Ervin Johnson who was my manager at ArrowPoint Communications and who was the key factor in ArrowPoint supporting me for one year at WPI. In addition, I would like to thank other people at ArrowPoint with whom I worked, especially Chris Collins, Walter Kelt, Raj Nair, Peter Salinger, and Charles Teague.

I would like to thank Prof. Micha Hofri for allowing me to teach a senior-level undergraduate class (CS4241 Webware) at WPI (three times), and for helping me with various student-related issues. I would also like to thank Prof. Craig E. Wills and Prof. David Finkel for answering teaching-related questions and for their help in dealing with student issues. I am very grateful to Prof. David Finkel for giving me his Webware course materials to jump-start my preparation for the course.

Many people surrounded me outside of WPI and I would like to acknowledge them too. My biggest “**Thank you**” of all most certainly, without a hint of a doubt, goes to my parents, Сергею и Татьяне Михайловым. I would not even be writing this text if it was not for them. And I do not simply mean bringing me into this world. I mean absolutely everything that happened since then, at every stage in my life. Listing everything they have done for me would be too voluminous even for a separate biography publication. Here, I am explicitly acknowledging their vision for the future, constant prodding me in the right direction, non-stop care and encouragement, continuous concern for and involvement in my life, and their relentless pursuit to make me a better person—even across the Atlantic

Ocean. Thanks, guys, I am indebted forever.

I thank Janet Burge for being my girlfriend for the past five years, for her support and encouragement, for providing me with a very pleasant escape from the realities of Worcester, for pushing me to do things I thought I did not like, for joining me on a few trips within the US and abroad, for being my dive buddy, for proof reading numerous drafts of my papers and this dissertation, for making yummy cookies for my Ph. D. defense, and for many, many, many other things. Many thanks, Janet!

I also want to thank all the relatives and family friends back home, who always remember me and want to see me when I visit. I specifically thank Юрия Ивановича Зорина for the Матрёшка that he gave me in April 2002 at the Шинок restaurant and the words that accompanied his present.

Аркадий Синецын deserves a special thank you. For me, he has always been an example of a person who has found his place in life, who knows how to work hard, get results, and play hard. An example of a scientist, a researcher. An example of someone who enjoys life and always has a positive outlook on it. An example of someone who drinks large beers, because “who would drink a small one.”

I also thank Юрия Андреевича Тимофеева for being a knowledgeable, intelligent, interesting to talk to person, and for letting me use his laptop computer in the early 1990s when I did not have my own computer.

I am grateful to my landlord Prof. John Wilkes and his wife Sandra Ansaldi, who served as peers for various discussions, serious and otherwise, as well as numerous jokes. They provided countless entertainment as well as shelter from the elements. I thank Keith McKormick, who introduced me to many things American, such as tailgating, couscous, and Pimsleur language tapes. Keith took me to celebrate my first Thanksgiving and Christmas with his relatives. Keith also accompanied me on a trip across a few states and constantly asked to slow down on the foggy Blue Ridge Parkway. Robert Lee Danley was one of my best roommates at John’s house. We had a blast watching “Mad About You” at 11 pm

and I had fun eating Robert's chicken wings. Robert made me wear a tuxedo by inviting me to be in his wedding party.

In addition to all the people who were part of my life during these years, I would like to acknowledge a few furrrrrry friends. Thanks go to the late Cija, the cat, who always liked to be chased. It was extremely tough to see her go. . . . "Adorable feline Caspar," as one of my Webware students called him, now looks like a rabbit, but is a great cat anyway, especially when he rolls on the carpet. My appreciation also goes to the smart, sneaky, and ferocious beast Luna, the cat, who likes to hunt you down, or jump on the staircase pole and be patted, and often sleeps with all four paws up in the air. My life has also been enriched by squirrels, chipmunks, hawks, and other birds, animals, insects, and reptiles inhabiting Worcester, Stow, the area adjacent to Route 290, and numerous Audubon Sanctuaries. The sleepy opossum from the nature center in Kalamazoo, MI, deserves a special mention.

Last, but not least, I would like to thank the developers of and contributors to the following Open Source Software that I used while working on this dissertation: Linux, various GNU tools and utilities, Gnome, Emacs, Perl, Apache, MySQL, Galeon, Mozilla, Gimp, and L<sup>A</sup>T<sub>E</sub>X.

Thank you all!

I have looked at a couple Ph. D. dissertations and saw people promise their friends and relatives that they will not be working on another dissertation any time soon after completing the previous one. To those who are wondering, I would like to say that I cannot make the same promise 😊.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Why Large Distributed Systems? . . . . .	1
1.2 How is the Web Different from Other Distributed Systems? . . . . .	4
1.3 Scaling Distributed Systems . . . . .	5
1.4 The Thesis . . . . .	8
1.5 Roadmap . . . . .	11
<b>2 Related Work</b>	<b>14</b>
2.1 Cache Consistency . . . . .	14
2.1.1 Volumes . . . . .	16
2.1.2 Validation-Based Approaches . . . . .	18
2.1.3 Invalidation-Based Approaches . . . . .	23
2.1.4 Approaches Combining Validations and Invalidations . . . . .	28
2.1.5 Basis Token Consistency . . . . .	31
2.2 Caching of Frequently Changing Objects . . . . .	32
2.2.1 Delta Encoding . . . . .	33
2.2.2 <b>HTML Pre-Processing</b> . . . . .	35
2.2.3 Proxy Enhancement . . . . .	36
2.2.4 Data Update Propagation . . . . .	38
2.3 Summary . . . . .	39
<b>3 Background Studies</b>	<b>41</b>
3.1 Retrieval Software and Methodology . . . . .	43
3.2 Study 1: Rate of Change and Characteristics of Embedded Images . . . . .	44
3.2.1 Test Sets Based on Popular Sites . . . . .	45
3.2.2 Test Sets Based on User-Requested Resources . . . . .	46
3.2.3 Test Sets Summary Statistics . . . . .	47
3.2.4 Rate of Change . . . . .	48
3.2.5 Characteristics of Embedded Images . . . . .	49
3.2.6 Follow-Up Study . . . . .	51



---

3.3	Study 2: Changes to HTML Resources and Content Reuse . . . . .	53
3.3.1	Methodology . . . . .	53
3.3.2	Test Sets . . . . .	54
3.3.3	Content Reuse . . . . .	55
3.4	Study 3: Using Object Relationships to Eliminate Unnecessary Validations . . . . .	56
3.4.1	Methodology . . . . .	56
3.4.2	Results . . . . .	57
3.5	Summary . . . . .	58
<b>4</b>	<b>Problem Statement and Approach to Deterministic Object Management</b> . . . . .	<b>60</b>
4.1	Motivation . . . . .	60
4.2	Problem Statement . . . . .	62
4.3	Hypothesis . . . . .	62
4.4	Foundation for Our Approach to Deterministic Object Management . . . . .	63
4.5	Object Change Characteristics . . . . .	64
4.5.1	Object Changes . . . . .	64
4.5.2	Definitions of Object Change Characteristics . . . . .	65
4.6	Types of Object Relationships . . . . .	67
4.6.1	Composition . . . . .	67
4.6.2	Temporal . . . . .	68
4.6.3	Common Dependency . . . . .	68
4.6.4	Common Change Characteristic . . . . .	68
4.6.5	Shared Objects . . . . .	68
4.6.6	Access Patterns . . . . .	69
4.6.7	Structural Organization . . . . .	69
4.7	Combining Object Relationships with Object Change Characteristics . . . . .	69
4.8	Using Object Relationships to Ensure Strong Cache Consistency . . . . .	75
4.8.1	Retrieval Order . . . . .	76
4.8.2	Selecting a Manager Object . . . . .	77
4.8.3	Content Control Commands and Object Management . . . . .	78
4.8.4	Shared Objects . . . . .	80
4.9	Summary . . . . .	82
<b>5</b>	<b>Prototype Design and Implementation</b> . . . . .	<b>83</b>
5.1	Content and Its Organization . . . . .	84
5.1.1	Database . . . . .	85
5.2	MONARCH Content Management System . . . . .	87
5.2.1	Object Processing . . . . .	88
5.2.2	Volumes . . . . .	89
5.2.3	Web Object Cache Compiler . . . . .	92
5.3	MONARCH Web Server . . . . .	95
5.3.1	Volume Invalidation . . . . .	97
5.4	MONARCH Proxy Server . . . . .	100
5.4.1	Request Handling . . . . .	100

---

5.4.2	Determining the Freshness of Cached Objects . . . . .	101
5.4.3	Satisfying a Precondition . . . . .	101
5.4.4	Object Caching . . . . .	102
5.4.5	Statistics Gathered by the MONARCH Proxy Server . . . . .	103
5.5	Content Assembly . . . . .	105
5.6	Example . . . . .	106
5.6.1	The Page . . . . .	106
5.6.2	Object Change Characteristics . . . . .	108
5.6.3	Compilation . . . . .	109
5.6.4	Page Retrieval . . . . .	109
5.6.5	Second Retrieval of the Page . . . . .	112
5.6.6	Object Invalidation . . . . .	114
5.7	Summary . . . . .	115
<b>6</b>	<b>Evaluation of MONARCH</b> . . . . .	<b>117</b>
6.1	Collection Methodology . . . . .	117
6.1.1	Source Web Sites . . . . .	118
6.1.2	Content to Collect . . . . .	118
6.1.3	Content Collection Methodology . . . . .	120
6.1.4	Content Conversion . . . . .	121
6.2	Performance Evaluation . . . . .	123
6.2.1	Web Object Management Simulator . . . . .	123
6.2.2	Simulation Methodology . . . . .	124
6.2.3	Cache Consistency Policies . . . . .	125
6.2.4	Performance Metrics . . . . .	126
6.3	Results . . . . .	126
6.3.1	Effectiveness of the Current Practice Policy . . . . .	127
6.3.2	Comparison of Cache Consistency Policies . . . . .	128
6.3.3	Server Overhead . . . . .	130
6.3.4	Response Time Implications for Different Policies . . . . .	132
6.4	Content Reuse . . . . .	135
6.5	Summary . . . . .	137
<b>7</b>	<b>Conclusions and Future Work</b> . . . . .	<b>140</b>
7.1	Contributions . . . . .	140
7.2	Lifetime of Contributions . . . . .	143
7.3	Future Work . . . . .	144
7.3.1	Selective Content Assembly . . . . .	145
7.3.2	Assembling Customized Content . . . . .	145
7.3.3	Content Reuse . . . . .	149
7.3.4	Dynamic Change Characteristics . . . . .	149
7.3.5	Deploying MONARCH at an Experimental Site . . . . .	150
7.3.6	Coupling MONARCH with Existing Templating Mechanisms . . . . .	150
7.3.7	Deployment Issues . . . . .	151

---

7.3.8	Enabling End Client Nodes with the MONARCH Capability . . . . .	151
7.3.9	Applying Ideas in MONARCH to non-HTML Content . . . . .	153
7.3.10	Methodology for Active Content Collection . . . . .	154
7.4	Summary . . . . .	154
<b>Bibliography</b>		<b>156</b>

# List of Figures

1.1	Simple Caching Scheme on the World Wide Web . . . . .	7
2.1	Performance Characterization of Validation-Based Object Management Policies	19
2.2	Performance Characterization of Lease-Based Object Management Policies .	25
3.1	Cumulative Distribution of October/November, 1998 Test Set Change Ratio Grouped by Content Type . . . . .	49
3.2	Cumulative Distribution of January, 1999 Test Set Change Ratio Grouped by Content Type . . . . .	50
3.3	Cumulative Distribution of 2002 Test Set Change Ratio Grouped by Content Type . . . . .	52
4.1	Home Page of a Popular News Site . . . . .	62
4.2	Classification of Object Change Characteristics . . . . .	65
4.3	Container Object and a Set of Embedded Objects . . . . .	67
4.4	Tree Representation of a Composition Relationship Between Two Objects .	70
4.5	Possible Combinations of Change Characteristics for Two Related Objects .	71
4.6	Useful Combinations of Object Change Characteristics . . . . .	73
4.7	A Tree Representing the Home Page of a Popular News Portal . . . . .	75
4.8	A Tree Representing Change Characteristics of Objects Composing the Home Page of a Popular News Portal . . . . .	75
5.1	Architecture of the Prototype System . . . . .	84
5.2	Grammar for the CCC Commands . . . . .	95
5.3	Internet Explorer's Rendering of the Counter Data in XML via XSLT Trans- formation . . . . .	105
5.4	Tree Representing the Sample Web Page . . . . .	108
5.5	Tree Representing Change Characteristics of Objects Composing the Sample Web Page . . . . .	108
5.6	Tree Representing the CCC Commands Assigned by WOCC to Objects on the Sample Web Page . . . . .	109
5.7	Sample Page Rendered by Galeon . . . . .	110
5.8	Proxy Statistics after the First Retrieval . . . . .	111
5.9	Proxy Statistics after the Second Retrieval . . . . .	112
5.10	Proxy Statistics after the Third Retrieval . . . . .	115

---

6.1	CDF of the Median Time between Non-Deterministic Object Updates . . .	122
6.2	CNN, 31.4 Requests, 190.8 KB. . . . .	129
6.3	ESPN, 38.7 Requests, 159.5 KB. . . . .	129
6.4	Cisco, 19.2 Requests, 55.4 KB. . . . .	129
6.5	Estimated Response Time Performance for Different Policies for Web Site Pages Using Para-1.0 Results in [51] . . . . .	134

## List of Tables

3.1	Summary Information on October/November, 1998 Test Sets . . . . .	47
3.2	Summary Information on January, 1999 Test Sets . . . . .	48
3.3	Number of Embedded Images and Traversal Links Remaining in an HTML Page Between Successive Retrievals in October/November, 1998 . . . . .	50
3.4	Number of Embedded Images and Traversal Links Remaining in an HTML Page Between Successive Retrievals in January, 1999 . . . . .	51
3.5	Summary Information on May 2002 Test Sets . . . . .	52
3.6	Number of Embedded Objects and Traversal Links Remaining in an HTML Page Between Successive Retrievals in May 2002 . . . . .	53
3.7	Content Reuse for Popular Web Pages in October/November, 1999 . . . . .	55
3.8	Occurrence and Potential Elimination of Validation Checks in October/November, 1999 NLANR Proxy Logs. . . . .	57
3.9	Occurrence and Potential Elimination of Validation Checks in May 2002 NLANR Proxy Logs. . . . .	58
4.1	CCC Commands Used by MONARCH for St and Per Objects and BoA Objects with no Related ND Objects . . . . .	78
4.2	CCC Commands Used by MONARCH for the Manager and Managed Objects . . . . .	79
5.1	Mapping of Change Characteristics to CCCs for St and Per Objects . . . . .	94
5.2	Mapping of Change Characteristics to CCCs for Non-Manager Objects . . . . .	94
5.3	Mapping of Change Characteristics to CCCs for Manager Objects . . . . .	94
6.1	Web Sites Used in Study . . . . .	119
6.2	Dynamics of the Collected Objects . . . . .	123
6.3	Performance of the Current Practice Policy (* indicates stale content served in at least one simulation scenario) . . . . .	128
6.4	Server Overhead . . . . .	130
6.5	Requests and Bytes Served by the Server under the Optimal Policy due to Retrieval of New and Changed Objects . . . . .	136

# Chapter 1

## Introduction

The field of computer science is incredibly diverse, even though it is young compared to such fundamental sciences as mathematics and physics, and there is a myriad of exciting directions to explore. In this chapter, we discuss why the direction chosen for this dissertation is interesting and important and provide an introduction to the rest of the dissertation. We begin by discussing some of the early distributed systems and then discuss the Web, one of the largest known distributed systems. We use the Web in this dissertation as a motivation and a testbed.

### 1.1 Why Large Distributed Systems?

Ever since the first computer was invented, attempts were made to make it possible for stand-alone machines to exchange information and cooperate. Networked computers provide a more powerful environment than individual workstations. Over the years, various Local Area Network (LAN) and Wide Area Network (WAN) technologies have been developed and deployed to provide an infrastructure for higher level services. A number of such services or distributed systems were developed at research organizations and corporations to allow information exchange and collaboration within and outside organizations. Some of these systems are well-known and used today, others served as testbeds for early ideas and made

invaluable contributions to the field of distributed systems and computer science in general. Distributed systems erase physical boundaries, shrink distances, and compress time.

Distributed systems vary in their purpose, design, and scale, among other characteristics. For example, the Network File System (NFS) [79], developed by Sun Microsystems, was designed to allow transparent access to files residing on numerous servers on the same LAN. Multiple users can access and modify files as if they had dedicated access to each file. The Andrew File System (AFS or Andrew) [39], developed at Carnegie Mellon University, was an attempt to build a distributed file system that is much more scalable than NFS. Systems like Alex [14], Archie [30], Gopher [74] and Wide Area Information Servers (WAIS) [43, 58] were developed to provide indexing of, and easy read-only access to, information located on remote sites.

All distributed systems have certain issues in common, such as performance, scalability, and fault-tolerance. Larger size is commonly known to exacerbate these issues. A system consisting of two cooperating nodes and supporting five users has different scalability and performance concerns than a system with thousands of nodes and tens of thousands of users. Algorithms and data structures applicable in the former case might not work in the latter case. Quoting developers of AFS [39]: “large scale affects a distributed system in two ways: it degrades performance, and it complicates administration and day-to-day operation.” AFS itself was built with scalability in mind. Authors of [39], trying to emphasize the large expected scale of the system, gave a size of 5,000 to 10,000 nodes. This certainly is not small for a local area network, especially in 1988, when the paper was written. The situation has changed since the Internet and the World Wide Web (WWW or the Web), became so widespread.

Who could have imagined that a research project to facilitate data exchange between scientists would become one of the most talked about phenomena of our time? The Internet, and especially its Web component, has rapidly grown from just a few experimental nodes to one of the largest distributed systems in the world. It took only four years for the



Web to attract 50 million users. In comparison, radio, personal computer and television required 38, 16, and 13 years respectively [86]. The Web has made a huge impact on the way people live, work, collaborate, and learn. We have almost doubled our vocabulary by prepending words with “i” or “e-” or appending them with “.com”. As a Microsoft ad puts it “e-business, e-people, e-economy, e-planet, e-etc...” [69]. Often it is more convenient to find and buy goods on-line than in stores; close to nine million households in the US alone were expected to do holiday shopping on the Web [85]. Even routine daily chores, such as grocery shopping and paying bills, can be performed on-line [82, 88, 8].

This is only the tip of the iceberg. The Web is clearly a fertile soil for developing new applications. Numerous Internet startups, fueled by venture capital, as well as older and well-established companies, are poised to wire everything and make the Internet and the Web truly ubiquitous. From bathrooms and refrigerators to door locks and lawn sprinklers, every conceivable device will be “talking” to other devices on the Web [53]. This type of scale is orders of magnitude larger than that of any existing distributed system. The 32-bit address space of the Internet Protocol (IP) version 4 is rapidly becoming scarce and efforts are underway to introduce IP version 6 with 128-bit addressing. There is a joke that 128 bits would be enough to assign a unique IP address to every insect on Earth. Yes, the anticipated scale is *that* large.

Being such an enormous distributed system, the Web not only creates new opportunities, but also poses an abundance of interesting and unique research challenges. Even researchers from fields other than distributed systems and networks apply their ideas to this new problem space. We also consider this a great opportunity to contribute to the community.

## 1.2 How is the Web Different from Other Distributed Systems?

Sheer size, exponential growth, and client access patterns [36] make the Web fundamentally more complex than other distributed systems. However, there are other aspects distinguishing the Web from other distributed systems.

Early works on distributed systems talk about *files* that change rarely and are stored on disks. For example, systems such as Alex [14] and Archie [30] provide access to mostly static data, such as software distributions and source code archives, stored on FTP sites. As Cate, the designer of Alex [14], pointed out: “. . . data stored on FTP sites . . . change(s) much less often than do normal files.” Cate also noted that users of such systems can often tolerate out of date, or stale, data. On the early Web most pages were also relatively static and stored in files.

The Web has evolved significantly since its inception. It offers access to *objects* or *resources* distributed throughout the network and either stored in files or computed upon request. Unlike early Internet discovery systems [74], the Web supports not only hypertext but also multiple media formats, including raster and vector graphics, audio, and video. As the Web became more popular, Web pages became more sophisticated, rich in content and presentation, and updated frequently. Today, many Web pages are no longer *files* that servers simply store to and read from disks. Servers invoke programs or scripts, typically via Common Gateway Interface (CGI) or one of the many templating mechanisms, such as PHP [75] and Mason [59], and construct pages from multiple components, often with personalization features. Servers may store constituent page components in files or databases, or compute them upon client requests. The Web stopped being simply a medium for exchange of scientific information. It turned into a new and powerful medium to do business, attract customers, advertise, and sell goods and services. Individuals and businesses rely on the Web for everyday tasks and expect timely delivery of up-to-date information.

Since the inception of the Web, researchers have concerned themselves with its scalability and performance problems. How would a system scale to thousands or millions of nodes? How would Web and non-Web traffic affect each other on the Internet? How would the end-user response time be affected? Naturally, lessons and various techniques from previous distributed systems research were brought over and adapted to the new domain of the WWW.

### 1.3 Scaling Distributed Systems

One widely used technique to improve scalability and performance of computing systems is *caching*. Caching is a technique of keeping copies of data closer to the consumer(s) of that data. Caching is usually done on behalf of the requester, and the source of the data has little control over how the data is stored in the cache or managed by the cache, though it tries to influence it.

Many components of stand-alone systems use caching to improve performance: processors have built-in caches, hard drive controllers have internal buffers to cache data, and operating systems have the ability to cache data and information describing the data. Information describing data is called *metadata* or *meta information*. In distributed systems, caching plays an even more important role because, by their nature, distributed systems require data transfers over longer physical distances than in stand-alone systems. Caching saves network bandwidth, decreases end-user latency, and provides a degree of fault-tolerance and better scalability. However, in distributed systems caching must deal with additional complexities of communication, such as network and host failures.

Many well-known distributed systems use caching. NFS caches file data, results of directory lookups, and entire directories and file system information. AFS caches entire files, contents of directories, and symbolic links on every workstation, and maintains two separate caches: one for data, in main memory, the other for status information, on disks [39]. According to Howard et al. [39], caching is the key to Andrew's ability to scale well. Alex [14]

caches different types of information locally: copies of remote files and remote directory information, open FTP connections to avoid the cost of setting up new connections, failures of certain type, such as Domain Name System (DNS) lookup failures, etc. DNS is a core component of the Internet, caching name-to-address resolutions at multiple levels within its hierarchy [61, 62].

An alternative to caching, deployed primarily in large distributed systems, is *replication*, where origin servers copy data to multiple locations, or *mirror* it, and clients can either explicitly pick one of the mirror sites or be directed there transparently by the underlying network infrastructure [47]. The owner of the data has complete control over the replication process: when replication occurs, to which locations, when, and how updates are propagated, etc. Replication is typically used to bring rarely changing information closer to users located far away geographically from the source. For example, distributions of popular software packages, such as the Mozilla Web browser [68], are mirrored on multiple FTP or Web servers around the world.

Both caching and replication have been deployed on the Web to deal with its exponential growth. In his talk titled “How to kill the Internet” Van Jacobson said: “With 25 years of Internet experience, we have learned exactly one way to deal with exponential growth: **Caching**” [42]. On the Web, caching can take place at multiple locations. Web client software, such as Web browsers, cache pages and objects embedded in them in main memory and on disks on the end-user workstations. Internet Service Providers (ISPs) and organizations can install *proxy servers* [57, 34], also called *forward proxy servers*—programs that access remote sites on behalf of a set of users, convert user requests and server responses between various protocols, and optionally cache retrieved objects. Proxy servers that cache objects are called *caching proxy servers*. Caching proxy servers can serve cached objects without contacting remote sites, thus offloading work from remote servers, saving bandwidth on outgoing links, and improving end-user response time. Early installations of proxy servers ran on machines, called *firewalls*, separating an organization’s secure internal

network from the rest of the Internet, as shown in Figure 1.1. Users on the internal network access various external services via the Hypertext Transfer Protocol (HTTP) [6, 31], and the proxy server converts between HTTP and other protocols and caches data. The shading on the picture represents small amounts of Web objects cached on the user’s workstations, and larger amounts cached on the proxy.

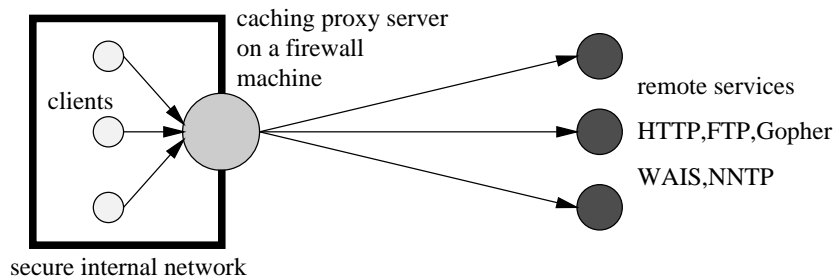


Figure 1.1: Simple Caching Scheme on the World Wide Web

Proxy servers serving multiple ISPs and organizations can access the rest of the Internet via a higher-level proxy server, to further improve scalability. Such a hierarchy can grow larger, and up to four levels have been proposed [78], not counting the browser cache: bottom, institutional, regional, and national levels. When a request cannot be satisfied by lower-level caches, it is forwarded to a higher-level cache. While forward proxy servers cache data on behalf of a set of clients, Web sites can deploy *reverse proxy servers*—programs (often hardware appliances, combining hardware and software) that reduce load on Web servers by serving part of the site’s content. Reverse proxy servers typically cache and serve rarely changing objects, such as images. In addition to deploying reverse proxy servers, many large Web sites, in an attempt to further reduce end-user latency and the load on their Web servers, voluntarily replicate static objects to servers located at ISPs, at points where users connect to the Internet. A collection of such servers is called a *Content Distribution Network (CDN)*. One of the largest CDNs is operated by Akamai [2].

Forward proxy servers, reverse proxy servers, and CDN servers are examples of Web *intermediaries*—programs and devices that handle end-user requests and server responses. These intermediaries, as well as Web browsers and servers, deploy many open and pro-

proprietary protocols, including DNS [11], Cache Digests [55, 37], CARP [84], HTTP [31], ICP [92, 91, 90], PAC [71], WPAD [33], and WCCP [19]. The diversity of various Web caching products, redirection devices, content distribution technologies, and the protocols they use, introduces new variables and complicates the Web infrastructure and the way objects are *managed* on the Web. It is evident that the current WWW caching and replication landscape is no longer the way it looked in 1994. Object management on the Web remains an important research topic. New technologies and protocols, such as WCIP [54] and WCDP [83], are being developed. Existing problems with caching and replication often get worse, and new ones surface.

We presented a brief overview of the Web caching and replication infrastructure and the complexities involved. Precise and complete definitions for the terms used in this section, as well as the description of various components and protocols related to Web caching and replication, are given in the Internet Web Replication and Caching Taxonomy (WREC) [23]—Internet Draft of the WREC working group of the IETF. Krishnamurthy and Rexford [47] provide an in-depth description of the history and evolution of the Web and various Web components, protocols, and technologies. Rabinovich and Spatscheck [77] provide a good treatment of Web caching and replication, as well as Web protocols.

## 1.4 The Thesis

We have discussed the effect that size has on the scalability of distributed systems and provided an overview of approaches addressing the scalability of distributed systems and the Web. To set the stage for the upcoming chapters, we now provide a brief description of the problem this dissertation addresses and outline our approach to that problem.

Managing distributed objects would be easy if they never changed. However, objects do change, often frequently and unpredictably, and require a mechanism to ensure that multiple copies of each object distributed throughout the network are synchronized with the master copy of that object. Mechanisms under which neither servers nor caches rely on heuristics

and it is *a priori* known how each object will be managed are called *deterministic*. Currently adopted time-based cache consistency mechanisms are heuristic in nature and thus result in *non-deterministic* management of objects. Server-driven invalidation mechanisms for strong cache consistency have been proposed, but they require servers to maintain per-client state and thus do not scale well. Server-driven invalidation has other issues as well, such as propagating invalidations to unreachable clients and delaying object updates.

While much research has already been done on managing distributed objects, we hypothesize that it is possible to devise a mechanism for deterministic object management that provides strong cache consistency and is more efficient than currently deployed and proposed mechanisms. In this dissertation, we investigate existing and proposed object management mechanisms and characteristics of distributed objects in order to better understand the issues involved and to devise a mechanism addressing these issues.

We propose a combination of techniques to manage objects deterministically in a large distributed system and use the Web as a motivation and a testbed for our work. First of all, we recognize that many popular pages are composed from a number of heterogeneous objects where the boundaries between these objects are not made visible to the outside world, resulting in monolithic pages. We propose to exploit the possibilities resulting from preserving object identities within pages. Second, we classify objects based on their change characteristics and deploy different management mechanisms based on the category to which an object belongs. Third, we recognize that objects are not stand-alone entities: they are tied together by relationships, such as a containment relationship or dependence on the same underlying database. Knowledge about these relationships can be effectively used to support deterministic management of objects. And last, we bring in the known technique of piggybacking as an effective means of communicating information between clients and servers. These techniques lead to deterministic object management and improved performance for clients and servers.

The deterministic object management with strong cache consistency offered by our ap-

proach allows content providers to make more of their content cacheable. Furthermore, our approach enables content providers to expose internal structure of their pages to clients.

To evaluate our approach and compare it to the existing approaches we use a novel methodology that allows us to study a wide range of cache consistency policies over a range of access patterns. The essence of our methodology is to actively gather snapshots of selected content from sites of interest and then use that content as input to a simulator. Traditional methodologies rely on server and proxy logs which do not contain the complete request stream to a particular site and provide no indication of when resources change. Logs from popular server sites are not generally available to the research community. Neither server nor proxy logs contain HTTP cache directives, making it impossible to evaluate the effectiveness of the cache consistency policy reflecting current practice. Our methodology is a step towards filling these gaps and obtaining data for any site of interest that is not otherwise available for study.

The results of our evaluation show that our approach provides strong cache consistency for all objects, even for unpredictably changing ones, and incurs smaller byte and message overhead than heuristic policies. The results also show that as the request arrival rate or the number of clients increases, the amount of server state maintained by our approach remains the same while the amount of server state incurred by server invalidation mechanisms grows.

The deterministic object management mechanism we designed and the novel methodology for evaluating cache consistency policies we developed are two important contributions made by this dissertation. Our evaluation methodology will remain useful as sites are unlikely to start providing server logs and records of object modification events to researchers. The issue of maintaining cache consistency in distributed systems is important today and will remain important, especially for systems that span large distances and attract large number of clients. While not all applications require strong consistency, some, such as Web-based business-to-business and business-to-consumer applications, on-line games, and distributed simulations depend on it. Our approach to cache consistency is applicable to



any set of related objects in a distributed system, not just to the Web, and can be used to improve many distributed applications.

## 1.5 Roadmap

The rest of the dissertation consists of six chapters, as follows:

- **Related Work:** Chapter 2 presents prior work that is most relevant to the focus of the dissertation: cache consistency issues and efficiency of object management. We summarize techniques for grouping objects into “volumes”, as some of the cache consistency approaches that we discuss use volumes. We discuss validation-based approaches to cache consistency and the tradeoff between consistency guarantees and amount of validation traffic. We discuss invalidation-based cache consistency approaches and the tradeoff between server state and validation requests. We also discuss combinations of validation- and invalidation-based approaches and techniques that use existing traffic between servers and caches to exchange invalidation information.
- **Background Studies:** Chapter 3 describes a series of background studies that we performed over the course of four years while working on this dissertation. We carried out these studies to investigate aspects of the Web that this dissertation builds on and to evaluate the potential of the techniques that we outlined above. We repeated some of the earlier work on more recent data, to see whether the earlier findings are still true, and also investigated aspects of the Web not studied in previous work. As part of these studies, we developed and used a novel methodology for studying Web resources and understanding how they change. We studied the following issues: frequency with which Web resources change and how images and other embedded resources change relative to their container HTML resources; whether relationships between embedded and container objects are stable over time; the nature (predictability, locality, and extent) of changes to HTML pages; potential for elimination of validation requests if

relationships between objects are exploited and potential for content reuse if pages are constructed from components.

- **Problem Statement and Approach to Deterministic Object Management:** Based on the tradeoffs of the existing approaches to cache consistency and improved understanding of Web resources described in the previous two chapters, in Chapter 4 we define the problem this dissertation addresses and present our approach to that problem. We discuss how objects change and present our classification of object change characteristics. We then discuss various types of relationships between objects and how these relationships can assist in object management. We present our approach, called MONARCH (Management of Objects in a Network using Assembly, Relationships and Change Characteristics), which combines object relationships with object change characteristics to ensure strong cache consistency.
- **Prototype Design and Implementation:** Chapter 5 presents implementation details of the MONARCH prototype system. We describe content organization at a Web site and design and implementation of the MONARCH Content Management System. We also describe MONARCH Web and Proxy Servers, followed by the description of content assembly mechanism present in both servers. We then present a detailed example of a real Web page and show how that page is handled by the prototype system.
- **Evaluation of MONARCH:** Chapter 6 presents the evaluation of MONARCH. We first describe a novel methodology that we developed for evaluating a wide range of cache consistency policies over a range of access patterns. We then use that methodology to evaluate the performance of MONARCH and compare it to the performance of existing and proposed cache consistency policies, including a policy modeling the behavior of modern caching proxy servers. We also use results from a prior study done by others to estimate the impact of various cache consistency policies on user-perceived

response time.

- **Conclusions and Future Work:** Chapter 7 summarizes the major contributions of this dissertation and presents ideas for future work. Our contributions include: methodologies devised as part of background studies and MONARCH evaluation; findings of the background studies; our taxonomy of object change characteristics; the important combinations of object relationships with object change characteristics that we identified; the MONARCH approach to strong cache consistency, which is more efficient than existing approaches; and the prototype system implementing MONARCH. We intend to investigate various extensions to the basic content assembly mechanism, such as selective assembly and assembly of customized content. We also plan to look into various deployment issues, ways of combining MONARCH with the existing templating mechanisms for building dynamic Web sites, and also ways to enable Web clients to support MONARCH. Another big area of future research is applying ideas of this dissertation to non-HTML objects, such as objects present in distributed computer games.

The following seven chapters comprehensively summarize existing research in cache consistency, caching of frequently changing objects, and our contributions to these fields.

## Chapter 2

# Related Work

There has been a significant amount of research, both in academia and industry, related to various aspects of the Web and Web caching in particular. A number of scientific conferences annually publish papers on Web caching and a number of companies develop and market Web caching and content distribution products and services. Instead of covering the entire body of work on various aspects of caching, we concentrate on research that is most relevant to this dissertation, namely cache consistency and caching of frequently changing objects. A good, but not exhaustive, and by now dated, survey of these and other aspects of Web caching was produced by Wang [87]. A much more extensive, in-depth treatment of various Web aspects, including history of the Web and Web protocols can be found in a book by Krishnamurthy and Rexford [47]. A good treatment of cache consistency issues and approaches to caching frequently changing pages can also be found in a book by Rabinovich and Spatscheck [77].

### 2.1 Cache Consistency

As objects are requested by clients or distributed by servers, copies of objects are stored at locations other than where they originated. Cached copies are called *fresh* as long as they remain identical to their respective master copies at the origin servers. When objects are

updated at their origin location, their copies distributed throughout the network become *stale* and need to be updated. The problem of synchronizing object copies distributed in the network with the master copy at the origin server is called the *cache consistency* problem. Failure to perform such synchronization, resulting in clients receiving stale object copies, is called *consistency failure*.

It should be noted that two different terms have been used in the research literature to refer to this problem. A number of papers use the term *consistency* [56, 98, 100, 101, 99, 28, 72, 83, 76, 25], and a number of other papers use the term *coherency* [48, 49, 9, 10, 7]. In this dissertation we use the term *consistency*, except in the context of describing other work.

The reason two terms are used in the research literature is to differentiate between two separate problems: 1) the problem of synchronizing cached copies of an object with the master copy and 2) the problem of ensuring mutual consistency within a set of cached objects. Bradley and Bestavros referred to the former problem as the problem of maintaining *coherency*, and called the latter problem the problem of maintaining *consistency* [9, 10]. At least one paper uses the term *consistency* to refer to *both* problems [83].

Approaches to cache consistency differ in the consistency guarantees they provide and in the amount of overhead they incur. Approaches that guarantee clients would obtain the same object from the origin server as they received from the cache are said to provide *strong* cache consistency. Approaches that allow caches to serve out-of-date objects to clients are said to provide *weak* consistency. The overhead is measured in the number of requests and bytes exchanged by caches and servers and by the amount of state that caches and servers maintain.

All approaches to cache consistency can be classified based on how caches learn about object updates. One set of approaches is known as *validation-based* because these approaches require caches to periodically validate cached copies of objects against master copies at origin servers. These approaches are also known as *client polling* and *pull* approaches.

Another set of approaches are called *invalidation-based* approaches because they require servers to notify caches when objects change at the servers. These notifications are called *invalidations*. The terms *callback* and *push* are also used to describe server invalidation.

Approaches that combine client polling and server invalidation have also been proposed. Caches and servers can use existing traffic between them to exchange additional information that is not otherwise part of the request and response messages. Adding additional information to existing messages is called *piggybacking*. Caches can piggyback validation requests for multiple objects onto messages they send to servers, and servers can piggyback invalidations for multiple objects onto their responses to caches. Servers can also group a set of (related) objects together and refer to the entire set as a *volume*. For example, a server can invalidate all objects in a volume with a single message.

### 2.1.1 Volumes

A number of cache consistency approaches discussed in this chapter group objects at a site into volumes. There are different ways to construct volumes. Krishnamurthy and Wills, in their work on *piggyback server invalidation* (PSI) [49], proposed grouping all resources at a site into a single volume and proposed grouping resources into volumes based on the first level prefix of the path name. The first possibility might result in a large number of piggybacked invalidations in cases where many resources at a site change often. The second option provides smaller volumes, but it also groups many unrelated resources together.

Cohen et al. [21] further explored volume construction at a server and introduced four metrics used for evaluation of volume construction techniques:

1. *Recall*—fraction of client requests to the proxy that benefit from server volume received by the proxy within the last  $T$  seconds.
2. *Precision*—fraction of hints in a server volume that accurately predict a client request arriving within the next  $T$  seconds.

3. *Update Fraction*—fraction of client requests to the proxy accessing a resource that was already requested from that proxy before, subsequently predicted by a server volume and updated by the proxy.
4. *Hint Size*—size of the hints that the server sends to the proxy. The smaller the hint size—the better.

These metrics represent competing trade-offs. Cohen et al. showed that the problem of optimizing them is NP-complete [22] and proposed and evaluated a number of heuristics and algorithms for volume construction [21, 22]. Krishnamurthy and Rexford summarize these studies and findings in their recent book [47]. In their studies, Cohen et al. considered grouping resources with the same directory prefix in the URLs, up to some number of levels (0, 1, 2), into a single volume. They used a heuristic that resources in the directory are likely to have related content or occur as embedded hyperlinks in related Web pages. The second technique investigated in [21] is probabilistic grouping of related resources into volumes. The idea is for the server to observe a stream of requests and estimate the pairwise dependences between resources thus predicting which ones are likely to be accessed together. These related resources are grouped into a single volume. Directory-based volumes can get excessively large and carry irrelevant information. Probabilistic volumes can be quite accurate but require additional computational overhead at the server. As an aid in reducing computational complexity the probabilistic volume construction technique can perform pairwise comparisons for resources with the same directory prefix. Cohen et al. [21] investigated further reduction in volume sizes, called *thinning*, by applying proxy filters, supplied by proxy servers. Proxy filters instruct origin servers to *not* piggyback certain, perhaps recently seen, volume elements, effectively thinning volumes. Focusing on most popular resources at a site and eliminating dependencies between pairs of resources that rarely occur together was investigated in [22]. The same work also studied greedy algorithms and suggested sampling logs prior to volume construction to reduce computational complexity. Performance evaluation of the proposed algorithms showed that all of them

can achieve high recall and high update fraction. Directory-based volumes, however, offer low precision and result in large hint sizes. On the other hand, a two-pass algorithm that removes ineffective hints was able to offer 60-80% recall and 80-88% precision for time intervals from one to five minutes, while keeping hint size to 2-10 hints per message.

### 2.1.2 Validation-Based Approaches

The HTTP protocol [31] provides two validators that caches can use to validate cached objects with the server. One validator is the timestamp indicating when the object was last modified. Servers use the `Last-Modified` response header to provide caches with such a validator. The other validator is opaque and is called an entity tag. Servers use the `ETag` response header to associate an entity tag with an object. To validate a cached object whose `Last-Modified` value is known, a cache sends an HTTP `GET` request to the server and supplies the `Last-Modified` value via the `If-Modified-Since` request header. Such requests are known as *conditional*. The server replies with the HTTP response code 200 `OK` and the new version of the object if the object was updated. If the object is up to date, the server replies with the HTTP response code 304 `Not Modified`.

Validation-based approaches differ in how they determine the length of time, called *Time-To-Live (TTL)*, during which they treat cached objects as fresh, before validating them with the server. These approaches use different TTL values to balance the tradeoff between consistency guarantees and the number of validation requests, as shown in Figure 2.1. Shorter TTLs reduce consistency failures but result in more validation requests, some of which are unnecessary, placing additional load on the network and origin servers. Studies by Krishnamurthy and Wills [49], Nahum [70], and Arlitt and Jin [4] showed that 15-18%, 30%, and 37% respectively of all requests received by servers resulted in responses with the HTTP response code 304 `Not Modified`, indicating to caches that cached object copies are up-to-date.

Validating objects reactively, while clients are waiting for them, results in higher client-



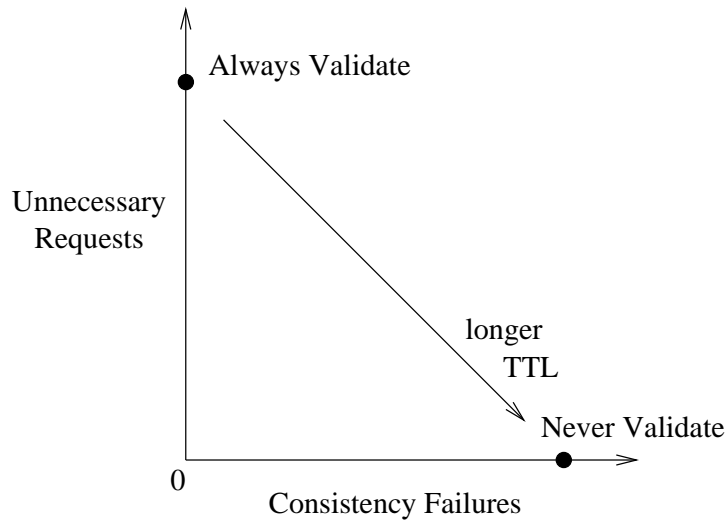


Figure 2.1: Performance Characterization of Validation-Based Object Management Policies

perceived latency than if validations are performed proactively, ahead of client requests, negating the latency reduction offered by caching in the first place. Dilley showed that revalidations increase the average latency by 1.0–5.7 times and the median latency by 5.5–9.2 times for an individual object [24]. Krishnamurthy and Wills found that client-perceived per-page latency increases due to cache validations by 2.6–5.1 times depending on the network distance between the cache and the server and on the HTTP protocol option used [50]. Krishnamurthy et al. [51] also reported that reducing the quality of embedded objects does not significantly improve client-perceived latency, suggesting that revalidation of embedded objects, particularly smaller ones, is not significantly better latency-wise than fetching these objects anew.

We now discuss specific validation-based approaches, starting with the approaches that validate cached objects only upon client requests.

### Always Validate, Never Validate, and Fixed Time-To-Live

Validating cached objects upon each client request is called the *polling every time* or *Always Validate* approach. This approach provides strong cache consistency and offers byte savings,

especially for large objects that rarely change, but makes clients wait for objects to be validated and places the same load on servers request-wise as without caching. If caches cannot validate cached objects due to network or server failure, they can either respond with an error message or a warning that the returned data is potentially stale. Caches can also cache objects once and never validate them, minimizing the network traffic and load on the origin servers, but providing weak consistency. This approach is called *Never Validate*.

Modern Web browsers, such as Netscape, can be configured to validate locally cached Web objects using either of the two extreme approaches. One could view Always Validate and Never Validate as approaches that assign cached objects Time-To-Live (TTL) values of zero and infinity respectively and validate objects after TTLs expire. Caches can also pick a value for the TTL from the range between the two extremes, reducing the number of validations as compared to Always Validate and providing stronger consistency than Never Validate. Such a middle ground approach is called *fixed* TTL.

### Server-Assigned Expiration

Web servers can explicitly assign TTL values to objects whose update patterns are well-known. The HTTP protocol supports TTLs via `Expires` and `Cache-Control: max-age` directives [31]. In practice, many objects change unpredictably making the assignment of accurate TTLs difficult. An object might remain unchanged for a long period of time, and then undergo multiple changes within a short period of time. Mogul found that the majority of objects do not carry server-assigned expiration times [63]. Our own observations support these findings [95]. We also found that even when servers do provide expiration times, the values they set may be short or negative (in the past) for objects that do not change [95]. Content providers often misuse this header, either deliberately or due to a lack of understanding. Instead of instructing caches for how long objects remain fresh, they tell caches not to cache their objects.

### Adaptive TTL

Servers may not know when the next update to an object will take place, but they do know when the previous update occurred and can supply that information to caches via `Last-Modified` response header. Cate [14] observed that file lifetime distribution is bimodal, i.e. younger files are likely to be modified sooner than older files, and should be considered fresh for a shorter period of time. This heuristic is the basis for the *heuristic* or *adaptive* TTL approach which takes a configurable percentage of object's age as the TTL value for that object. The approach is also referred to as the Alex protocol because it was first used in the Alex file system [14].

Caches compute the age of the object by subtracting the value of the server-supplied `Last-Modified` header from the time of the object's placement in the cache. Cache administrators can configure caches to apply different percentage values, such as 5% or 10%, to the age of cached objects, balancing object staleness and amount of traffic to servers. Most, if not all, modern caches, such as Squid [89], support the adaptive TTL mechanism.

Due to its heuristic nature, the adaptive TTL mechanism provides weak cache consistency and generates many unnecessary validation requests to servers—up to a third of all requests received by servers as was shown earlier—for objects that have not been updated. Its performance is characterized by the intermediate points in Figure 2.1, between the two extreme policies. In an attempt to retain control over their content, servers often mark objects, even infrequently changing ones, as uncacheable to preclude caches from using the heuristic TTL mechanism on these objects.

Many Web sites today build their pages upon client requests, using Common Gateway Interface or numerous templating mechanisms, such as PHP [75] and Mason [59]. Unless application programmers specifically instrument their code to compute last modification time for such pages and add it to server responses, the server cannot determine when the page was last modified and omits the `Last-Modified` header from its responses. Research studies confirm that many server responses do not carry `Last-Modified` information and

also show that when present such information may be incorrect. For example, studies by Douglis et al. [26], Krishnamurthy and Wills [49], and Kroeger et al. [52] found that only 50-80% of server responses contain `Last-Modified` headers. Our own study [95] found that `Last-Modified` information is generally available (in 82% to 86% of cases, considering HTML pages and images embedded in them) and generally corresponds to whether the resource changed or not. However, we found instances where the resource does not change, but the value of the `Last-Modified` header does: 1.53% and 9.36% for different data sets. Even more problematic are a relatively few instances (0.32% and 0.03%) where the resource has changed but the value of the `Last-Modified` header did not. We obtained similar results in another study [94], although in a test set containing just HTML objects only 35% of the resources had `Last-Modified` information. The latter suggests that images are more likely to have `Last-Modified` information than HTML resources because servers store most images in files on disk instead of creating them upon client requests. While pages without `Last-Modified` information can still be cached, caches have no way to validate them with the server and usually treat such pages as uncacheable.

### Proactive Validation

Caches can proactively validate selected cached objects that have already expired or are about to expire, extend the lifetime of those objects that are still fresh, and evict or prefetch stale ones. This technique improves cache consistency and client latency. At least one vendor of Web caching appliances used this *Active Asynchronous Refresh* technology [12].

Cohen and Kaplan [20] studied various proactive cache *refreshment* policies and applied traditional cache replacement algorithms, such as Least Recently Used and Least Frequently Used, to decide which objects to refresh. They showed that about half of all freshness misses, i.e. validation requests induced by clients and resulting in the HTTP response code 304 `Not Modified`, could be eliminated at the expense of two added proactive validation requests per eliminated freshness miss.

### 2.1.3 Invalidation-Based Approaches

The fact that many cache validation requests turn out to be unnecessary suggests that many objects remain unchanged for long periods of time. Servers can accept a more active role in object management and explicitly notify caches of object updates. Caches treat all cached objects as fresh until they receive server invalidation, thereby eliminating all validation requests.

Server invalidation (callbacks) was used by Howard et al. in AFS [39] as a replacement for client polling. Liu and Cao [56] compared server invalidation to adaptive TTL and polling every time approaches using simulations driven by Web server logs. They showed that server invalidation is a better technique for maintaining strong cache consistency than polling every time based on the bandwidth used, except in cases when file lifetimes are short (on the order of minutes). Their results also indicated that invalidation is comparable to the adaptive TTL approach in terms of the amount of generated network traffic, average user response time and server workload, but provides strong rather than weak cache consistency.

In order to provide clients with invalidations, servers must keep track of a potentially large number of clients. The more clients servers keep track of the larger memory requirements are and the more invalidation messages servers need to send out upon object updates. Servers need to decide whether they notify clients of updates to all objects or only to those objects that clients requested previously and may still have in their caches. The former wastes network and client resources while the latter requires servers to maintain more state. If clients have already evicted objects for which they receive server invalidations, the resources are also wasted. Servers also need to decide whether they wait for all clients to acknowledge each invalidation message before proceeding with object updates or not. Waiting for client acknowledgements delays object updates at the server, potentially indefinitely if some clients crash or become unreachable due to a network partition. Proceeding with object updates before all clients responded relaxes consistency guarantees. While server invalidation is meant to provide strong cache consistency, caches may serve

stale objects to clients if they unknowingly become disconnected from the server and cannot receive updates.

Instead of keeping information about all previously seen clients for indefinitely long period of time, servers can expire such information after a short period of time, effectively keeping track of a smaller number of active clients. However, servers and caches must agree on such a TTL value since servers have an obligation to notify clients of object updates and cannot just stop sending invalidations at will.

### Leases

Servers use the notion of *leases* to control how often they expire information about clients. A lease is an agreement between a cache and a server that gives the cache certain rights on the cached objects for an agreed upon period of time, called the *lease length*. Servers stop notifying clients whose leases expired and wait for acknowledgements from unreachable clients only as long as leases are valid. Caches periodically renew expired leases, introducing requests similar to validation requests for cached objects, some of which may be unnecessary, as shown in Figure 2.2. Longer leases require servers to maintain more per-client state, but induce fewer unnecessary lease renewals. In contrast, shorter leases reduce the amount of server state, but incur more lease renewals. An invalidation-based policy with a zero-length lease is equivalent to the Always Validate policy.

Leases were proposed by Gray and Cheriton who studied lease performance and the selection of the appropriate lease length using an analytical model and data from the V distributed system [35]. Liu and Cao suggested the use of leases on the Web as a means to address the scalability problem [56]. Subsequently, Yin et al. introduced *volume leases* to further reduce the cost of server invalidation [98]. They showed that the introduction of volumes reduces traffic at the server by 40% and can reduce peak server load when popular objects are modified. Yin et al. further explored scalability aspects of volume leases and proposed to extend volume leases to cache consistency hierarchies [100]. They also explored

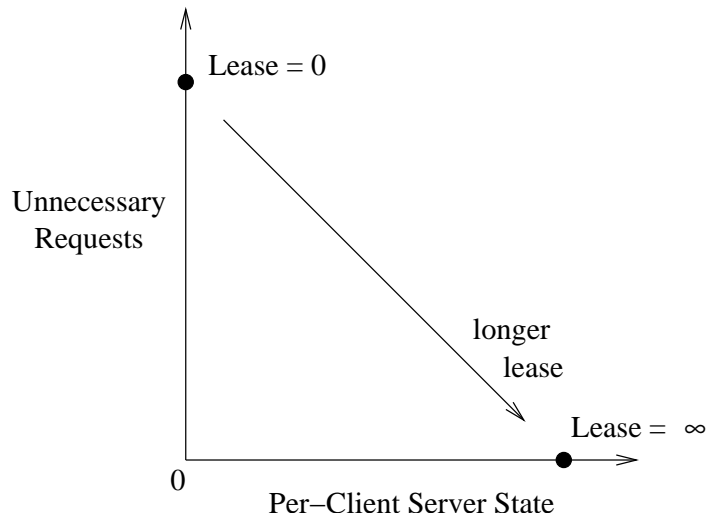


Figure 2.2: Performance Characterization of Lease-Based Object Management Policies

engineering techniques for improving scalability of server-driven invalidation [101, 99].

### Adaptive Leases

A critical parameter controlling the amount of server overhead, the number of server invalidation and client lease renewal messages, is the duration of a lease. The work on adaptive leases by Duvvuri et al. [28] focused on developing analytical models and policies for determining the optimal lease duration under various conditions: when the server state or when the number of control messages is the constraining factor. The server can periodically (at large time scales, on the order of tens of minutes or hours) re-compute lease duration and make adjustments based on the current load. Duvvuri et al. also presented a set of policies that enable the server to re-compute lease duration at smaller time scales, such as on every lease renewal request. These policies are: 1) age-based leases, where the lease duration is a configurable percentage of object's age, which is similar to computing the object's adaptive TTL; 2) leases based on the activity of caches, where the server grants longer leases to more active caches; 3) leases based on the amount of server state, where the server makes lease duration inversely proportional to the amount of its state, thereby decreasing lease

duration under heavy load. Duvvuri et al. [28] performed simulations driven by proxy logs with synthetically generated object updates and observed that object lease of one hour provides 425% improvement in server state overhead over the server invalidation approach and 138% improvement in client validation requests over the client polling approach. The results also showed that for a one hour object lease the server state for the most popular server in one proxy log is 1030 leases and the number of client validation requests is about one message every 33 minutes. These results are for the Digital Equipment Corporation proxy trace containing a little over 1.2 million client accesses. The duration of the trace is about 41 hours.

### Cooperative Leases

Ninan et al. [72] argued that consistency mechanisms, in particular leases, designed for single proxy servers do not scale to large collections of proxy servers under the same administrative control, such as in a CDN, and proposed a generalization of leases called *cooperative leases*. Under current consistency mechanisms, each of the thousands of proxy servers in a CDN network needs to maintain consistency independently of other proxies, burdening origin servers with a high volume of control messages and state overhead. Ninan et al. also argued that standard leases require the server to notify its clients of all updates to objects, thereby providing the same consistency guarantee for all objects, which might be too restrictive for a CDN.

The cooperative leases approach allows servers to grant leases to a *group* of proxy servers, represented by a *leader*, rather than to each proxy server individually. Thus, origin servers maintain state for and send object update notification to each leader of the group rather than to each individual proxy server. Leaders then propagate object update notifications to all group members. In essence, this approach offloads the work of managing objects from the origin server to one of the proxies in a group. Proxies in a CDN are partitioned into non-overlapping regions, and each proxy within a region maintains mapping between cached



objects and the leader responsible for maintaining consistency for that object. Ninan et al. also introduced a rate parameter indicating the rate at which the server agrees to notify clients of object updates, thereby providing a mechanism for varying levels of consistency guarantees.

Ninan et al. used trace-drive simulation with synthetically generated object update events to study how the following policies affect performance: 1) leader selection policies; 2) eager and lazy lease renewal policies; 3) sending object invalidation versus object update to the leader. The work assumed that the lease duration was determined as discussed in the prior work by Duvvuri et al. [28] on adaptive leases. Comparison of cooperative leases with standard leases found that the server overhead decreases, but not as much as expected due to large number of objects being requested by only one proxy, in which case cooperative leases provide no benefit over standard leases. The number of object invalidations sent out by the server does go down, at the expense of consistency maintenance state and traffic between proxies in a group.

### **Web Cache Invalidation and Web Content Distribution Protocols**

In an attempt to create an open standard for invalidating objects cached at CDNs and other participating intermediaries, researchers from academia and industry jointly developed proposals for two server invalidation protocols. Li et al. [54] developed the Web Cache Invalidation (WCIP) protocol, the older of the two protocols. Tewari et al. [83] have recently presented their proposal for Web Content Distribution (WCDP) protocol, which is still work in progress at the time of this writing. The two proposals are similar, except WCDP supports multiple levels of consistency: strong, delta, explicit, and mutual, supports updates in addition to invalidations, and is a request/response protocol, unlike WCIP. Both WCIP and WCDP are instantiations of protocol specifications, thereby including specific details of request and response message formats, message encodings, etc. For example, WCDP encodes messages using eXtensible Markup Language (XML) and sends them via HTTP

POST method.

In WCDP, content providers group objects related by user interests into *content groups* and clients subscribe with servers to receive update notifications for objects in these groups. Servers notify all subscribed clients of object updates and wait for acknowledgments from all clients before making the updated version of the object publicly available. Servers can invalidate individual objects or *objects groups* (which appear to be similar in spirit to volumes). Objects can belong to multiple object groups.

#### 2.1.4 Approaches Combining Validations and Invalidations

Instead of using either client polling or server invalidation, one could use both. Two main proposals for such a combination have been proposed.

##### Adaptive Push-Pull

Bhide et al. [7] argued that client pull and server push approaches to maintaining coherence have complementary properties and limitations. The pull approach is simple to implement and does not require servers to maintain per-client state, but it generates many validation requests to the server and can miss updates to the cached objects. The push approach provides strong coherency, but requires servers to maintain per-client state. They also observed that the frequency of changes to time-varying data (such as stock quotes, the example they used in the paper) itself changes over time, making it difficult to choose between pull and push approaches beforehand. Bhide et al. [7] proposed to combine pull and push approaches to produce the *adaptive Push-Pull* approach.

They assumed that the user specifies the *temporal coherency requirement* (*tcr*) for each cached object. They also noted that the *tcr* can be expressed in units of time or object value. The former means that an object returned from the cache must be at most *tcr* time units older than the version of that object at the origin server. The latter means that the difference between the value of the object returned from the cache and the value of the

same object at the origin server must be at most  $tcr$ . In the adaptive Push-Pull work, the authors considered only the latter form of  $tcr$ . They combine pull and push approaches as follows.

One combination is called *Push and Pull (PaP)*. Clients register with the server, provide the desired  $tcr$  value, and pull updates from the server based on that value. The server monitors object changes, detects those that clients are likely to miss, and pushes these updates to clients. The server must maintain per-client state, but the authors argue that this state is *soft*, meaning that if the server loses that state the mechanism degrades to and performs no worse than pure pull. Also, if a client becomes unreachable, and the server cannot inform that client of an object update, the resulting coherence guarantees are the same as those of the pure pull approach.

Another combination is called *Push or Pull (PoP)*. The server pushes updates to all clients by default, if it has sufficient resources to do so, and switches to the pull approach for certain clients, when resources become constrained. The server can dynamically determine whether to use push or pull on a per-client basis.

Bhide et al. also discussed two additional variations. One variation replaces push clients in PoP with PaP clients, resulting in PaPoP approach. The other variation amends PaPoP approach with leases, whereby the server pushes updates only to clients that hold valid leases. Clients must revert to pure pull or renew expired leases.

### **Piggyback (In)Validation**

Instead of switching between polling and invalidation one could make use of existing traffic between clients and servers to relate invalidation information to caches. Krishnamurthy and Wills proposed the *piggyback cache validation* (PCV) [48] and PSI [49] approaches based on this idea. Both mechanisms attempt to eliminate stale entries from a cache, extend lifetime of unmodified objects, and minimize the number of cache validation requests. The PCV mechanism can easily be implemented within HTTP.

In the PCV approach [48], when a proxy cache has a reason to communicate with a server it piggybacks a list of cached objects from that server, for which the explicit expiration time is unknown and a heuristically assigned TTL has expired. The server replies with the requested object and indicates which objects on the list are now stale. The proxy uses that information to invalidate stale resources and may extend the lifetime of resources not explicitly invalidated by the server.

While the PCV approach takes advantage of the information available only to a proxy cache, the PSI approach [49] is server initiated and takes advantage of the information available to the server, but not to the proxy caches. Servers partition available resources at a site into volumes, and maintain unique identifiers and version information for each volume. When a resource or a set of resources change within a volume, the server updates volume version number and notes which resources changed. Requests from proxy caches normally contain the volume identifier and volume version for the requested resource. If a proxy cache does not have that information, it requests that the server provide it as part of the response. The server response contains the volume identifier, current volume version and a list of resources which have changed since the volume version was provided by the proxy.

Krishnamurthy and Wills studied combinations of PCV with TTL and adaptive TTL mechanisms using simulations driven by proxy logs and showed that PCV can reduce cache validation traffic by up to 16-17% and the staleness ratio by up to 57-65% [48]. They also showed that the average cost, which takes into account the response latency, number, and size of request and validation messages, is reduced by 6-8% [48].

Krishnamurthy and Wills used the same cost metrics (response latency, bandwidth, and number of requests), the same proxy logs as in their PCV study [48], and a number of additional server logs and showed that the PSI technique provides close to strong cache consistency and reduces the amount of proxy-server traffic as compared to TTL-based approaches [49]. They also showed that PSI sometimes performs better and sometimes per-

forms worse than PCV, and that the two techniques should be combined to obtain the best overall performance. Such a hybrid approach reduces the overall cost by 7-9% and the staleness ratio by 82-86%, as compared to TTL policies [49].

### Dynamic Selection of Consistency Mechanisms

Recent work by Pierre et al. [76] on dynamic selection of optimal distribution strategies for Web documents suggests using different consistency mechanisms for different documents instead of applying the same consistency mechanism to all documents. The authors propose finding such an arrangement of (document, strategy) pairs for all documents and available strategies (such as those discussed above) that achieves globally optimal system performance along specified metrics. They show, using a trace-driven simulation, that when the same consistency strategy is applied to all documents, different metrics are optimized under different policies, and no policy provides the best performance along all metrics. Under their proposed scheme, all the servers maintaining document replicas periodically send recent portions of their logs to the main server. The main server then combines these logs with its own log and runs simulations on this newly acquired data to reevaluate whether consistency strategies currently assigned to the documents are still appropriate or need to be changed.

#### 2.1.5 Basis Token Consistency

The cache consistency approaches discussed so far are designed to ensure that cached copies of objects are synchronized with their master copies at the origin server. Bradley and Bestavros in their recent work on Basis Token Consistency (BTC) [9, 10] use the term coherence to refer to that type of consistency and focus their work on another type of consistency. In the BTC work, the term consistency describes recency of one cached object relative to a related cached object. The relationship in this context means that the two objects have a common dependency on one or more data sources. Two related objects residing in a cache may differ from their master copies at the origin server, but they are

fresh relative to each other since they both were produced from the same, albeit by now stale, data. A strongly consistent cache always provides a non-decreasing view of the origin server state. The BTC work relies on other techniques, such as those discussed earlier, to maintain coherence.

The idea of BTC is to add the `Cache-Consistent` header to server responses, listing each data source (origin datum or token) used in production of the response along with its version number. BTC-compliant caches index tokens found in server responses and keep track of token version numbers. Upon receiving a response from the server that contains a newer version number for a datum, the cache invalidates all cached entries that depend on that datum. BTC can provide consistency only for objects that have common dependencies on the underlying data. BTC requires servers to maintain dependencies between underlying data and the resulting pages, but does not require servers to maintain any per-client state. BTC also requires client caches to maintain an index of tokens, with the possibility that the index could grow arbitrarily large. BTC servers do not expose page structure to clients, forcing caches to operate at page granularity.

## 2.2 Caching of Frequently Changing Objects

Traditional caching mechanisms work well for objects that change rarely. Many Web sites today construct their Web pages from multiple data sources, add personalization features, or simply change the location of various items on pages to create an illusion of frequent updates to retain users. Such frequently changing pages, and pages generated upon client access, present problems to current caching mechanisms. While it may be impossible to benefit from caching entire pages that change frequently, various approaches have been proposed to assist in caching pieces of dynamic pages and to generate required pieces without contacting the origin server.

### 2.2.1 Delta Encoding

To optimize Web transfers, an origin server (or a proxy) can compute a difference between an old and a new copy of an object and communicate that difference to the client, provided that the client has that old copy of the object. The client can construct the new object by applying the difference to the old object. Sending differences instead of entire objects reduces bandwidth requirements and lowers the response time, and is called *delta encoding*. The delta encoding technique can be applied uniformly to all resources, irrespective of how frequently they change and whether they are textual or binary. Williams et al. [93] were first to suggest delta encoding when they studied various cache removal policies and envisioned changing the HTTP protocol to let caches obtain the difference between the cached and the updated version of an object.

Housel and Lindquist used delta encoding in their WebExpress system to make browsing in wireless environments possible [38]. WebExpress makes use of client- and server-side proxies in the form of the Client Side Intercept (CSI) module running within the user's mobile device and the Server Side Intercept (SSI) module running on the wired network. Both CSI and SSI modules cache a common *base object*. SSI obtains a new object from the origin server, and relates the difference (*difference stream* in the paper), consisting of a sequence of copy and insert commands, to the CSI. CSI merges the difference with the base object. Delta encoding in WebExpress is performed transparently to the browser and the origin server (and proxy servers) and does not require any changes to the HTTP protocol. The use of differencing technology in WebExpress was motivated by the fact that "...different replies from the same program (application server) are usually very similar" and focused on applying deltas to responses to CGI queries. Housel and Lindquist also noted that differencing may prove useful if applied to HTML files undergoing minor changes. They did not apply differencing to images. Evaluation of the system showed that delta encoding significantly reduced the amount of bytes transferred and latency, but was performed on a small number of test cases.

Banga et al. also used delta encoding between a client-side proxy and a server-side proxy to reduce latency in a low-bandwidth environment [5]. They proposed two variations of the delta encoding technique: *simple delta*, which is the same as in WebExpress, and *optimistic delta*, neither requiring any changes to the HTTP protocol. The optimistic delta approach builds on the fact that pages change incrementally and works as follows. If the client-side proxy does not have a cached older copy of an object, the server-side proxy optimistically sends the older copy of the object in an attempt to improve latency by anticipating that the object has not changed on the origin server. If the object has changed, the server-side proxy later sends the delta to the client. Optimistic deltas actually increase the number of bytes transferred while reducing end-to-end latency. Banga et al. evaluated the performance of their system on a small set of selected URLs and showed a reduction in latency by 12-30% across pages studied. The results also indicated that delta encoding, computed with *vdelta* [46], produces smaller update messages than simply compressed documents.

Mogul et al. [64] were the first to quantify the benefits of end-to-end delta encoding and compression using large traces of actual user requests: a packet-level trace, collected in November 1996, and a proxy trace, collected in December 1996. They also suggested in [64] and subsequently described in RFC 3229 [65], which is now a proposed standard, extensions to the HTTP protocol to support end-to-end delta encoding and compression. Mogul et al. observed that even when a Web object changes, the new *instance* is substantially similar to the old one, and sending the difference between the two can save bandwidth and reduce latency. Results of their study support these observations and show that delta encoding can provide significant improvements in response size and time, but mostly for text-based Web objects, such as HTML. Mogul et al. noted that while it is possible to extract deltas from image files and images generated by Web cameras, the resulting deltas do not reduce number of bytes by much because images are already compressed.

Mogul et al. also evaluated and compared a number of different programs to compute deltas and perform compression, and found *vdelta* [46] to be the best overall. In addition,



they measured computational overhead of creating and applying deltas and of compressing and decompressing server responses and found that throughputs for almost all computations with the library implementations of *vdelta* are significantly faster than the throughput of a T1 line (193 KBytes/sec) [64]. Mogul et al. also found that the cost of applying a delta or decompressing a response is lower than the cost of creating the delta or compressing a response. They concluded that delta encoding and compression would be useful for users of dialup and T1 lines and might even be useful for multiple hosts sharing a T3 line. Mogul et al. also suggested that cost of computing deltas can be further reduced if deltas are precomputed ahead of time and cached at the server, at the expense of additional storage.

### 2.2.2 HTML Pre-Processing

Douglis et al. in their **HTML Pre-Processing (HPP)** work [27] observed that for a common class of dynamically generated resources, such as search engine queries, most of the content on a page is static, with only small portions changing, and the locations of these dynamic portions within a page are the same. They suggested separating the *static* and *dynamic* portions of a page.

The static portion, called the *template*, contains HTML code extended with a few new tags, and can be cached. Tags added to the HTML encode macro-instructions for inserting dynamic information, and have the ability to represent powerful concepts, such as conditional branching and loops. The dynamic portion, called *bindings*, is retrieved on each access, and contains access-specific values for the variables specified within the template. The cached template is expanded with the new bindings before the page is rendered in the user's browser. The expansion can be done by a modified browser, by a proxy server, by a Java applet supplied by the content provider, or by a browser plug-in. If the template is not available in the cache, it can be retrieved from the origin server. Bindings always have a pointer to their respective template.

HPP saves network bandwidth by separating and caching static portions of dynamic

pages. Assuming that templates are cached, the size of the compressed dynamic bindings is comparable to delta-encoding with vdelta. Douglass et al. found that the size of the bindings is 4-8 times smaller than the size of the original resource, and 2-4 times smaller when both the bindings and the original resource are compressed. HPP requires no modification to the HTTP protocol, allows compact representation of repetitions within resources, and lessens the load on Web servers by letting them generate smaller amounts of dynamic data.

HPP is well suited for a class of resources which are generated upon request by software programs, such as search engines, but is not general enough to tackle other types of frequently changing resources, such as those frequently edited manually. Content designers need to learn yet another, albeit simple, language—HPP—and figure out how to encode dynamic portions for their particular situation. HPP macro-instructions have to be intertwined with HTML, making page construction more difficult. Since the template and bindings are now treated as separate resources, modifications to a page might involve updating both resources. In that case, care should be taken to properly synchronize changes. Template expansion also requires a separate pre-processor, which must be available on the client side for HPP to work.

### 2.2.3 Proxy Enhancement

A number of research projects proposed various enhancements enabling proxy servers to handle a subset of requests for dynamic pages locally, without contacting the origin server. In the *Active Cache* approach proposed by Cao et al. [13], origin servers attach cache applets (cachelets) to Web objects, and require proxies to invoke these applets upon cache hits. Once invoked, applets perform necessary processing on the proxy and generate the required response. The approach is general and flexible. Cache applets can perform various functions, such as rotating advertising banners, constructing customized pages, logging user accesses, performing delta encoding, and supporting access control. Cao et al. implemented a prototype of Active Cache in Java and showed that it increases response time by a factor of

1.5-4.0 (by 47%-75% for a “null” applet), mostly due to the increased CPU utilization [13]. The Active Cache approach raises security concerns, since proxies have to trust applets supplied by servers, and requires content providers to learn how to write applets. To address the issue of performance degradation when many applets are running simultaneously, Active Cache allows proxies to simply forward the request to the origin server, but if this happens often enough the benefit of the approach will not be realized.

Smith et al. proposed the *Dynamic Content Caching Protocol (DCCP)* to allow Web applications to explicitly specify equivalence between dynamic pages they generate [81]. The authors contrasted their approach with previous proposals, such as delta encoding, in that in their scheme neither proxies nor servers are responsible for identifying equivalence (or computing deltas): control is given to the application. For some Web applications, such as server-side image maps, on-line weather reporting services, and on-line maps, a number of syntactically different requests result in identical responses. For example, the weather conditions might be the same for a dozen different ZIP codes, or the same URL might be fetched for 50 different screen coordinates on a server-side image map. Smith et al. classified locality in dynamic Web content into three categories: *identical* requests (requests and responses are identical), *equivalent* requests (requests are different but responses are identical) and *partially equivalent* requests (requests and responses are different but responses overlap to some degree) [81]. Identical and equivalent requests can be satisfied from the DCCP-aware cache, and partially equivalent requests can be optimistically satisfied from the cache followed by the correct response from the origin server. Equivalence directives are related to DCCP-aware proxies using the extension mechanism of HTTP 1.1 cache control directives [31]. The DCCP approach is less general than Active Cache and applies only to a small subset of dynamic content on the Web.

### 2.2.4 Data Update Propagation

Challenger and Iyengar, researchers at International Business Machines (IBM) working on a Web site to host the Olympic games, observed that Web servers utilize orders of magnitude more CPU cycles to construct dynamic pages on the fly than to serve static pages. Caching dynamically generated pages at the Web server after fulfilling the first request and serving the subsequent requests from the cache significantly improves the performance. The key challenge in caching dynamically generated pages is keeping the cached copies consistent with the underlying data. Challenger and Iyengar developed Data Update Propagation (DUP) [41] mechanism and Distributed Cache Manager [16]. In the early stages the main focus of their work was to improve the performance of large Web sites withstanding high request rates and serving large number of dynamically generated pages [40].

Dependencies between underlying data and complete pages (also called objects) are represented by an object dependence graph (ODG). In a generalized graph, each vertex can represent underlying data, a fragment of an object, or a complete object. Each directed and weighted edge represents the inclusion relationship between entities. An application program is responsible for communicating data dependencies between underlying data and objects to the cache manager. When underlying data changes, the application program notifies the cache manager, which invalidates appropriate objects in the cache. Weights allow the cache manager to evaluate the importance of changes and avoid invalidations in favor of performance improvement for insignificant changes.

DUP and Cache Manager were implemented in the DynamicWeb cache [40] which is part of IBM's net.Data software. DynamicWeb cache is general enough to function as a proxy server but is better than existing proxy servers because it provides an API for application programs to explicitly cache, invalidate, and update cached objects. However, originally DUP was designed for reverse proxies and its main application is within a Web site's infrastructure. DynamicWeb was successfully deployed at a number of large and highly dynamic Web sites and proved to significantly improve the performance in the presence of

frequent changes to underlying data [40, 15].

Challenger et al. subsequently combined DUP with prefetching to improve cache hit rates. When underlying data changes, the Cache Manager immediately re-calculates all affected cached objects instead of simply invalidating them [15, 17]. Challenger et al. also proposed to use *fragments* to simplify construction of pages with the same look and feel and improve performance by caching individual fragments instead of complete dynamically produced pages [18]. They noted that selection of which part of an HTML page becomes a separate fragment is based on change dynamics and used two types of underlying data (and hence fragments) in their work: data from databases (automated feeds), which translates into *immediate fragments*, and data produced manually by humans, which translates into *quality controlled* fragments [17, 18]. Users specify how pages are composed from fragments by creating templates in an extended HTML markup language.

## 2.3 Summary

Cache consistency and caching of frequently changing objects are both well-known problems. In this chapter we discussed the most significant previously proposed approaches addressing these problems. Validation-based approaches to cache consistency balance the tradeoff between consistency guarantees and the number of validation requests. Invalidation-based approaches provide strong cache consistency, but require servers to maintain per-client state. Servers can control the amount of state they maintain, the number of invalidation messages they send out, and the amount of time they wait for unreachable clients before proceeding with object updates via leases. Shorter leases reduce server state and place tighter bound on object staleness, but require clients to renew leases more often. Longer leases reduce lease renewal traffic, but increase server state and require servers to wait longer for unreachable clients. We also discussed various ways to combine client polling with server invalidation and techniques for grouping objects into volumes. The idea of piggybacking additional information about object or volume updates onto existing traffic between caches

and servers is particularly promising.

Approaches proposed to assist in caching frequently changing objects are delta encoding, HPP, Active Cache, DCCP, and DUP. Delta encoding can be applied uniformly to all resources and was shown to be fast enough to be useful for users of dialup and T1 lines, when computed upon client requests. Precomputing and caching deltas ahead of time can reduce the cost of computing deltas, at the expense of additional storage. The HPP approach identifies frequently changing portions within dynamically generated HTML pages and replaces them with special markup, thus creating a static template. Clients cache templates and replace the special markup with the bindings that they fetch on every access. The Active Cache approach is more general, but raises security concerns and performs poorly. DCCP is a narrow solution that applies to a small subset of dynamic content on the Web. The work on DUP has laid a foundation for caching and reusing fragments of changing pages, but is more concerned with improving origin server performance than with devising an end-to-end solution.

## Chapter 3

# Background Studies

In Chapter 2 we discussed previous work on cache consistency and caching of frequently changing objects. However, certain aspects of the Web that are important for this dissertation were not addressed by previous work. Also, while some characteristics of the Web have already been studied, we wanted to carry out our own characterization studies, on more recent data sets than in previous work, to see whether the previous findings are still true. In this chapter we present three background studies investigating the following aspects of the Web:

1. One aspect that interests us is the frequency with which Web resources change. Resources that do not change at all or change on every access can easily be managed deterministically. The question is whether there are resources that change with frequencies between these two extremes, at irregular intervals. Improving the management of such resources is our goal because currently these resources are not cached at all or cached with heuristically determined expiration times, as discussed in Chapter 2. Douglis et al. studied a packet trace and found that Web objects change at widely different intervals [26]. We investigate whether these findings are still true in our first background study.
2. Another important aspect is how images and other embedded resources change relative

to their container HTML resources. If related resources change at different rates, then caches can obtain invalidations for non-deterministically changing resources while retrieving frequently changing ones. Prior work by Douglass et al. [26] indicated that images change much less frequently than HTML resources. We investigate whether that result is still valid in the first background study.

3. We need to determine whether relationships between objects are stable enough over time to rely upon them, as suggested above. No prior work that we are aware of studied whether the set of embedded images changes or remains relatively constant as the container resource undergoes modifications. We investigate this aspect in the first background study.
4. We need to better understand the predictability, locality, and extent of changes to a resource. This is particularly important for resources that change often, such as dynamically computed content. Techniques such as delta-encoding [64], HTML pre-processing [27], and active caches [13] have been proposed to allow resources that change frequently but predictably, to be cached. Also, if changes to a resource occur in the same locations, the affected portions can be separated from the resource and treated as distinct resources. We investigate this aspect in the second background study.
5. Caches issue many unnecessary validation requests (GET requests accompanied by the `If-Modified-Since` request header) to servers. If servers provide invalidations to caches for non-deterministically changing objects, caches can avoid issuing validation requests for these objects. Eliminating unnecessary validation requests would be a substantial improvement over current practice. In the third background study we quantify the reduction in the number of such validation requests to understand whether the improvements are significant.



We now describe the retrieval software and methodology used in the first two background studies and then present the three studies.

### 3.1 Retrieval Software and Methodology

We wrote software, called the Content Collector, to retrieve a set of URLs. The Content Collector supports the following configurations that determine what exactly is retrieved:

- **GetGivenURLs.** Fetch only resources identified by the provided URLs. In case of HTTP redirects (responses with the status codes 301 and 302) the Content Collector fetches the URL provided by the server.
- **GetFullPages.** Fetch resources identified by the provided URLs as described above and all objects embedded in the retrieved HTML pages. The Content Collector does not interpret or parse JavaScript code embedded within HTML, missing those objects that need to be retrieved because of the JavaScript code execution. The Content Collector detects the `FRAME`, `IFRAME`, and `LAYER` HTML tags, then fetches them along with their embedded objects.
- **GetFullPagesToGivenDepth.** Fetch all objects described under **GetFullPages**. Then identify all HTML pages accessible from the fetched HTML pages and fetch them recursively, up to a configurable level (or depth).

The Content Collector can be instructed to use specific HTTP headers in its requests to Web servers, such as `Cache-Control: no-cache`. For each retrieval of each object, the Content Collector stores the current time, a complete set of the HTTP response headers, the length of the response body, and the MD5 checksum that it computes on the object's body. The Content Collector discards images and keeps HTML and text objects if they changed from the previous retrieval. The Content Collector also parses HTML resources and records all embedded and traversal links. The reason for calculating an MD5 checksum on the

contents of each resource is to determine whether the resource changed between successive retrievals. Our calculation of the MD5 checksum is independent of the `Content-MD5` header field defined in HTTP/1.1 [31] for an end-to-end message integrity check. We could not rely on the presence of the `Content-MD5` header in server responses because servers rarely supply it.

Previous work used proxy logs, server logs, and network traces of real user requests and server responses, which constrained the resulting studies to the available data. In contrast, our approach was to retrieve each resource in a test set at intervals and for a duration appropriate for characterizing the nature of each resource in the test set. In addition, logs and traces are affected by browser and “lower-level” proxy caches, which hide some of the requested resources. We disabled caching for more complete data gathering.

Our retrieval methodology was to perform an unconditional HTTP `GET` request for each of the URLs in a test set on a daily basis using the HTTP request headers shown below for the sample URL `http://owl.wpi.edu/index.html` (the host and path vary for each request).

```
GET /index.html HTTP/1.0
Pragma: no-cache
Accept: */*
Host: owl.wpi.edu
User-Agent: Mozilla/4.03 [en] (WinNT; I)
```

The time between successive retrievals for a URL may be lengthened or shortened as needed, but we used a retrieval interval of one day.

## **3.2 Study 1: Rate of Change and Characteristics of Embedded Images**

In this background study, we investigated the first three aspects of the Web listed above. We studied a set of URLs at a variety of sites and gathered statistics about the rate and

nature of changes correlated with the resource content type. We configured the Content Collector with the `GetFullPagesToGivenDepth` option with depth one. Hence all traversal links in the home page of each site are retrieved along with the embedded images of each of these links. This approach allows us to not only follow the dynamics of individual URLs, but to follow the dynamics of the set of resources used at a site.

In this background study, we used two approaches for determining which resources to study. One approach was to identify frequently visited sites and study their home pages and pages accessible from home pages. Such resources are likely to have the most impact on long-term Web usage. We explored different sources for gathering resource usage information such as Media Metrix [60], Keynote Systems [44] and [100hot.com](#) [1]. We used home pages from a set of Web sites identified by [100hot.com](#) as a basis for our study.

Our other approach was to gather a set of URLs from current NLANR proxy logs [73]. These logs are from an upper-level cache typically servicing requests not satisfied by caches closer to clients making the requests. This approach has the advantage of focusing on URLs actually being retrieved by users across a number of different servers and content types.

### 3.2.1 Test Sets Based on Popular Sites

We constructed four test data sets using the September, 1998 ratings from [100hot.com](#). Data from the first test set, `com1`, were gathered on a nightly basis for a two-week period during October, 1998. The `com1` test set consists of home pages for 19 Web sites identified as the top 10 on-line properties by [100hot.com](#) (some properties included multiple sites).

The three remaining test sets were studied during a two-week period in November, 1998. The `com2` test set consists of 13 URLs from the next most popular sites from [100hot.com](#). The `netorg` test set was derived from the set of all sites in the [100hot.com](#) top 100 whose top level domain was other than `.com`. These sites are primarily from the `.net` and `.org` domains. The final test set, `edu`, was constructed based on rankings of the `.edu` domain site usage given by [100hot.com](#) along with the home page of [Worcester Polytechnic Institute](#)

(WPI). Because relatively few queries were included in the four test sets, we added a fifth test set `query` to our study. This test set was studied for six days in November, 1998 and included queries to ten search engines, searching for “search engines.” For this test set, the query result was retrieved along with embedded images, but traversal links were not retrieved.

### **3.2.2 Test Sets Based on User-Requested Resources**

We obtained seven daily proxy traces from NLANR [73] for the late December, 1998 to early January, 1999 time period. We extracted all HTTP `GET` requests that resulted in HTTP responses with the response code `200 OK` or `304 Not Modified`. These accesses encompassed 214,000 distinct URLs from over 33,000 distinct servers.

We chose to focus our study on non-image URLs in the traces because images are primarily retrieved as embedded images in HTML container pages and can be retrieved as needed by our study. We eliminated all image URLs, accounting for 74% of accesses, from the study set. We also eliminated all queries—URLs containing a “?”. Such queries could not be used because all parameters after the question mark were sanitized in the trace data making replication of such requests impossible. We further reduced the resulting set of 3237 URLs by removing all non-existent URLs and URLs referenced fewer than 20 times. The final set contained 1129 URLs.

We divided these URLs into five test sets based on their content type and reference count.

1. `cnt100`: resources with 100 and more references and content type `text/*` (in our data set we only had `text/html`, `text/css`, and `text/plain`). All embedded images for this set are retrieved in addition to the resource itself.
2. `cnt20`: resources with 20-99 references and the same content type as above. Embedded images for this set are not retrieved to save on system resources.

3. `audio`: resources with content type `audio/*`.
4. `appldata`: resources with content types `application/octet-stream` and `application/zip`.
5. `appltext`: other resources with application content type, primarily `application/x-javascript`.

### 3.2.3 Test Sets Summary Statistics

Summary statistics for all test sets are given in Tables 3.1 and 3.2. While all headers from all responses were saved and cataloged, the table focuses on statistics related to caching and content type.

Table 3.1: Summary Information on October/November, 1998 Test Sets

Item	Test Set				
	com1	com2	netorg	edu	query
Number of Base URLs	19	13	11	10	10
Number of Resources	1938	2048	127	110	149
Content-Type: HTML/text	15.8%	10.7%	13.4%	11.8%	6.7%
Content-Type: image	83.5%	89.2%	86.6%	88.2%	93.3%
Number of Repeated Resources	1121	910	113	110	74
Content-Type: HTML/text	21.9%	15.5%	15.0%	11.8%	13.5%
Content-Type: image	77.9%	84.3%	85.0%	88.2%	86.5%

The bottom sections of Tables 3.1 and 3.2 focus on the resources that were retrieved more than once in our tests. Because only the base set of URLs is fixed in our measurements, the actual set of images and links can and obviously did change over the course of the study (except for the four data sets in Table 3.2 where the set of URLs is fixed). Only about 50% of the resources were retrieved more than once for the commercial and query test sets while this ratio was much higher for the other two test sets in Table 3.1 and for the `cnt100` test set in Table 3.2. For multiply retrieved resources, the ratio of HTML resources is a bit higher than for all resources. As part of the background study, we further classified HTML

Table 3.2: Summary Information on January, 1999 Test Sets

Item	Test Set				
	cnt100	cnt20	audio	apldata	apptext
Number of Base URLs	122	927	7	10	63
Number of Resources	1131	927	7	10	63
Content-Type: HTML/text	10.8%	100.0%	0.0%	0.0%	0.0%
Content-Type: image	89.2%	0.0%	0.0%	0.0%	0.0%
Number of Repeated Resources	754	927	7	8	63
Content-Type: HTML/text	15.8%	100.0%	0.0%	0.0%	0.0%
Content-Type: image	84.2%	0.0%	0.0%	0.0%	0.0%

resources as “static” or “dynamic” by applying heuristics to the resource name, but found little difference in the characteristics of resources in the sub-categories.

### 3.2.4 Rate of Change

Our first step in analyzing the data was to repeat the rate of change calculations as done by Douglass, et al. [26] a year earlier on a packet trace. Our study is performed on more recent data and is not constrained to the data available in a packet trace. Our calculations are based upon the MD5 checksum computed for a returned resource and not on the information reported by the server in the `Last-Modified` or `ETag` response headers. The `ETag` header carries an entity tag—an opaque validator that caches can use to compare object versions.

Figure 3.1 shows the results for HTML and images for each of the test sets based on resources at popular sites. Figure 3.2 shows the results for all test sets based on user-requested resources. Results for the `cnt100` test set in Figure 3.2 are broken down into results for HTML resources and images.

The images for all test sets show virtually no change as found in [26]. Resources in the `audio` and `apldata` test sets in Figure 3.2 show little or no change, and resources in the `apptext` test set, also in Figure 3.2, show substantially more changes. The HTML resources show much variation in change characteristics. The `netorg` and `edu` results in Figure 3.1 show 60-70% of the HTML resources did not change (comparable to the HTML

results in [26]). The `cnt100` and `cnt20` results in Figure 3.2 show 40-50% of HTML resources never changed. The HTML resources for the commercial sets `com1` and `com2` in Figure 3.1 show much more volatility. Only 10-20% of these resources did not change during the study while 70-80% of these resources changed on each retrieval. 100% of the `query` HTML resources in Figure 3.1 changed on each retrieval. The results show that Web objects, across and within Web pages, change at widely different rates, and many objects change at irregular intervals. These findings are important and we capitalize on them in this dissertation.

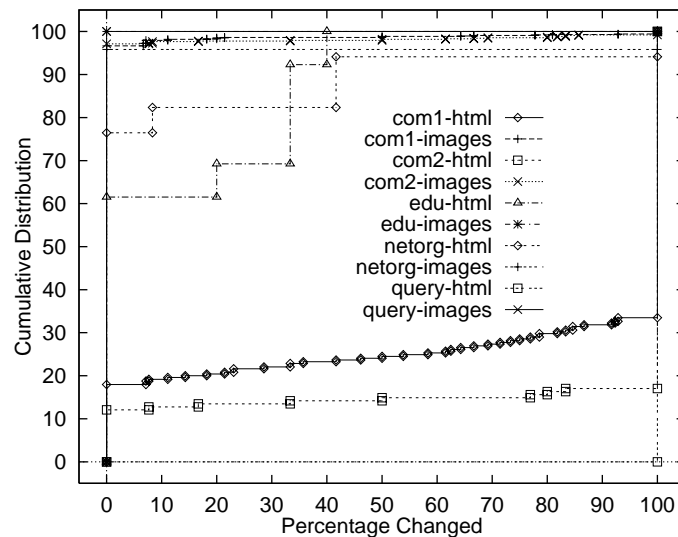


Figure 3.1: Cumulative Distribution of October/November, 1998 Test Set Change Ratio Grouped by Content Type

### 3.2.5 Characteristics of Embedded Images

The rate of change results in Section 3.2.4 indicate that HTML resources change frequently. However, what these results do not indicate is the nature and degree of changes. One question that arises is whether changes to HTML resources affect the set of embedded images, since HTML resources are often “containers” for embedded images. In this section we examine the frequency with which embedded images remain in an HTML resource between successive retrievals. Tables 3.3 and 3.4 provide results on the number of images that remain

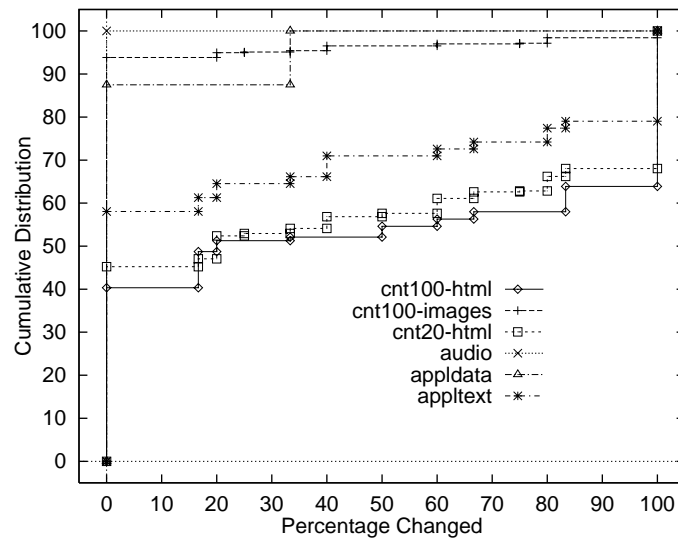


Figure 3.2: Cumulative Distribution of January, 1999 Test Set Change Ratio Grouped by Content Type

between successive retrievals of an HTML page from each test set.

Table 3.3: Number of Embedded Images and Traversal Links Remaining in an HTML Page Between Successive Retrievals in October/November, 1998

Item	Test Set				
	com1	com2	netorg	edu	query
Number of HTML Pages	245	141	17	13	10
Avg. Number of Embedded Images Per Page	9.24	31.72	8.45	16.34	25.03
Avg. Number of Remaining Embedded Images	4.67	17.57	7.32	11.39	5.64
Avg. Number of Links Per Page	73.74	63.77	14.31	28.10	75.50
Avg. Number of Remaining Links	64.10	41.39	13.52	26.16	53.69

The results show that the percentage of images remaining is a little over half for the commercial test sets `com1` and `com2`. A similar percentage was found for frequently requested URLs in the `cnt100` and `cnt20` test sets. The results for the `netorg` and `edu` test sets show that for 70-80% of resources the set of images remains the same between retrievals. The `query` test set yields the least amount of reuse on average, although the median is close to 50%. These results have two significant implications for caching:



Table 3.4: Number of Embedded Images and Traversal Links Remaining in an HTML Page Between Successive Retrievals in January, 1999

Item	Test Set	
	cnt100	cnt20
Number of HTML Pages	119	920
Avg. Number of Embedded Images Per Page	10.31	17.06
Avg. Number of Remaining Embedded Images	6.61	8.03
Avg. Number of Links Per Page	43.97	42.39
Avg. Number of Remaining Links	37.92	34.49

1. Despite the fact that HTML resources change frequently there is a significant amount of reuse of images, and
2. Cache replacement policies need to associate an image with its container resource so that if an image is no longer used by any container resource then it should be garbage collected and removed from the cache.

Tables 3.3 and 3.4 also show the frequency at which traversal links remain the same between successive retrievals. While not having direct implications for caching, the results show that a significant ratio of links remain between retrievals.

### 3.2.6 Follow-Up Study

Since the original study was conducted four years ago, we repeated the study over a period of ten days in May/June 2002 to determine if the results presented here are still true. We used the same methodology as before, but this time we examined the 50 most popular URLs requested by users. We obtained these URLs from seven NLANR proxy logs collected in May 2002. Summary statistics on our new data set are shown in Table 3.5. We can see that the statistics are similar to those shown in Table 3.1. The rate of change results for the html objects, images, and other objects are shown in Figure 3.3. Embedded objects other than images in our data set are Cascading Style Sheet (CSS) documents, external JavaScript code and Shockwave Flash files. We can see that the results mimic those in

Figure 3.1, in that images exhibit virtually no changes, and HTML resources show much variation in their change characteristics. Embedded objects other than images exhibit more changes than images, but substantially fewer changes than HTML resources.

Table 3.5: Summary Information on May 2002 Test Sets

Item	Test Set
	nlanr-popular
Number of Base URLs	50
Number of Resources	5136
Content-Type: html/text	14.4%
Content-Type: image	75.3%
Content-Type: other	10.2%
Number of Repeated Resources	1972
Content-Type: html/text	6.3%
Content-Type: image	85.8%
Content-Type: other	7.9%

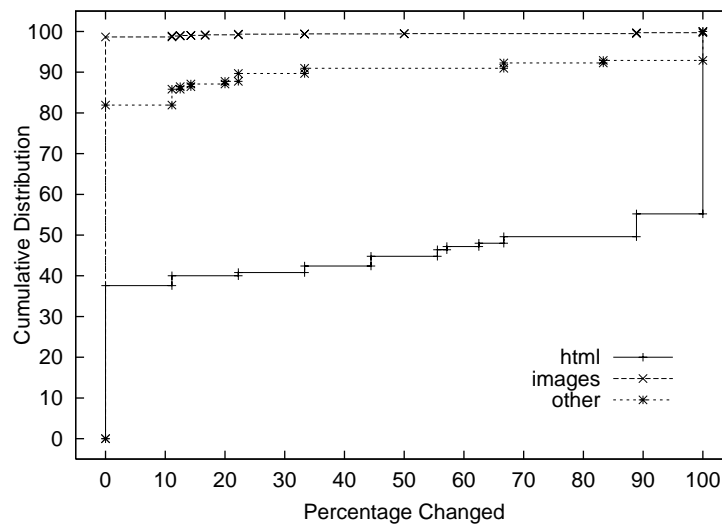


Figure 3.3: Cumulative Distribution of 2002 Test Set Change Ratio Grouped by Content Type

We also computed the average number of embedded objects and traversal links in Web pages, and the average number of embedded objects and traversal links that remain on pages across retrievals. The results for the May 2002 data set are shown in Table 3.6.

The results indicate that despite frequent changes exhibited by HTML resources, the reuse of embedded objects and traversal links is substantial. The results are similar to those obtained in the Fall of 1998.

Table 3.6: Number of Embedded Objects and Traversal Links Remaining in an HTML Page Between Successive Retrievals in May 2002

Item	Test Set
	nlanr-popular
Number of HTML Pages	123
Avg. Number of Embedded Objects Per Page	11.01
Avg. Number of Remaining Embedded Objects	9.61
Avg. Number of Links Per Page	37.01
Avg. Number of Remaining Links	29.47

### 3.3 Study 2: Changes to HTML Resources and Content Reuse

To better understand the nature of changes to HTML resources, potential for dynamic content reuse, and byte savings resulting from constructing pages from components, we studied HTML resources at popular Web sites and frequently requested HTML resources, as described below.

#### 3.3.1 Methodology

We used the Content Collector with the `GetFullPages` configuration and retrieval methodology discussed earlier in Section 3.1 and retrieved content for 11 days in October/November, 1999. In a negligibly small number of cases, images could not be retrieved due to use of `https` as the protocol portion of the URL. We kept track of images retained on pages between retrievals and assumed that images do not change, as was shown by Douglis et al. [26] and confirmed in Section 3.2.4.

To automatically analyze changes to HTML objects, we developed software, called the Chunking Tool, that decomposes HTML objects into smaller *chunks*. Decomposition is based on the inherent structure of HTML objects and approximates how a content designer might decompose an HTML page into smaller components. For example, the `<TABLE>` tag is commonly used to define structure. Our Chunking Tool separates all content enclosed between the `<TABLE>` and `</TABLE>` tags into a separate chunk. We calculated an MD5 checksum for each chunk and used it to determine the number of chunks and number of bytes common between two retrievals of a page.

### 3.3.2 Test Sets

We used four data sets for this portion of the background study. One of them, `Cnt300`, was derived from seven NLANR proxy logs [73], collected at the end of October, 1999. Aggregate number of entries in all seven logs was over 8.5 million. We selected only accesses to HTML resources with the HTTP response codes 200 `OK` and 304 `Not Modified` and then filtered out all entries that did not have a valid content type, leaving us with 866,000 distinct URLs. We also eliminated query URLs because they are sanitized and cannot be retrieved, as explained in Section 3.2.2. Since in this study the Content Collector retrieves images embedded in Web pages, we further pruned the test set so it contains about 100 URLs. We eliminated URLs with fewer than 300 accesses bringing the resulting test set to 128 URLs.

We constructed three other data sets based on the data obtained from `100hot.com` [1] on November 2, 1999. The `Top50` data set contains the 50 most popular Web sites. Some sites are represented by more than one host name, so this data set has 71 URLs. The `Ecom` data set contains the home pages of the 50 largest shopping sites (business-to-consumer). The `Srcheng` data set lists home pages of eleven well-known search engines: `altavista`, `askjeeves`, `excite`, `google`, `goto`, `hotbot`, `infoseek`, `lycos`, `mckinley`, `northernlight`, and `webcrawler`.

### 3.3.3 Content Reuse

Results for the potential of content reuse due to HTML page decomposition with our Chunking Tool are shown in Table 3.7. The second column shows the average number of bytes for the base HTML object in each test set. The first value in parentheses is the percentage of bytes that could have been reused from the previous retrieval if objects were composed from our chunks. The second value in parentheses is the percentage of bytes that can be reused if we only considered cases when objects do not change between retrievals. The high rate of change for the HTML objects is consistent with the findings discussed in Section 3.2.4. The results indicate that separating static portions of frequently changing HTML objects from dynamic portions results in substantially higher amount of content reuse across all test sets.

Table 3.7: Content Reuse for Popular Web Pages in October/November, 1999

Test Set	HTML Bytes (% Reuse, No Chg)	Images (% Reuse)	Image Bytes (% Reuse)	Total Bytes (% Reuse)	diff -e
Cnt300	11495.1 (77%, 23%)	5.6 (85%)	14023.8 (70%)	25519.0 (73%)	84%
Top50	17276.7 (75%, 16%)	12.1 (86%)	23921.0 (75%)	41197.6 (75%)	82%
Srcheng	14977.8 (75%, 6%)	8.4 (89%)	10686.5 (79%)	25664.3 (77%)	81%
Ecom	16826.4 (70%, 16%)	16.2 (92%)	33061.0 (83%)	49887.4 (79%)	76%

The third and fourth columns in Table 3.7 show the average number of embedded objects and embedded object bytes respectively for each test set. The percentages in parentheses show the amount of reuse from the previous retrieval of the page. The number of objects shows a high rate of reuse with a bit less for the object bytes.

The total number of bytes (HTML and embedded objects) along with the percentage of their reuse is indicated in column five. Approximately 75% of the bytes needed for these popular Web pages could be reused from the previous retrieval of the page. While computing these figures we measured only changes to the object content and ignored any cache control directives returned by the server.

The last column of Table 3.7 shows the percentage of object bytes saved if only the difference between successively retrieved copies of the base HTML object is transmitted to caches, instead of the entire object. We performed differencing using UNIX *diff* command with option *-e*—the form required for the *ed* text editor. This was not the best differencing tool tested in [64], but it is widely available. As shown in Table 3.7, the *diff -e* output is slightly more efficient in representing the content reuse than our Chunking Tool. These better results can occur because even chunks that change may have large portions that do not. If HTML pages are constructed from distinct components, then differences can be computed for each component.

### **3.4 Study 3: Using Object Relationships to Eliminate Unnecessary Validations**

In this study, we quantify the reduction in the number of unnecessary cache validation requests. We use the same NLANR proxy logs as in the previous study.

#### **3.4.1 Methodology**

We assumed that each 304 Not Modified response in the logs could be eliminated if a 200 OK or another 304 Not Modified response came from the same server within a 10 second window (virtually no variation was found for larger window sizes) on either side of the 304 Not Modified response (approximating the same page). This technique is most beneficial for elimination of validation requests for embedded objects, such as images, CSS, and Shockwave files. We determined whether a response carries an embedded object by examining content type stored in the logs and by applying heuristics to the URLs present in the logs (content type was often missing from the logs). Our heuristics examined URLs for known file extensions, such as *gif*, *jpg*, *css*, *js*, *swf*, etc.

3.4.2 Results

Results for each of the seven NLANR proxy logs are shown in Table 3.8. The percentage of 304 Not Modified responses is high, even though NLANR caches generally serve as second-level caches. The third column shows that 15-32% of all requests result in a 304 Not Modified response, which is consistent with previously published results [49, 70, 4].

Table 3.8: Occurrence and Potential Elimination of Validation Checks in October/November, 1999 NLANR Proxy Logs.

Proxy	Number of Responses (% Total)				
	200/304	all 304	emb. objs. 304	in 10 sec window	
				all 304	emb. objs. 304
bo1	541056	155762 (29%)	143195 (26%)	87749 (16%)	86845 (16%)
bo2	659583	185935 (28%)	170625 (26%)	113349 (17%)	111726 (17%)
lj	413290	78283 (19%)	69780 (17%)	54210 (13%)	53393 (13%)
pa	418096	62377 (15%)	56379 (13%)	38994 (9%)	38324 (9%)
pb	505395	161539 (32%)	143407 (28%)	97727 (19%)	93944 (19%)
sd	288723	72909 (25%)	66199 (23%)	44301 (15%)	43643 (15%)
sv	872231	172385 (20%)	156794 (18%)	110770 (13%)	109069 (13%)

More important for our study is the large percentage of 304 Not Modified responses that contain a related 200 OK or another 304 Not Modified response in their window and the large percentage of these 304 Not Modified responses that are for embedded objects. Column 6 shows that 9-19% of all object requests could be eliminated by removing validations that fall within a window of a 200 OK or another 304 Not Modified response from the same server. These results are significant. They show the majority of validation requests currently handled by servers due to inefficient cache consistency mechanisms can be eliminated.

We repeated this investigation two and half years later, in May 2002, using seven new NLANR proxy logs. Our updated results, shown in Table 3.9, are consistent with the previous findings.

Table 3.9: Occurrence and Potential Elimination of Validation Checks in May 2002 NLNR Proxy Logs.

Proxy	Number of Responses (% Total)				
	200/304	all 304	emb. objs. 304	in 10 sec window	
				all 304	emb. objs. 304
bo1	174692	33474 (19%)	29928 (17%)	20826 (12%)	20315 (12%)
bo2	216955	43288 (20%)	38098 (18%)	28573 (13%)	27807 (13%)
pa	73285	11589 (16%)	10499 (14%)	7747 (11%)	7598 (10%)
pb	770677	198938 (26%)	178694 (23%)	157195 (20%)	154306 (20%)
rtp	1196419	399832 (33%)	362304 (30%)	339702 (28%)	334229 (28%)
sd	472673	93508 (20%)	84596 (18%)	70693 (15%)	69507 (15%)
sj	96210	13655 (14%)	12762 (13%)	8591 (9%)	8428 (9%)

### 3.5 Summary

This chapter presented a series of studies that investigated various aspects of the Web that this dissertation builds on. We examined how resources change at a collection of servers and found that changes in objects composing Web pages span an entire spectrum—from no changes to changes on every access. We also found that while HTML resources change frequently, their overall structure remains the same and some of the objects embedded in these HTML pages are retained across page retrievals. The improved understanding of the nature of changes to Web resources and relationships between objects composing pages highlights inefficiencies in current approaches to caching and points at the potential for improvements in Web cache performance. One potential improvement is to explicitly associate embedded objects with their containers, so that caches can garbage collect objects that are no longer embedded in any pages. Another potential improvement is to use retrievals of frequently changing container objects to validate or invalidate embedded objects retained within these containers. In fact, we showed that the latter improvement can also eliminate most of the unnecessary validation requests that are present in the current Web. We also evaluated component-based approach to page construction and found that it increases the reuse of cached page bytes by 50%. Based on the improved understanding of the nature of Web



objects and inefficiencies in how they are currently managed, in the next chapter we define the problem that this dissertation addresses.

## Chapter 4

# Problem Statement and Approach to Deterministic Object Management

In Chapter 2, we discussed various issues involved in the management of distributed objects and the extensive body of previous work that examined and addressed these issues. Chapter 3 presented a series of background studies that helped us better understand the nature of Web objects. Coupled with the results from previous work, this better understanding points at the potential for improvements in the performance of caching in distributed systems. In this chapter, we define the problem this dissertation addresses and then present our approach to that problem.

### 4.1 Motivation

Many items that surround us in everyday life are built from heterogeneous components, or objects, combined to produce a whole, finished product. A computer monitor or an office chair is constructed from components that have different shapes, color, and are made from

different materials. Similar to items in the physical world, content available in distributed systems is also often produced via composition of heterogeneous objects. Web pages may contain a combination of text, graphics, audio, animation and executable code. Modern computer games involve sophisticated virtual worlds with complex interaction between numerous simulated objects—buildings, people, wizards, monsters, and weapons. SMIL [80] presentations, CAD projects, MPEG-4 clips—all combine individual heterogeneous components to produce a whole.

As a motivation for our problem, we consider one example of a composite object, a Web page, shown in Figure 4.1, which mimics the home page of a popular news portal. Our choice of this example is based on the fact that the Web is ubiquitous, and more readers are likely to be familiar with the home page of a news portal than with a SMIL presentation, specifics of a CAD project, or a distributed computer game.

The container object  $CO$  in Figure 4.1 is changing frequently—every few minutes—because content designers update the top story, and add and remove links leading to the major news articles. Irrespective of the manual updates, every request for  $CO$  results in a different response because the origin server dynamically generates  $CO$ , changing which ad banner image to display and where on the page to place it. Servers either explicitly mark  $CO$  as uncacheable or supply no cache control information at all. Caches usually do not cache such objects. Content designers manually update embedded objects  $EO1$ — $EO3$  at irregular and unpredictable intervals. The time of the next update is unknown, but servers can provide `Last-Modified` information for these objects. Instead of making changes to objects  $EO4$  and  $EO5$  content designers replace them within  $CO$  with new objects  $EO4'$   $EO5'$  respectively. Servers should be, and sometimes are, configured to explicitly assign these objects large expiration times. Servers can also provide `Last-Modified` information for these objects.

In Chapter 2, we discussed how existing and proposed approaches to cache consistency manage objects  $EO1$ — $EO5$  and how these approaches balance the tradeoffs between va-

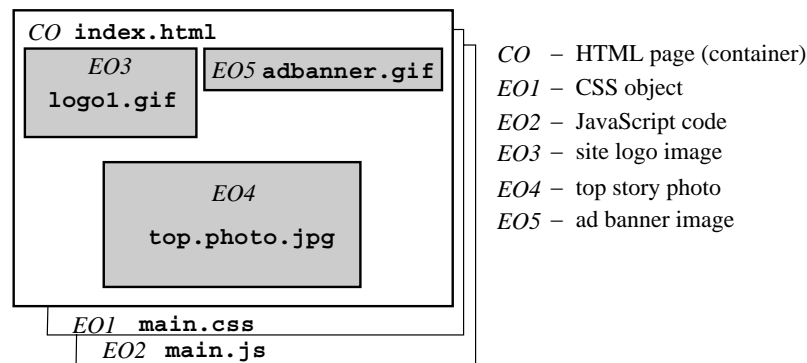


Figure 4.1: Home Page of a Popular News Site

validation requests, consistency failures, and the amount of per-client server state. In the context of the example in Figure 4.1 and the performance tradeoffs of validation-based and invalidation-based consistency approaches, we now define the problem of managing a set of heterogeneous objects in a large distributed environment in the presence of distributed caches.

## 4.2 Problem Statement

Given a set of related objects, with cached copies of some or all of these objects placed throughout the network, ensure that objects are managed deterministically so no client receives an outdated replica of an object, while minimizing the utilization of client, cache, network, and server resources to ensure the scalability of the overall system to a large number of clients and objects.

## 4.3 Hypothesis

The hypothesis of this dissertation is that we can design an object management approach that improves upon existing heuristic- and invalidation-based object management techniques so that a group of related objects in a distributed system can be managed with both consistency and efficiency. We postulate that we can exploit object change characteristics

and relationships to design such an approach. Given the tradeoff between consistency and efficiency, our main objectives are: 1) eliminate consistency failures and 2) maintain no per-client state at the server. Within the constraints of the main objectives, our third objective is to minimize the amount of consistency maintenance traffic generated by servers and caches.

## 4.4 Foundation for Our Approach to Deterministic Object Management

The fact that objects are combined together to produce a larger object suggests that objects are tied with relationships. Existing cache consistency approaches view each object in Figure 4.1 in isolation from other objects and synchronize each object with its replicas independently of other objects, thus ignoring information about object relationships. We believe we can exploit relationships between heterogeneous objects to address the problem defined above. For example, servers can first examine objects constituting the page in Figure 4.1 and identify the container *CO* as the most frequently changing object. Servers then inform caches that objects *EO1—EO5* should be cached until servers explicitly invalidate them. Servers also instruct caches to fetch object *CO* on every access. When caches subsequently return to obtain a new version of *CO*, servers piggyback invalidations for objects *EO1—EO5*. We could also separate parts of the container *CO* that change frequently from parts that change rarely and treat the resulting components as distinct objects.

In our background studies we examined the Web for the presence of key elements—heterogeneity of object changes and object relationships—and found that these elements are present. We also found that exploiting object relationships can reduce validation requests and constructing pages from components can save bytes. In the rest of this chapter, we describe how we exploit the information about object relationships and object heterogeneity for deterministic object management in distributed systems.

## 4.5 Object Change Characteristics

Objects differ along a number of dimensions: size, content type, and frequency of change. All of these object characteristics may have implications for object management. In this dissertation we focus on the consistency issues, and, therefore, are more interested in object characteristics that directly influence consistency of object replicas. The characteristic that interests us most is the frequency of object changes. In this section we first discuss how objects change and then provide definitions of object change characteristics. We also suggest how information about object changes can assist in object management.

### 4.5.1 Object Changes

In Section 3.2.4, we studied the rate of change of real Web objects and found objects that never change, that change on every access, and that change at intervals between these two extremes. In Section 4.1, we discussed an example of a realistic Web page where characteristics of objects mimic the findings of our experimental studies. The natural way to classify objects, therefore, is based on how frequently they change. Such a classification should span the entire spectrum of possible update intervals. The example in Section 4.1 also mentions another dimension along which we can classify object changes—predictability of changes. An object could be updated at well-known or at unpredictable intervals. There could be other dimensions along which one could classify object changes. For example, one could categorize all changes based on their extent—whether an object was changed just a little, substantially, or whether the object’s content is completely different from the previous version. In fact, the HTTP/1.1 protocol [31] provides a mechanism to mark an entity tag as “weak” indicating that servers prefer to change entity tag validators only on semantically significant changes. While taking into account the extent of changes could improve performance via techniques such as delta encoding, as discussed in Section 2.2.1, the extent of changes is irrelevant for issues of consistency. In this dissertation, we consider only the frequency and predictability of object changes.

### 4.5.2 Definitions of Object Change Characteristics

Classifying objects based on how frequently they change and whether their changes are predictable is a problem of partitioning a two-dimensional space into zones, or categories. We have identified four categories of object changes, shown in Figure 4.2. The two-dimensional space can be divided into two subspaces based on the predictability of changes, since a change is either predictable or not. The three categories on the left side of the figure represent predictably changing objects. Objects in these categories can be managed *deterministically* because the server has a priori knowledge at what time or upon which event the changes occur. The category on the right side of the figure represents objects that change unpredictably and cannot be managed deterministically on their own. Objects in the unpredictable category are the ones for which we must find a way to manage them deterministically. Our division of the two-dimensional space based on the frequency of object changes results in three subspaces: one where objects never change, another where objects change on each access, and the subspace that includes all changes in between the two extremes.

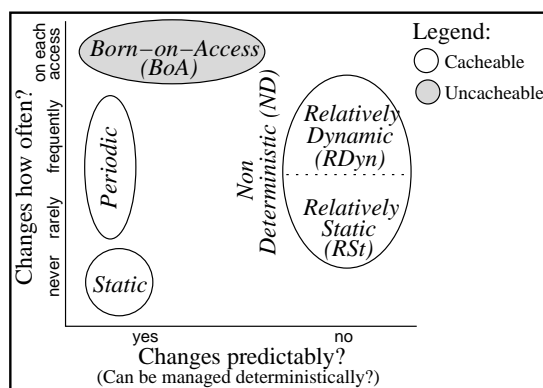


Figure 4.2: Classification of Object Change Characteristics

We have shown the logic behind our classification of objects based on their change characteristics. We now define each category of object change characteristics and give realistic examples of objects in each category. We use the following notation:  $t_0$  is the

creation time of an object;  $t_i$  is the time of the  $i$ th modification of an object; and  $r$  is the time at which a cache receives a request for an object. In formally defining change characteristics, we are asking the following question: “Given  $t_i$  and  $r$ , can one determine  $t_{i+1}$ , and if so, how?” If the server can compute  $t_{i+1}$  for an object, then it can provide caches with explicit instructions for how to manage the object.

An object is classified as **Static (St)** if the time of its first modification is given as:  $t_{i+1} = \infty, i = 0$ . Caches can store St objects for an arbitrarily long time and do not need to validate them because St objects never change. Examples of St objects are articles in on-line digital libraries and mailing list archives.

An object is classified as **Periodic (Per)** if its modification times can be computed as follows:  $t_{i+1} = f(t_i)$ . In the simplest case,  $f(t_i) = t_i + \text{const}$ . In general,  $f(t_i)$  can be arbitrarily complex. In practical terms, Per objects change at predictable intervals, such as every few minutes, every hour, day, or week. Servers can attach explicit expiration times to Per objects. Caches can deterministically store such objects and mark them as stale when they expire. Examples of Per objects are samples automatically collected by a device at specified intervals, such as snapshots captured by a camera.

A **Born-On-Access (BoA)** object is one whose next modification time coincides with the next request for it:  $t_{i+1} = r$ . The content of a BoA object is unknown until the object is accessed due to its dependency on client-specific information contained in the request, statistics accumulated by the server, or other data available only at the time of the request. Servers should inform caches that BoA objects must be retrieved on every access.

An object is classified as **Non-Deterministic (ND)** if it is not possible to determine the time of the next update, even if an update is imminent:  $\nexists f(t_i)$ , such that  $t_{i+1} = f(t_i)$ . The sub-classification of the ND objects into **Relatively Dynamic (RDyn)** and **Relatively Static (RSt)** is based on the relative frequency of their updates. RSt objects are not expected to change in the near future, while RDyn objects are expected to undergo modifications in the near future. We address the non-deterministic nature of freshness



intervals for these objects in Section 4.8 and discuss how the distinction is used.

## 4.6 Types of Object Relationships

In distributed systems, objects are often related to each other. In this section we examine a wide range of object relationships, discuss the attributes that objects share, and show how to exploit the relationships for object management.

### 4.6.1 Composition

One type of object relationship is a *composition* relationship, where each object is a building block of a whole, finished entity, such as a document or a presentation. In the context of object composition, there is the notion of a *container* (or parent) and *embedded* (or child) objects. A container is an object that contains place holders for, or descriptions of, embedded objects. Embedded objects may in turn be containers. For example, an HTML page may embed a FRAME or a LAYER containing an image. Figure 4.3a presents a graphical depiction and Figure 4.3b shows a tree representation of the composition relationship.

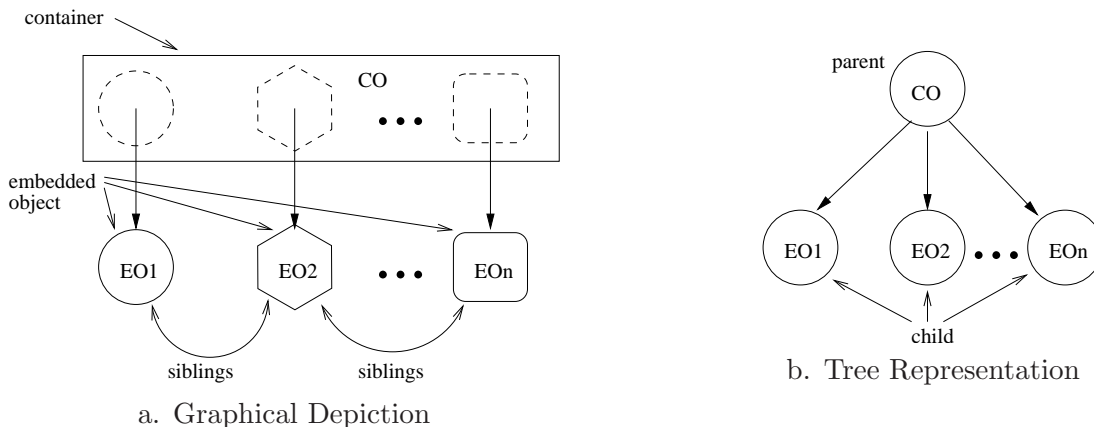


Figure 4.3: Container Object and a Set of Embedded Objects

The set of objects composing a document could be treated as a volume. A client's retrieval of the container or an embedded object from a server allows the server to piggyback

invalidations for other objects from the same volume that the client has cached.

#### 4.6.2 Temporal

Objects composing a document or a presentation may have a temporal ordering, such that one set of objects overlaps with or follows another set of objects in time. Consider a SMIL [80] multimedia presentation that simultaneously displays text and images, and plays out an audio clip and a video fragment, followed by a longer movie accompanied by a sound track, concluding with additional text and images. Retrievals of the objects required at the start of the presentation yield opportunities for validation (or invalidation) of cached objects required later in the presentation.

#### 4.6.3 Common Dependency

Servers group objects generated from the same underlying data, such as a database, into volumes. Caches cache and reuse copies of these objects until the underlying database changes and the server invalidates the entire volume. Caches mark all invalidated objects as stale and validate them with the server before reuse.

#### 4.6.4 Common Change Characteristic

Objects that have the same change characteristic may be considered related. Such objects may be updated by a script that runs at a given time or may depend on the same underlying data source. This type of relationship could be exploited if objects also have another type of relationship. For example, all BoA objects that also have a composition relationship could be bundled together for more efficient content delivery [96].

#### 4.6.5 Shared Objects

Objects that belong to more than one composite object are shared. For example, Web pages at a site may all include a corporate logo or a navigation menu. Shared objects should be

grouped into a *global* volume and managed separately from the page-specific objects. Ideally, objects within the global volume would also share the same change characteristic so that few invalidations would be needed for infrequently changing objects.

#### 4.6.6 Access Patterns

Establishing relationships between objects based on user requests was proposed in [21]. Grouping objects into volumes based on the probability of a subsequent request is an opportunity for servers to provide hints to client caches, make prefetching predictions, or improve heuristic cache consistency. However, since the relationship is probabilistic, and subsequent accesses are not guaranteed, unlike with the composition relationship, this type of relationship cannot be used alone for ensuring strong cache consistency.

#### 4.6.7 Structural Organization

Grouping objects based on structural organization, such as directory structure, was proposed in [21]. Objects (files) residing in the same directory may, in fact, share a common characteristic. They could belong to the same composite object or have the same change characteristic. A separate directory may be used for globally shared objects. Thus, structural organization may indicate that objects are related, but does not guarantee that this relationship can be exploited for managing objects.

### 4.7 Combining Object Relationships with Object Change Characteristics

We have provided a classification of object change characteristics and discussed various relationships that objects may have. We now show how object relationships can be combined with object change characteristics and identify combinations that are the most useful for object management. We focus on the composition relationship as it is a useful one to

exploit.

In Figure 4.3b we showed a tree representation of a composition relationship for a general case of a parent object and  $n$  child objects. Here we first consider a simple case of two objects—a parent and a child—tied by a composition relationship, as shown in Figure 4.4. There are four possible change characteristics that can be assigned to each of the two objects. The total number of possible combinations of object change characteristics is thus 16, as shown in Figure 4.5. Not all of these combinations are useful, however. St objects never change and thus their replicas never need to be synchronized with the originals. If a cache has a copy of a St object, it has no need to contact the server again. Therefore, the relationships that St objects have with other objects, shown in Figure 4.5 with the fill style S1, cannot help us with object management and we eliminate them from further consideration. That leaves us with nine possible combinations.

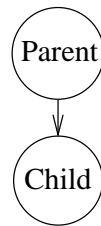


Figure 4.4: Tree Representation of a Composition Relationship Between Two Objects

Objects with the Per change characteristic change predictably, as we discussed in Section 4.5 and, therefore, can be managed deterministically on their own. Can Per objects help manage other objects? They do change, unlike St objects, and must be periodically fetched from the server. The points in time when they need to be fetched, however, may not coincide with the client request for the related object. If requests for the two related objects arrive less frequently than the update interval of the Per object, then the relationship is useful. If, however, requests are more frequent than Per object updates, then the relationship is not useful. In general, requests can arrive at any point in time. We thus

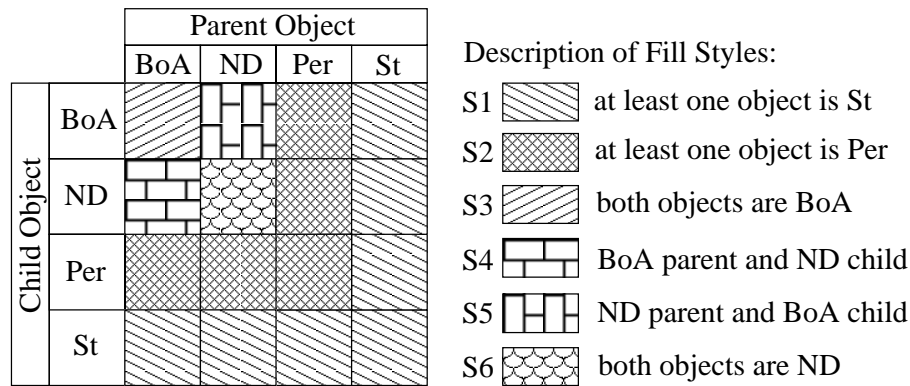


Figure 4.5: Possible Combinations of Change Characteristics for Two Related Objects

cannot predict whether in any given situation the relationships that Per objects have with other objects, shown in Figure 4.5 with the fill style S2, is going to be useful or not. We eliminate Per objects from further consideration in the context of object relationships. That leaves us with four possible combinations of object change characteristics.

Of the four remaining combinations, we eliminate one where both objects have the BoA change characteristic, shown in Figure 4.5 with the fill style S3, since both objects can be managed deterministically on their own. We could, however, exploit the fact that the two objects have a common change characteristic and bundle them together into one (larger) BoA object to reduce the number of required requests to the server and achieve more efficient delivery [96].

We are left with the following combinations:

- **BoA-ND** and **ND-BoA**. Each of these two combinations involves a BoA and an ND object. The BoA-ND combination has a BoA parent object and an ND child object and is shown in Figure 4.5 with the fill style S4. The ND-BoA combination has an ND parent object and a BoA child object and is shown in Figure 4.5 with the fill style S5. The relationship that a BoA object has with an ND object is useful for our purpose. Since the ND object changes unpredictably, it cannot be managed deterministically

on its own. The retrieval of the related BoA object, however, can be used to manage the ND object. Caches can be instructed that they may store the ND object until the server explicitly notifies them that the ND object has been updated. The server provides such notification when the cache retrieves the related BoA object.

- **ND-ND.** In this combination, shown in Figure 4.5 with the fill style S6, both parent and child objects have the ND change characteristic. The relationship between two ND objects does not appear to be useful, since both objects change unpredictably and both cannot be managed deterministically. One approach is to manage both objects heuristically, and use the retrieval of one object to invalidate the other. Our goal, however, is deterministic object management. Therefore, we force the validation of one of the two ND objects on each access. The server will provide invalidation for the other ND object, if it has changed, upon receiving such a validation request from the cache. When deciding on which object of the two to validate on each access, the distinction between RSt and RDyn subcategories becomes important. Validating a RSt object results in fewer unnecessary requests to the server.

We have examined 16 possible combinations of object change characteristics that two objects in a composition relationship produce, and selected three combinations that are useful for deterministic object management. These three combinations are shown in Figure 4.6. Our discussion so far involved only two objects—the simplest case. We now extend the discussion to an arbitrarily large number of related objects. Since we have already determined that relationships that St and Per objects have with other objects are not useful for our purpose, we do not consider these two types of objects here.

Suppose we add one more ND object to each of the three combinations in Figure 4.6 such that the container object embeds two objects instead of one or the child object becomes the parent to the new ND object. In each of the three cases, the newly added ND object should be grouped with the other ND object to produce a volume. The retrieval of the BoA object in the first two combinations, and the retrieval of one of the three ND objects

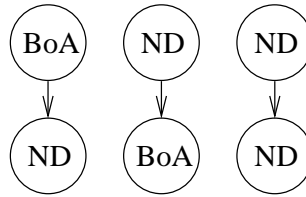


Figure 4.6: Useful Combinations of Object Change Characteristics

in the last combination, invalidates all objects in the volume that have changed since the previous retrieval. It does not matter whether we have only one ND object that is being managed or two—the approach to their management remains the same. We can add more ND objects, as many as desired, and group them all into a volume and manage them in the same fashion as a single ND object.

Suppose instead of an ND object we add one BoA object to each of the three combinations in Figure 4.6, again as a child of the existing container or the child of the existing child object. While the newly added BoA object can be managed deterministically on its own, the question is whether the addition of the BoA object affects the management of the existing ND object. To answer the question we consider the three combinations separately:

- **BoA-ND.** The BoA-ND combination now has two BoA objects, each of which could potentially be used to obtain invalidations for the ND object. We choose to retain the original, top-level BoA object for that purpose as it is retrieved before any other object in the tree. So the addition of a new BoA object does not change how the ND object is managed. We can add as many BoA objects to the BoA-ND combination as desired and still manage the ND object using the top-level BoA object.
- **ND-BoA.** Adding one or more BoA objects to the ND-BoA combination does not change how the ND object is managed. We can use the existing BoA object or one of the newly added ones to manage the ND object.
- **ND-ND.** Before the BoA object is added to the ND-ND combination, one of the two ND objects is used to validate the other one. With the addition of the BoA object

we no longer have to force the retrieval of one ND object. We can now group the two ND objects into a volume and use the new BoA object to manage all objects in the volume. This management strategy is the same as the one we applied to the ND-BoA combination. Adding subsequent BoA objects to the ND-ND combination has no further effect on the management strategy.

In all three cases just discussed, it might be possible to bundle two or more BoA objects into one, as discussed earlier. Bundling multiple BoA objects into one does not affect the management strategy.

We conclude that the three combinations shown in Figure 4.6 depict important combinations of a composition relationship and object change characteristics that are useful for deterministic object management. Adding more objects to each of the three combinations leads to more complex trees than those shown in Figure 4.6. However, we showed that objects in more complex trees, irrespective of the change characteristics of the newly added objects, can be managed in the same way as objects in one of the three simple trees in Figure 4.6. We call each of the three combinations a *pattern*. In the context of a Web page we call each combination a *page pattern*.

The set of objects composing a Web page in Figure 4.1 can be represented by a tree, as shown in Figure 4.7. If we replace the names of objects in the tree with their change characteristics we obtain a tree shown in Figure 4.8. The two St objects can be managed deterministically on their own, as can the BoA object. The three RSt objects need to be grouped into a volume. The retrieval of the BoA object is used to obtain invalidations for the volume. So, the management strategy for this Web page is the same as for the BoA-ND combination. We thus say that this Web page can be represented by the BoA-ND page pattern.



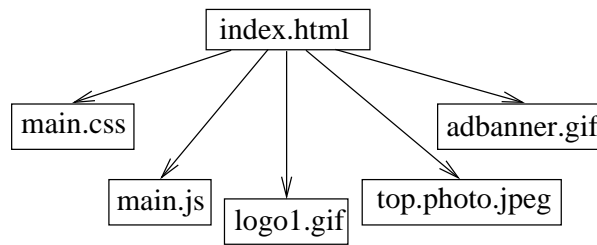


Figure 4.7: A Tree Representing the Home Page of a Popular News Portal

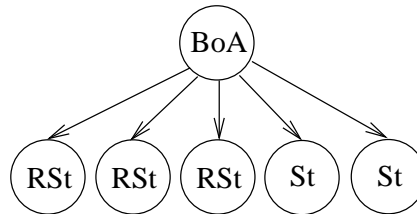


Figure 4.8: A Tree Representing Change Characteristics of Objects Composing the Home Page of a Popular News Portal

## 4.8 Using Object Relationships to Ensure Strong Cache Consistency

In the previous section, we have discussed all possible combinations of object change characteristics with the composition relationship between objects and identified three important patterns. In this section, we describe our approach to deterministic management of distributed objects that uses these patterns.

Our approach, called MONARCH (Management of Objects in a Network using Assembly, Relationships, and Change Characteristics), takes advantage of the unique opportunities presented by a large distributed system, such as the Web, where individual objects with a mix of change characteristics are often grouped together into a composite object (a Web page). In MONARCH, servers first classify objects based on object change characteristics and group related objects into volumes. In general, any set of related objects could be called

a volume. MONARCH groups objects constituting a composite object into a volume. After objects are classified, servers analyze objects in each volume and determine which of the tree patterns the volume can be described by. The servers also designate one of the objects in each volume, the *manager* object, to manage all ND objects in the volume. The servers then assign all objects in each volume *Content Control Commands* (CCCs). Servers and caches use these commands to manage all objects deterministically. When a cache needs to serve objects from a particular volume, it must first contact the server to retrieve or validate the manager object for that volume. The cache indicates to the server which volume it is interested in and which version of that volume it currently has. The server satisfies the request for the manager object and piggybacks [49] invalidations for those ND objects from the same volume that have changed since the previous request from the cache. For the server to be able to invalidate these ND objects on subsequent visits from clients, the server must know the versions of these ND objects that each client has. MONARCH does not maintain per-client state at the server, however, and servers do not keep track of which versions of which objects they served to clients. Instead, servers supply clients with version information for each object and volume identifier and volume version. Servers rely on their clients to provide that information on subsequent visits.

#### 4.8.1 Retrieval Order

A composition relationship leads to certain ordering of object retrievals: a parent object is retrieved before its child objects. In fact, a client may not even know which child objects are available until it retrieves and examines a parent object. MONARCH exploits the retrieval order of the composition relationship to enable servers to report volume information to their clients. Servers attach volume information to the top-most container object.

### 4.8.2 Selecting a Manager Object

The manager object is determined by a given pattern of the set of objects. If a pattern is BoA-ND, then the top-most BoA object is the manager. If a pattern is ND-BoA, then one of the BoA children of the top-most ND object is the manager. Unless all BoA objects are combined into one (larger) BoA object, based on the fact that they have the same change characteristics, the pattern alone does not completely determine which of the BoA objects should be chosen as the manager. One approach would be to randomly select one BoA object. Since all BoA objects must be retrieved from the server, it makes no difference which one is used as the manager. On the other hand, if one BoA object contains another BoA object, and if the contained one is selected as the manager, it may not be part of the set of objects on the second retrieval due to change in the container BoA object. Thus the retrieval of the manager in such a scenario would be unnecessary. In situations where more than one object in a set is a candidate to be a manager object, MONARCH favors those objects that are closer to the root of the tree.

Finally, if a pattern is ND-ND the distinction between RSt and RDyn objects, mentioned in Section 4.7, becomes important. If the ND-ND pattern can be represented as the RDyn-RSt pattern, then the top-most RDyn object is the manager. If the ND-ND pattern can be represented as the RSt-RDyn pattern, then one of the child RDyn objects is the manager. The rules for selecting one of many RDyn objects as the manager are the same as for the ND-BoA case. If, however, the ND-ND pattern is represented by either the RSt-RSt or the RDyn-RDyn pattern, then the top-most object is selected as the manager.

The top-most object is also designated as the manager when ND objects are not subclassified into RSt and RDyn. The consistency guarantees provided by MONARCH are not undermined by the lack of subclassification of ND objects into RSt and RDyn or erroneous subclassification where rarely changing objects are marked as RDyn and frequently changing objects are marked as RSt. The lack of proper classification of ND objects, however, may cause inefficiency as some validations of the manager object become unnecessary.

A perfect manager is a BoA object because caches must retrieve it on each access and can use that retrieval to validate or receive invalidations for ND objects related to the BoA object. When a BoA object is not available, however, we use an ND object as the manager, even though its validation may be unnecessary. As we mentioned in Section 4.7, we are willing to sacrifice on efficiency in order to provide strong consistency.

### 4.8.3 Content Control Commands and Object Management

The idea behind CCCs is for servers to distill all the information that they have access to into concise and explicit instructions for caches on how to manage each object. Caches examine a CCC command associated with each object and behave accordingly. In this section we discuss the CCC commands used by MONARCH.

#### St, Per, and BoA Objects

The CCC commands that MONARCH assigns to St and Per objects and to BoA objects that have no related ND objects are shown in Table 4.1. MONARCH uses CCC commands for objects with these change characteristics for uniformity reasons. These objects could be managed using mechanisms currently available in the HTTP protocol [31]: Per objects can be assigned explicit expiration time via the `Expires` or the `Cache-Control: max-age` header, and St objects can also be assigned an expiration time that is far into the future; BoA objects with no related ND objects could be marked as uncacheable using the `Cache-Control: no-cache` header.

Table 4.1: CCC Commands Used by MONARCH for St and Per Objects and BoA Objects with no Related ND Objects

Object Change Characteristic	CCC Command
St	Cache
Per	Cache, Validate after TTL or Expires
BoA	Not Cache

**BoA-ND Pattern**

As we have already discussed, when a set of objects can be represented by a BoA-ND pattern, the server selects the top-most BoA container as the manager and groups all ND objects in the set into a volume. The server assigns each ND object a CCC informing its clients that these objects may be cached and will be explicitly invalidated by the server, as shown in Table 4.2. The CCC that the server attaches to the BoA object instructs caches to discard the contents of the object, but keep the volume identifier and volume version that it carries, as discussed in Section 4.8.1. On subsequent retrievals of the BoA object, caches present the cached meta information to the server. Based on that information the server determines if any members of the volume associated with the BoA object have changed. The server piggybacks [49] invalidations for ND objects onto its response for the BoA object. Lack of invalidations in the server response indicates to caches that all cached ND objects in that volume are still fresh.

Table 4.2: CCC Commands Used by MONARCH for the Manager and Managed Objects

Page Pattern	CCC for the	
	Manager Object	Managed ND Objects
BoA-ND	Cache Meta info	Cache until invalidated
ND-BoA	Not Cache	Cache with precondition
ND-ND	Cache, Validate	Cache until invalidated

**ND-BoA Pattern**

When a set of objects is described by a ND-BoA pattern, the server selects one of the BoA objects as the manager and groups all of its ND siblings along with the ND container into a volume. The CCC that the server assigns to the ND container instructs caches that they may cache the object but must satisfy the provided *precondition*—retrieval of the manager BoA object—before re-using the cached copy, as shown in Table 4.2. The CCC that the server assigns to the BoA manager instructs caches to discard the object. The

server provides invalidation for the volume members when the cache returns to satisfy the precondition.

### ND-ND Pattern

When a set of objects is described by an ND-ND pattern, the server assigns different CCCs, depending on which of the two ND objects is chosen as the manager. In situations when the set of objects is described by one of the following three patterns—RSt-RSt, RDyn-RDyn, and RDyn-RSt—the server assigns the top container a CCC that allows caches to cache the object, but requires them to validate that object on every access. In the fourth case—RSt-RDyn—the server instructs caches to cache the top-most container object, but satisfy a precondition—validation of the RDyn manager object—on every access. In all cases, the server groups all ND objects in the set, except for the manager, and invalidates them when caches return to validate the manager object. CCCs for the objects on the page with the ND-ND pattern are shown in the last row of Table 4.2.

The CCC command assigned to the manager object in the ND-ND pattern is the same as the one assigned to Per objects in Table 4.1, except it does not explicitly specify a Time-To-Live or expiration time. Lack of such explicit value indicates validation on every access by default. The CCC command assigned to the managed ND objects is actually the same as the one assigned to St objects in Table 4.1, except invalidation can never occur for St objects. When a page contains no ND objects, including cases when the page is described by the BoA-BoA pattern, all objects on the page are assigned CCC commands shown in Table 4.1.

#### 4.8.4 Shared Objects

At the core of the MONARCH approach is the composition relationship between a BoA or an ND object and a set of ND objects. Exploiting such a relationship fails to guarantee strong consistency, however, if objects are shared between composite objects. Consider the

following case of consistency failure. Two distinct objects  $CO1$  and  $CO2$  contain the same ND object  $EO$ . A cache retrieves and caches object  $CO1$  and its volume. Subsequently, object  $EO$  changes at the server. After that the cache retrieves object  $CO2$  and all members of its volume except for  $EO$ , which the cache already has. The cache uses its stale copy of  $EO$ . The reason the server never invalidated  $EO$  is because the cache never indicated to the server that it had a cached copy of that object since the cache was retrieving a page that it has never seen before.

To solve this problem, we introduce a *shared*, or *global*, volume containing all ND objects at a server that are shared by more than one composite object. The server might identify such objects by using a common directory for their storage. Volumes that incorporate objects contained in only one composite object are called *local*. When the server sends the manager object to the cache, it includes meta information for both local and global volumes. This approach provides efficient management of shared objects while ensuring strong consistency when they are cached.

We expect that all objects at a site either belong to one of the local volumes or to the shared volume. However, it is possible that an object may be transferred from a local volume to the shared volume or from the shared volume to one of the local volumes. For example, suppose  $EO$  is originally contained only in  $CO1$ . The cache retrieves and caches objects  $CO1$  and  $EO$ . Later on,  $CO2$  is updated such that it now embeds  $EO$ , thereby transferring  $EO$  from the local volume associated with  $CO1$  to the shared volume. If  $EO$  changes, and if the cache revisits the server to obtain  $CO2$ , the cache will serve stale  $EO$  to its client. If objects at a site do migrate between local volumes and the global volume then the server can always provide global volume version to caches. When the cache retrieves  $CO2$  from the server, it presents the server with the version of the global volume that it obtained from the server earlier. The server determines that  $EO$  has been added to the global volume since the cache's previous visit and notifies the cache. The notification includes the current version of  $EO$ . The cache determines that its copy of  $EO$  is stale and

---

obtains the new version. To simplify matters, all objects, especially those that are expected to change, should be separated into local and global. If an object keeps moving between local and global volumes, it should always be treated as global.

## 4.9 Summary

In this chapter, we motivated the problem of deterministic object management in a distributed system using a realistic example taken from the Web. We then defined the problem addressed in this dissertation. Our foremost objective is to eliminate consistency failures, while maintaining no per-client state at the server. In addition to that, our goal is to reduce the amount of consistency maintenance traffic generated by servers and caches.

This chapter also described the approach that we take in this dissertation to address the problem of providing strong consistency for objects in a distributed system. We defined object change characteristics, discussed different types of relationships that objects in a distributed system may have and suggested ways to exploit these relationships for object management. We also showed how we use object relationships in conjunction with object change characteristics to ensure strong cache consistency in a distributed system. In the next two chapters, we validate our approach via implementation and evaluation using simulations.



## Chapter 5

# Prototype Design and Implementation

We have designed and built a prototype system to investigate whether it is feasible to implement the MONARCH approach to object management. The system consists of three components: the MONARCH Content Management System (MCMS), MONARCH Web Server (MWS), and MONARCH Proxy Server (MPS), as shown in Figure 5.1. We describe the design of each component below. All components are implemented as a set of Object-Oriented (OO) Perl modules with the MWS and MPS components running in the Apache address space under `mod_perl`. Apache was chosen over Squid because it has better documentation, making it easier to install, configure, and grasp. Another advantage of the Apache Web server is the availability of the `mod_perl` module, which embeds the Perl interpreter into the address space of the Web server. The combination of Perl and Apache creates a flexible environment for rapid prototyping.

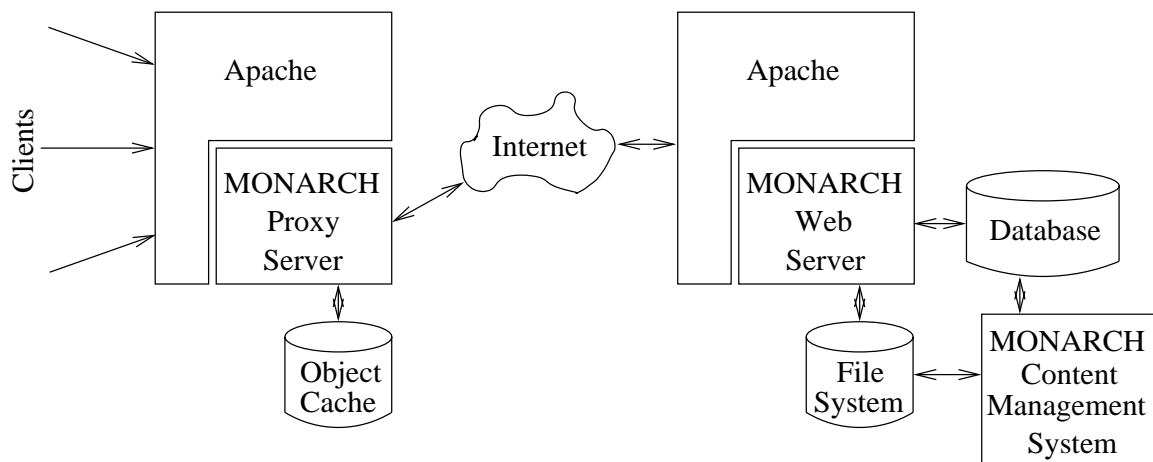


Figure 5.1: Architecture of the Prototype System

## 5.1 Content and Its Organization

In this section we describe how Web pages are constructed, where they are stored, how objects are marked with their change characteristics, and how and where change characteristics and CCC commands are stored in the prototype system.

To make our discussion more concrete, consider a Web page, `index.html`, with a few embedded objects, such as `logo.jpeg`, `navmenu.cmp`, and `top_story.cmp`. All objects are stored in files residing in the directory `test` accessible to the Web server. Embedded objects include images and components. Images are embedded using the standard `img` tag defined in HTML, and components are embedded (included) using the `GI` (“Generic Include”) tag that we introduced. To simplify our page parser, we represented other possible embedded objects, such as CSS and JavaScript code, using images. Components embedded using `GI` tags are stored in separate files with the `cmp` extensions. The specific extension for components is a matter of convenience, rather than a requirement. Components may contain arbitrary HTML content, and may embed other objects, such as images and components.

Once objects are created, it is necessary to tag them with appropriate change characteristics. With industrial-strength content management and publishing systems, it should be possible to use a graphical interface to perform that task. In our prototype system the task

is carried out using a text editor. For each object, there is an identically named file in a special subdirectory `.chgchar`, containing the change characteristic of the respective object. For example, for the object `top_story.cmp`, residing in the directory `test`, there is a file named `top_story.cmp`, residing in directory `test/.chgchar` that contains the following text: `CHCHAR = RDyn`, indicating that `top_story.cmp` has the `RDyn` change characteristic. If a given object does not have a matching file in subdirectory `.chgchar`, it is assumed that that object has the `RSt` change characteristic. Having a *default* change characteristic allows one to minimize the number of *explicitly* tagged objects at a site. Only those objects that are known not to be `RSt` must be tagged. One might also decide to use the file system structure to aide in assigning change characteristics. For example, all objects in the `images` directory are `RSt`, and all objects in the `papers-in-pdf` directory are `St`, etc. In our implementation, in addition to the change characteristic itself, the file in the `.chgchar` subdirectory may also contain extra information, such as the TTL or absolute expiration time for periodic objects.

In addition to the file describing the change characteristic, each object also has another file associated with it describing the CCC assigned to the object. The CCC file is named after the object file, and is stored in the directory `.meta`. For example, the CCC assigned to object `top_story.cmp` is stored in the `.meta/top_story.cmp` file. Similar to having a default change characteristic, it is possible to have a default CCC command. However, given that all CCC commands, and files they are stored in, are created automatically by software, we did not implement a default CCC command.

### 5.1.1 Database

Information about all objects at a site is added to a relational database. The database consists of the following tables:

- The `change_char` table stores all possible change characteristics: `St`, `RSt`, `RDyn`, `Per`, and `BoA`. Each change characteristic is associated with a unique identifier, which is

used by other tables to refer to change characteristics.

- The `objects` table stores names and change characteristics of all objects at a site. Change characteristics are stored as identifiers taken from the `change_char` table. Each object is also associated with its own unique identifier.
- The `object_revisions` table stores object revisions. Each entry in this table associates an object's identifier with that object's version. Currently, an object's version is implemented as the object's last modification time.
- The `volume_membership_revisions` table stores volume membership revisions. Each entry in this table contains the volume identifier, volume version, lists of objects added to and deleted from the previous version of the volume, and a list of current volume members. Initially, the first two fields for each volume start out simply as the identifier and version of the top-most container object. We provide more details on volume versioning in Section 5.3.1. Storing a list of added and deleted members, in addition to the complete list of volume members, for each volume revision, is redundant. Initially, we stored only differences, or deltas, between successive volume revisions, which required us to determine the complete list of volume members upon object updates and upon client requests. We also realized that computing a difference between two volume revisions required us to examine all container objects in the volume and not just the modified objects because different containers may embed the same objects. To simplify our implementation, we switched to storing the complete list of volume members for each volume revision. The lists of added and deleted members are not currently used by the prototype system and the respective columns could be safely deleted from the table. We mention them here to emphasize the evolution of the system design process and the fact that deltas were considered.

The database itself is implemented using the popular and open-source MySQL software. Similar to Apache, MySQL is widely used, robust, and has great on-line documentation

and published books available. In addition, there exist freely available Perl modules that provide an easy to use interface to the MySQL database. MySQL proved to be easy to learn, install, and use. All database operations, such as adding an object or volume revision to the database, retrieving information about a particular volume from the database, etc. are carried out via a set of custom object-oriented accessor methods that we wrote. Once developed and debugged, all their functionality is hidden in a Perl object that is used by the caller scripts.

## 5.2 MONARCH Content Management System

Once the desired objects are created or updated and are copied into a directory readable by the Web server, they must be processed. Processing could be triggered by the script that copies objects from a staging server to the production server. It could also be initiated from a shell script from which the text editor used to create/update objects is called. In this section we describe the details of how such processing is done by the MONARCH Content Management System (MCMS).

We implemented MCMS as a collection of OO Perl modules. The top-level Perl script is fairly short since it simply calls a few methods implemented by the `Site` object (`Site.pm` Perl module). A number of site-wide configuration options can be specified in a file that is passed to the `Site`'s constructor. The top-level script takes a list of objects to be updated as its arguments, or constructs such a list itself by reading all files with extensions `jpeg`, `cmp`, and `html` from the default directory, and passes that list to the `update_objs` method of the `Site` object.

The core of the MCMS system is the Web Object Cache Compiler (WOCC) that analyzes the relationships between objects in conjunction with the object change characteristics and compiles them into Cache Control Commands (CCCs).

### 5.2.1 Object Processing

In this section we provide a high-level overview of how object processing in the `update_objs` method is done. In later sections we provide more details of the algorithm.

Given a list of objects to be created or updated, the `update_objs` method performs the following steps:

1. It first constructs two lists: one containing only root (top-most parent) objects and the other one containing all other (non-root) objects. Currently, all objects with the `html` extension fall in the former category.
2. For each root object, the method calls the `_preprocess_volume` method, which creates a new `Object` object and adds it to an internal hash table. The `_preprocess_volume` method also parses the root object, constructs a list of objects embedded in the root object, and then calls itself recursively on each embedded object. Thus, after the `_preprocess_volume` method is done with one root object, the internal hash table contains all objects that are required to render the top-level object (Web page) in a Web browser.
3. The next step is to process each non-root object that was passed to the `update_objs` method using the same recursive `_preprocess_volume` method. Those non-root objects that were already processed during the previous step are skipped.
4. For each non-root object it is necessary to determine which root objects embed them. This information is obtained from the `volume_membership_revisions` table of the database. Each of the root objects (there could be more than one for each non-root object) is also processed using `_preprocess_volume` and is added to the internal hash table.
5. At this point we have a hash table that contains all the objects that must be updated in the database. The next step involves finding a manager for each of the volumes.

Method `_find_manager` is invoked on each of the root objects to carry out that task, followed by another method, either `_assign_cccs` or `_assign_standalone_cccs`, that assigns CCCs to all members of the volume. For those objects that are not part of any volume, we also assign CCCs, using the `_assign_standalone_cccs` method.

6. Once the manager is found and the appropriate CCCs are determined, the objects are tagged with their respective CCCs. Version information for each object is added to the database. For each root object, we also construct a full list of embedded objects and add volume membership information to the database.

### 5.2.2 Volumes

In the current implementation, MONARCH groups all objects composing a Web page into a volume. We faced a number of design decisions regarding volume representation and describe them in this section. It is helpful to envision a Web page represented as a tree, with the container object represented as the root of the tree, as shown in Figure 4.7. Children of the root node may be leaf nodes or parent nodes containing their own children.

#### Volume Revisions

We can identify two types of content changes: changes that do not affect volume membership and changes that do affect volume membership. The question is which changes should affect the volume version. One possibility is to update the volume version when any of the objects in the volume changes, even if no objects are removed from or added to the volume. Another possibility is to create a new volume version only when volume membership changes. We chose the latter option because we wanted to decouple the two types of changes and be able to differentiate between them. One could envision a situation where volume membership remains unchanged while individual volume members undergo numerous updates. In our implementation, there is no need to create new volume revisions, and, therefore, the amount of state maintained does not increase.

### Subvolumes

Another question is what information should be stored for each volume. We certainly need to assign a name or an identifier and version information to each volume. We also need to keep track of which objects are the members of each volume. We could compile and store a list of *all* members of a volume, or we could compile and store a list of only the *children* nodes. The latter approach uses the notion of *subvolumes*, where we store a list of children for each node in the tree, and recursively assemble the list of all members when required.

Intuitively, it may seem that the subvolume approach is better than storing a full list of volume members (it did seem that way to us at first; we even made an attempt to implement subvolumes). The subvolume approach, however, has a number of drawbacks. It does not offer any space savings over its alternative, and may even require a bit more space. A more serious problem with the subvolume approach is how to maintain volume versions. When one object in a volume changes, we can easily identify the affected subvolume and update its version. Do we then recursively find all encapsulating subvolumes and update their versions as well? Another issue is how to handle invalidations. When a client presents the server with the name and version of the previously retrieved volume, the server needs to determine which objects need to be invalidated. Finding all such objects requires the server to construct the list of volume members by combining lists obtained for each subvolume. Two or more subvolumes within the same volume may contain some of the same objects, in which case the server needs to eliminate duplicates. The server needs to do all this processing as part of servicing the client's request, increasing the response time. To summarize, while at first subvolumes seemed a good design decision, further thinking and prototyping convinced us to compile and store a list of all volume members once, when objects are added to the database, simplifying implementation and eliminating extra work that the server has to do while serving requests.



### Volume Deltas

When a page is updated in a way that some objects are removed from or added to the page, volume membership changes, and MCMS must create a new volume revision. A question arises whether MCMS should store a list of volume members for the new revision or a difference (delta) between the previous and the current lists of volume members, to reduce the amount of space required. Note that this question applies even if MCMS uses subvolumes.

Early on in the design of the MCMS system, we decided to store volume membership deltas. Each new revision of a volume contains a list of objects deleted from the previous version of the volume and a list of objects added to the volume. Implementation complexity, and the fact that the server needs to re-construct older volume versions while processing client requests, resulted in our decision to switch to storing a complete list of volume members for each volume revision. The table columns for storing added and deleted volume members are still present in the database, but they are not currently used by the prototype system and could be safely deleted from the database.

### Volume Version

One issue we still have not discussed is the assignment of versions to volumes. Object versions are currently implemented as a last modification time, as discussed in Section 5.1.1. What should be used as a volume version?

Since the volume name is the name of the root node, we could think of using the version of the root node as the volume version. However, since object changes that do not result in objects being added or deleted do not affect volume membership, the version of the root node may change independently of the volume version. Furthermore, volume membership changes that are due to changes in objects other than the root object, need to be taken into account.

Another option is to use the version of the object that is responsible for the volume

membership change as the volume version. Initially, the volume version is the version of the root node in the tree, and subsequently it takes on versions from other objects in the volume. However, since the server does not use the notion of subvolumes, as discussed above, determining which object in the volume is responsible for volume membership changes requires the server to do extra work. In our implementation, volume version starts out as the version of the root node and is incremented by one for each volume revision.

### 5.2.3 Web Object Cache Compiler

The Web Object Cache Compiler is responsible for compiling the relationships between objects in conjunction with object change characteristics into CCCs. A CCC that WOCC assigns to an object depends on the change characteristic of the object itself and on the change characteristics of the related objects. We first describe how WOCC determines the overall management strategy for a volume and then discuss the rules that WOCC uses to assign CCCs to objects in the volume.

#### Finding a Manager

WOCC exploits the relationships between objects composing a page using the notion of a manager object. Currently, only objects that have either BoA or RDyn change characteristics can be managers (our approach also allows RSt objects to be managers, when neither BoA nor RDyn objects are available, but our current implementation needs to be extended to support that). BoA objects must be retrieved on every access and are thus perfect candidates for helping to manage other objects. RDyn objects are assumed to change frequently and thus represent the second best choice.

Each volume is represented as a tree, with the root object being the top-most node. The `_find_manager` method uses a generic tree traversal method `_traverse_volume`, which traverses any given volume in depth-first fashion and applies all functions passed to it as arguments to each node of the tree. Upon encountering the next node, the `_find_manager`

method attempts to determine if that node qualifies as a good candidate for being chosen as the manager object using the following algorithm:

- No object has been picked as the manager so far.
- The current object has a more *dominant* change characteristic than the current manager. BoA is a more dominant change characteristic than RDyn, which, in turn, is more dominant than RSt. St and Per objects cannot be manager objects.
- The current object has the same change characteristic as the current manager, but it is located higher up in the tree than the current manager.

In addition to determining which object in a volume should be chosen as the manager, it is also important to analyze the mix of change characteristics in the volume in order to determine whether using a manager is actually helpful. Consider a volume with one BoA and multiple St objects. In this case, we can find a manager using the algorithm above, but we have no objects that need to be managed. The following set of rules is used to decide if there is a need for a manager:

- A volume contains a single ND (RSt or RDyn) object and at least one BoA object
- A volume contains more than one ND object

### Assigning CCCs

If WOCC determines a manager is required, and which object is the manager for a given volume, it has all the information to tag each object in the volume with the appropriate CCC. WOCC uses a pre-defined set of rules to assign CCCs to objects. We describe these rules below.

St and Per objects have deterministic change characteristics and can be managed without any assistance from other objects. One approach to managing them in MONARCH is to simply use mechanisms already present in the HTTP protocol. MONARCH can use **Expires**

or `Cache-Control: max-age` headers to assign such objects the proper expiration time. We chose to assign CCCs even to `St` and `Per` objects, for uniformity sake. The rules for assigning CCCs to `St` and `Per` objects are shown in Table 5.1.

Table 5.1: Mapping of Change Characteristics to CCCs for `St` and `Per` Objects

Change Characteristic	CCC
<code>St</code>	C (Cache)
<code>Per</code>	CV, Expires (Cache, Validate when expires)

The rules for mapping change characteristics of non-manager and manager objects to CCCs are shown in Table 5.2 and Table 5.3 respectively.

Table 5.2: Mapping of Change Characteristics to CCCs for Non-Manager Objects

Change Characteristic	CCC
<code>BoA</code>	NC (Not Cache)
<code>ND (RDyn and RSt)</code>	C (Cache)

Table 5.3: Mapping of Change Characteristics to CCCs for Manager Objects

Change Characteristic	CCC
<code>BoA (top-most container)</code>	CM (Cache Meta information only)
<code>BoA</code>	NC (Not Cache)
<code>ND (RDyn or RSt)</code>	CV, TTL=0 (Cache, Validate each time)

As Table 5.3 shows, there are two distinct rules for scenarios when the manager is a `BoA` object. In general, since `BoA` objects change on every access, they should not be cached by client caches. The second row in Table 5.3 shows a CCC command for such a general case. The first row in Table 5.3 shows a special case, when the manager is the top-most container object with a `BoA` change characteristic. The server uses a CCC that instructs client caches to store only meta information, such as volume name and version, associated with that object. On subsequent accesses from their client, caches relate that

meta information to the server.

In cases where the top-most object is not a manager, but the volume does have a manager, the CCC for the top-most object is enhanced with a *precondition*. The precondition indicates to the cache that before assessing the freshness of the container object, it must perform some action. In our prototype system that action is the retrieval of the manager object from the server. The name of the manager object is provided in the precondition.

### CCC Syntax

The syntax of the CCC commands is described by the grammar shown in Figure 5.2. We use the augmented Backus-Naur Form described in RFC 2616 [31] for describing our grammar.

```

CCC = 'cmd=' cmd-C | cmd-NC | cmd-CM | cmd-CV | cmd-INV
cmd-C = 'C' ['; pre=' token]
cmd-NC = 'NC'
cmd-CM = 'CM'
cmd-CV = 'CV' [';' 'ttl | expires' '=' 1*DIGIT]
cmd-INV = 'INV; objs=' '<'> invalidation-list '>'
invalidation-list = invalidated-object [';' *invalidated-object]
invalidated-object = token '^' 1*DIGIT

```

Figure 5.2: Grammar for the CCC Commands

## 5.3 MONARCH Web Server

We implemented the MONARCH Web Server (MWS) as a plug-in for the Apache Web server. Apache can be configured to hand off all or certain requests to the MONARCH plug-in. The main responsibility of the MWS is to communicate with the MCMS and obtain object and volume version and volume invalidation information that is included in the server response. MWS maintains no per-client state, relies on its clients to provide volume information on subsequent visits, and provides its clients with targeted invalidations that are likely to be immediately useful.

We introduce a number of new HTTP headers that the MONARCH server uses to provide information to its clients. The server uses the `Version` header to inform clients of the object version. The server uses the `VName` and `VVersion` headers to report volume name and volume version information respectively. The server sends CCC commands to its clients using the `CCC` header. Clients also use the first three headers to report previously obtained information to servers.

For every request the server first connects to the database, obtains the current version of the requested object, and attaches that version to the outgoing HTTP response headers using the `Version` header. If a client is requesting the top-most container object that it had cached from the previous retrieval or is requesting a precondition object, the client's request contains meta information for the volume that the client is currently interested in. In that case, the server attempts to obtain the current version number for that volume from the database. The server combines the volume version with the current time and adds the resulting value to the outgoing HTTP response headers using the `VVersion` header. The server also adds a volume name (which is currently the same as the name of the top-most container) to the outgoing HTTP response headers using the `VName` header. We explain the reason for combining volume version with the current time in Section 5.3.1.

The server then examines the incoming HTTP request headers to see whether the client provided volume name and volume version, using the `VName` and `VVersion` headers respectively. If so, the server consults the database and attempts to invalidate objects in the given volume. The server uses the volume version provided by the client to determine which objects were part of the volume at the time of the client's previous request. The server uses the time of the client's previous request (also available in the `VVersion` header) to determine if any of the identified objects have changed. The server constructs an invalidation list consisting of changed objects along with their current version numbers. The server then builds an invalidation CCC command and adds it to the outgoing HTTP response headers using the `CCC` header.

For each requested object, the server also reads the CCC command from a corresponding file and adds it to the outgoing HTTP response using the CCC header. The server response may contain more than one CCC response header.

As an example, consider a request for a sample object `index.html` which results in the following HTTP response headers (some standard response headers, such as `Server` and `Connection`, were removed for readability):

```
HTTP/1.1 200 OK
Date: Thu, 14 Mar 2002 03:06:01 GMT
Version: 1008104231
VVersion: 1008104231-1016075161
VName: index.html
CCC: cmd=C; pre=username.cmp
Content-Length: 1110
Content-Type: text/html
```

The sample response headers indicate that the current version of object `index.html` is 1008104231. In the current implementation, this is simply last modification time of the file, in the UNIX format. Volume version is constructed from the version of the top-most container and the current time. Since our sample object has the `html` extension, the server treats it as the top-most container and associates a volume with it. The name of the volume is the same as the name of the object: `index.html`. The server also found the CCC command that WOCC assigned to this object. The CCC command indicates to client caches that the `index.html` object itself can be cached, but before the cache is allowed to reuse the cached copy, it must first satisfy the attached precondition: validate the freshness of object `username.cmp`.

### 5.3.1 Volume Invalidation

In the current implementation, MONARCH groups all objects composing a Web page into a volume. The primary reason for grouping objects into volumes is to limit the amount of invalidation information that the server reports to its clients. In this section we examine

volume invalidation.

Suppose a client request for object  $O$  indicates that the client has previously obtained version  $i$  of volume  $V$ . As part of serving this request, the server invalidates those objects in volume  $V$  that have been updated since the client's previous retrieval of volume  $V$  (the client may have these objects cached or may have already evicted some or all of them from its cache). The server may choose to invalidate all objects that composed volume  $V$  at the time of the client's previous visit, or only those that are still part of volume  $V$ . Volume invalidation is thus a process that involves the following steps:

1. The server creates a set of objects that composed volume  $V$  at the time when the client obtained version  $i$  of that volume (call it **Set1**).
2. The server creates a set of objects that are currently composing volume  $V$  (call it **Set2**) and computes the intersection of the two sets **Set1** and **Set2**, resulting in **Set3**.
3. The server determines the current version for each object in **Set3**.
4. The server invalidates those objects in **Set3** whose current version is newer than the time of the client's previous retrieval of volume  $V$ .

In our implementation, obtaining information for Step 1 is straightforward. Volume revisions are stored in the database, as described in Section 5.1.1. When membership of a volume changes, MCMS updates the version of that volume and adds a new revision into the database. The server simply takes the volume name and version,  $V$  and  $i$ , supplied by the client and looks up members of that volume for that version in the database. Similarly, the server can complete Step 2 by retrieving members of the latest version of volume  $V$  from the database and computing the intersection of the two sets. The server can also easily complete Step 3 by consulting the database table that stores object revisions.

The most difficult part of the volume invalidation process is determining versions of volume  $V$  members at the time when the client obtained that volume (Step 4). Volume



versions are not time based in our implementation, as discussed in Section 5.2.2. Suppose we make volume versions time based upon the time when volume membership changes. One could argue that given time-based volume versions, and since MCMS updates versions when objects change, it is possible to determine which volume members have changed by simply comparing the volume version, supplied by the client, with the current versions of volume members. Those objects that are still members of the volume in question and have higher version numbers than the volume version supplied by the client, must have changed since the client retrieved them. Such an argument has a drawback, however.

To understand the issue with the argument above, consider the following scenario. Version  $i$  of volume  $V$  is created at time  $t_i$ . Two objects from volume  $V$ ,  $O_1$  and  $O_2$ , are updated at times  $t_j$  and  $t_l$  respectively. Updates to both objects involve only content of the objects themselves and do not affect volume membership (no embedded objects are added or removed). A client first retrieves volume  $V$  at time  $t_k$  and then returns and fetches the same volume again at time  $t_m$ . On the second visit, the client validates or retrieves the manager object, presenting the server with the version  $i$  of volume  $V$ , and expects the server to provide invalidations. All the times are related to each other as follows  $t_i < t_j < t_k < t_l < t_m$ . The server determines that since versions of both objects  $O_1$  and  $O_2$  are higher than the value of  $i$ , these objects must have been modified after the client's previous visit and must be invalidated. In reality, however, object  $O_1$  was modified *before* the client retrieved it and therefore should not be invalidated.

In order to invalidate only those objects that actually changed since the client's previous visit, the server must be able to determine version numbers of objects that composed the volume retrieved by a client at the time of the previous retrieval. In our implementation, MWS combines the volume version obtained from the database with the time of the client's request and uses that combined value as the volume version given to the client. On the client's subsequent visit, the server knows when the previous request took place and can use the time of the previous request to determine which objects must be invalidated.

## 5.4 MONARCH Proxy Server

Similarly to the MONARCH Web Server, the MONARCH Proxy Server (MPS) is also implemented as a plug-in for the Apache Web server using Object-Oriented Perl modules running in the Apache address space under `mod_perl`. Upon receiving a client request, the proxy attempts to find the requested object, either retrieving it from its cache or fetching it from the origin server. MPS always manages objects using the MONARCH object management policy, but falls back to a heuristic policy if the server does not provide CCC commands.

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

### 5.4.1 Request Handling

When the proxy receives a request from its client for an object, it attempts to find a fresh copy of that object. MPS first checks its local cache to see if the object is available locally. If not, MPS sends a request to the origin server. If the object is available in the local cache, the following are the possible outcomes:

- The object is available and is fresh,
- The object is available but is stale,
- The object is available but its freshness is unknown because the object has a precondition associated with it, or

- Only meta information for this object is available, the object itself is not (this case occurs only for BoA containers, as discussed in Section 5.2.3).

In the last case, the proxy appends the object's meta information to the request that it sends to the server. That meta information indicates the version of the object and the name and version of the volume to the server.

### 5.4.2 Determining the Freshness of Cached Objects

The cache first checks whether the object has an `X-Cache-Expires` header. That header is assigned to certain objects when they are cached. If the header is present, its value is compared with the current time. If the current time is greater than or equal to the expiration time, the cache prepares a list of headers that the proxy can use to validate the object with the server. Otherwise, the object is considered fresh.

If the `X-Cache-Expires` header is absent, the cache checks whether the object has any CCC headers. If a precondition is present, the cache instructs the proxy to satisfy the precondition first. If the object has the “C” CCC command, which indicates that the object can be cached until the server explicitly invalidates it, the cache considers the object fresh.

### 5.4.3 Satisfying a Precondition

In the current implementation, there is only one type of precondition available. A precondition instructs the cache to retrieve or validate an object that is specified in the precondition. While instructing the proxy server to satisfy a precondition, the cache passes the proxy server the name and version of the volume for which this precondition is required. The proxy server includes that information in the request that it sends to the origin server. The origin server then uses that information to invalidate other members of that volume that may have changed since the last time the proxy retrieved them.

Retrieved precondition objects are usually uncacheable. In the current implementation, however, MPS caches the retrieved precondition objects (even if they have the “NC” CCC

command) and assigns them a usage count of one. MPS adds the `X-Cache-Usage-Count` header to each cached precondition object to store the usage count. We expect that precondition objects will be used almost immediately by the proxy. When MPS actually uses a cached precondition object, it decrements the usage count and removes the object from the cache.

#### 5.4.4 Object Caching

The caching proxy server first examines all CCC headers and caches the object using the MONARCH approach. With each cached object (even if only meta information is cached instead of the entire object) the proxy stores the URL that it used to fetch that object using the `X-Cache-Base` header.

#### Caching Using the MONARCH Approach

The cache handles cache-related CCC commands as follows:

- **NC**—Neither the object itself nor its meta information is cached. (As described in Section 5.4.3, objects with this CCC command may actually be cached for a short period of time if they are precondition objects).
- **CM**—Only the HTTP response headers are cached, not the response body. The header includes the volume name and object and volume versions.
- **C**—The entire server response is cached.
- **CV**—The cache first determines the object expiration time and then stores it using the `X-Cache-Expires` header along with the object. The expiration time is determined as follows. If the CCC command provides an explicit expiration time, then that is what the cache uses. Otherwise, if the CCC command provides a TTL value, the cache computes expiration time by adding that TTL value to the current time.

### Caching Using the Standard Approach

When either no cache-control CCC commands are present or more than one is present, making it difficult to decide which one should be used, the cache resorts to using the standard caching mechanism. The cache first checks whether object has any `Cache-Control` headers. If either `Cache-Control: no-store` or `Cache-Control: private` is present, the object is not cached. The current implementation ignores the `Pragma: no-cache` response header since HTTP/1.1 specification [31] does not specify that header as valid in the response.

Otherwise, the cache checks whether the object has the `Expires` and/or `Last-Modified` headers. If neither is present, the object is not cached. The object is also not cached if the value of its `Expires` header indicates that the object has already expired. In all other cases, the cache adds the `X-Cache-Expires` header to the object and caches the object.

When only the `Last-Modified` header is present, the expiration time is determined as follows. The object's last modification time is subtracted from the current time to produce the object's age. A percentage is then taken from the computed age and added to the current time. The percentage is configurable (it can be specified as a parameter to the cache constructor); the default value is 10%.

#### 5.4.5 Statistics Gathered by the MONARCH Proxy Server

We instrumented MPS to keep track of its activity. The following is a list of statistics gathered by MPS. We provide names and descriptions of all counters:

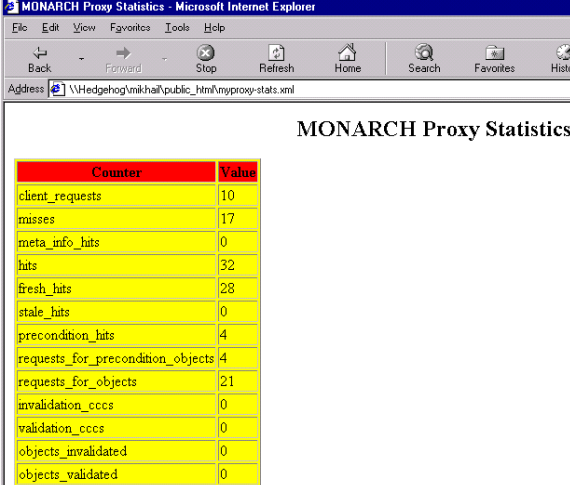
- `client_requests`—number of client requests that MPS has received.
- `misses`—number of times MPS could not find a required object in its cache. This counter takes into account objects explicitly requested by a client and objects that MPS needed to perform page assembly.
- `meta_info_hits`—number of times MPS has found only meta information for the required object in its local cache (not the body of the object).

- `fresh_hits`—number of times MPS has found an entire object in its cache and the object was fresh.
- `stale_hits`— number of times MPS has found an entire object in its cache and the object turned out to be stale.
- `precondition_hits`— number of times MPS has found an object in its cache and that object had a precondition associated with it.
- `hits`—number of times MPS has found the required object in its cache. This counter takes into account all four types of hits given above.
- `requests_for_precondition_objects`— number of requests for precondition objects that MPS has sent to the origin server.
- `requests_for_objects`— total number of requests that MPS has sent to the origin server (superset of the previous counter).
- `invalidation_cccs`—number on invalidation CCCs that MPS has received.
- `objects_invalidated`—number of invalidated objects.
- `validation_cccs` and `objects_validated`— number of validation CCCs that MPS has received and the number of validated objects respectively. These two counters were introduced during the initial development of the prototype system. Currently the system does not implement the object validation mechanism and these two counters are always zero.

In order to conveniently initialize the counters and to view the statistics collected by the proxy server, we implemented two special URLs that the proxy server understands:

- `/myproxy-stats-init`—accessing this URL zeros out all the counters.
- `/myproxy-stats`—accessing this URL results in all of the counters being returned.

By default, both URLs above return counter data formatted in XML. XML formatting is convenient in cases where counter data must be subsequently processed by an automated client and perhaps re-formatted for the final presentation. By default, MPS also associates an XML-encoded style sheet with the returned XML document. The style sheet specifies a transformation of the XML document, called an XSLT transformation, allowing an XSLT-compliant client to convert XML counter data into an HTML page on the fly. Figure 5.3 shows how one such XSLT-compliant Web browser, Microsoft Internet Explorer, renders XML counter data by applying a style sheet to it.



Counter	Value
client_requests	10
misses	17
meta_info_hits	0
hits	32
fresh_hits	28
stale_hits	0
precondition_hits	4
requests_for_precondition_objects	4
requests_for_objects	21
invalidation_cccs	0
validation_cccs	0
objects_invalidated	0
objects_validated	0

Figure 5.3: Internet Explorer's Rendering of the Counter Data in XML via XSLT Transformation

We have also instrumented MPS to return HTML-formatted counter data on demand for browsers that do not support XSLT transformations, such as Mozilla [67] and Galeon [32]. Appending `?format=html` to both URLs above results in MPS returning HTML-formatted counter data.

## 5.5 Content Assembly

Both the MONARCH Proxy Server and the MONARCH Web Server can perform Content Assembly. Content assembly is the process of replacing each occurrence of the GI tag within

object content with the content of the object specified in the `src` attribute of the tag.

Both MPS and MWS examine the `Accept` HTTP request header in the client's request before serving the requested object. If the client cannot accept the `text/x-dca` content type, then content assembly is performed. Our Content Assembler software parses the main container object and replaces each `GI` tag with the contents of the appropriate object. Each of the objects included in the main container in such a fashion is also parsed and all `GI` tags found in the included objects are also replaced with the appropriate content. The process of replacing `GI` tags is recursive.

Once the page is assembled, it is necessary to adjust the HTTP response headers that are sent to the client. One header that requires adjustment is the `Content-Length` header. The length of the resulting page is larger than the length of the original container. Headers related to cache control also must be adjusted. In the current implementation, all assembled objects are marked as uncacheable. Also, the content type of the assembled object is changed from `text/x-dca` to `text/htm`.

## 5.6 Example

This section provides an example of a real Web page and shows how it is handled within the prototype system. We show the CCC commands that the WOCC compiler assigns to page objects during the compilation process. We show what happens when the user fetches the page using a standard Web browser, accessing the Web via MONARCH Proxy Server. We provide details of the communication between MPS and MWS.

### 5.6.1 The Page

We first describe the Web page itself and show a tree representing all page objects. The following is a fragment of the top-most container object. We removed content that is not essential for this example.



```
<html>
<body>




<table border=2>
<tr>
  <td> 
  <td> <GI src="username.cmp">
  <td> <GI src="ad.cmp">
</tr>

<tr>
  <td> <GI src="navmenu.cmp">
  <td> <GI src="top_story.cmp">
    <GI src="vote.cmp">
  <td> <GI src="top_articles.cmp">
</tr>
</table>

<GI src=policy.cmp>

</body>
</html>
```

The page is using standard HTML markup and GI tags to include components. Components may include other components or embed images. For example, the `top_story.cmp` component embeds an image, as shown below:

```
<table bgcolor=#00FF00 border=0>
<tr align=center>
  <td> Top Story Component <strong>CMP4 top_story.cmp</strong>
    
</table>
```

A tree representing our sample page is shown in Figure 5.4. The top container and embedded objects are represented by rectangles, and components are represented by boxes with rounded corners. We use different geometrical shapes purely to enhance visual presentation, not to differentiate objects semantically.

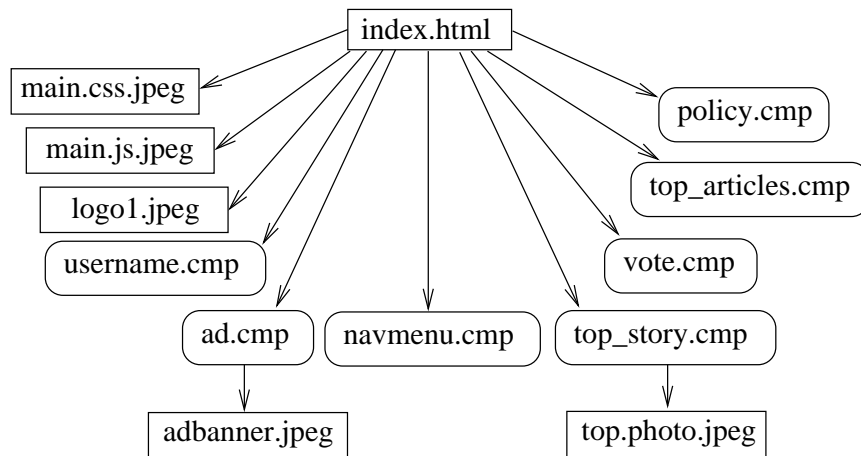


Figure 5.4: Tree Representing the Sample Web Page

### 5.6.2 Object Change Characteristics

All objects composing our sample page are assigned change characteristics. As was discussed earlier, only objects whose change characteristic is *other* than RSt need to be explicitly marked with their proper change characteristic. All other objects are assumed to be RSt by default.

Change characteristics for all objects on the page are shown in Figure 5.5. All nodes in the tree are shown in the same left-to-right order as in Figure 5.4. The tree with object change characteristics is how the WOCC compiler views the page.

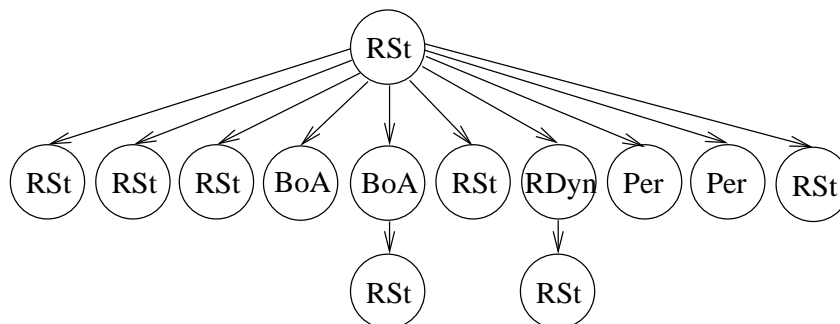


Figure 5.5: Tree Representing Change Characteristics of Objects Composing the Sample Web Page

### 5.6.3 Compilation

The WOCC compiler examines the page and selects one of the three patterns in Figure 4.6 that represents the page. The tree in Figure 5.5 can be described by the ND-BoA pattern because the root of the tree has the RSt change characteristic and one of the objects on the page is BoA. The WOCC compiler then decides on which object on the page will be the manager, and assigns all objects a CCC command. In Figure 5.6 we show the result of the compilation process, with each node in the tree showing the CCC command that WOCC assigned to the respective node. Again, the left-to-right order of the nodes in the tree corresponds to the order of the nodes in the trees above. The manager object is shown with a thicker circle.

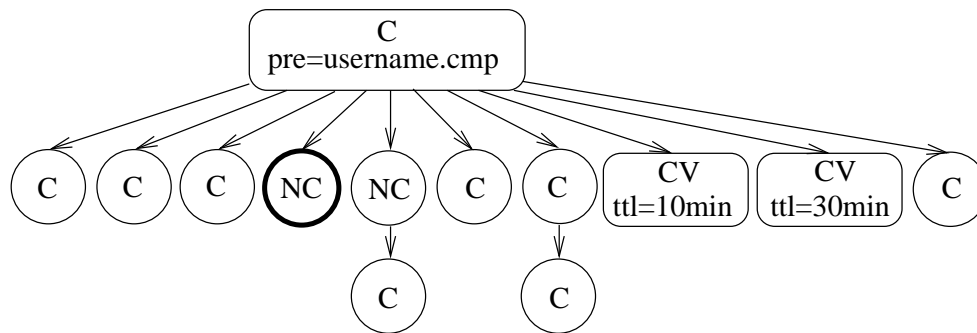


Figure 5.6: Tree Representing the CCC Commands Assigned by WOCC to Objects on the Sample Web Page

As Figure 5.6 shows, most of the objects can be cached until explicitly invalidated by the server. Those objects that have the BoA change characteristic are not cached. Objects with the Per change characteristic can be cached with an explicit expiration time. The container object can be cached, but the cache must satisfy the provided precondition before re-using the cached copy of the container.

### 5.6.4 Page Retrieval

Before retrieving our sample page with a Web browser, we first initialize the proxy server's cache and zero out the proxy's statistics. We then configure a Web browser, such as Mozilla,

Galeon, or Internet Explorer, to access the Web via our proxy server. In our setup, MWS, MPS, and the Galeon Web browser are running on a machine with the Linux operating system. MWS is running on port 80, and MPS is running on port 82.

The screen shot of our sample Web page rendered by Galeon is shown in Figure 5.7. Even though Galeon currently does not implement GI tags, the figure shows that all page components are present because MPS performed page assembly once it recognized that its client is not compliant with our page components. We use different shades of grey to highlight the location of components on the page.

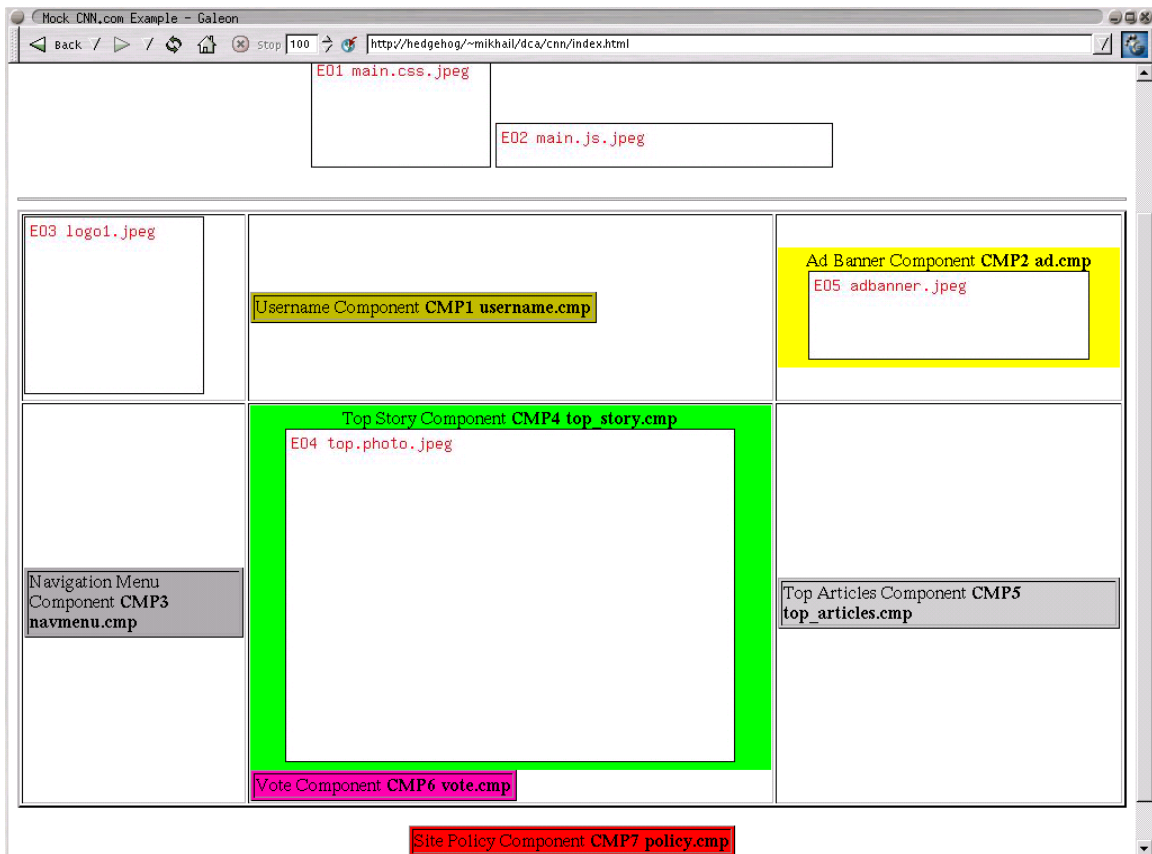
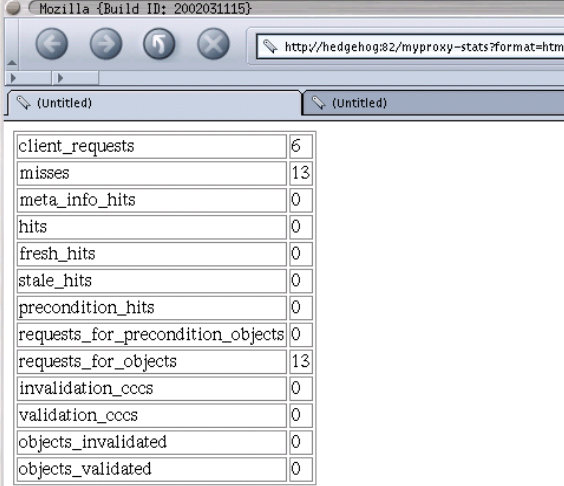


Figure 5.7: Sample Page Rendered by Galeon

After fetching the page, we retrieve the statistics gathered by MPS. Figure 5.8 displays values for all MPS counters. Statistics shows that MPS received six requests, which is exactly the number of objects on the page that the browser is aware of, i.e. all objects

shown with rectangular boxes in Figure 5.4. The total number of objects composing the page is 13 and MPS correctly reports 13 misses and 13 requests that it sent to the origin server.



client_requests	6
misses	13
meta_info_hits	0
hits	0
fresh_hits	0
stale_hits	0
precondition_hits	0
requests_for_precondition_objects	0
requests_for_objects	13
invalidation_cccs	0
validation_cccs	0
objects_invalidated	0
objects_validated	0

Figure 5.8: Proxy Statistics after the First Retrieval

We have also instrumented MPS to log all requests that it sends to MWS and all responses that it receives from MWS. Here we show a sample request/response exchange between the proxy and the origin server. The following is a request for the main container. MPS informs MWS via `Accept` header that it is capable of performing content assembly.

```
GET http://hedgehog/~mikhail/dca/cnn/index.html HTTP/1.0
Accept: text/x-dca
```

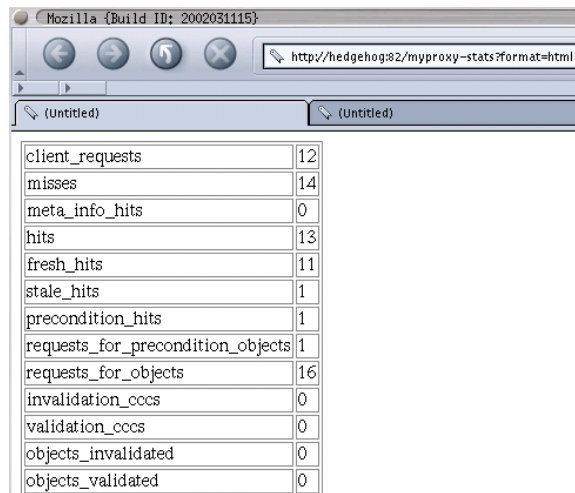
MWS responds as follows:

```
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2002 21:19:56 GMT
Server: Apache/1.3.19 (Unix) (Red-Hat/Linux) mod_perl/1.24_01
Content-Length: 1110
Content-Type: text/x-dca
CCC: cmd=C; pre=username.cmp
Version: 1008104231
VName: index.html
VVersion: 1008104231-1036271997
```

The `VName` and `VVersion` headers provide the name and the version of the volume associated with this container object. The right-hand part of the volume version is the time of the request at the origin server, as discussed in Section 5.3.1. The `CCC` command indicates that the object can be cached, and specifies a precondition that the cache must satisfy before re-using the cached copy of the object. Requests and responses for the other 12 objects are similar, except responses for other objects have different `CCC` commands and do not carry volume information.

### 5.6.5 Second Retrieval of the Page

We now use Galeon once again to fetch the same page via MPS. Some of the objects composing the page are now available from the proxy's cache and should not be fetched from the server. The second retrieval takes place a little over 10 minutes from the first retrieval, and one of the two `Per` objects should have expired in the cache. The statistics that MPS reports after the second retrieval of the page are shown in Figure 5.9.



client_requests	12
misses	14
meta_info_hits	0
hits	13
fresh_hits	11
stale_hits	1
precondition_hits	1
requests_for_precondition_objects	1
requests_for_objects	16
invalidation_cccs	0
validation_cccs	0
objects_invalidated	0
objects_validated	0

Figure 5.9: Proxy Statistics after the Second Retrieval

As before, the client fetched six objects from the proxy, doubling the total number of requests that the proxy served. MPS sent three requests to the origin server on this retrieval: two requests to fetch `BoA` objects, and one request to fetch the expired `Per` object. The

`stale_hits` counter indicates that one object was found in the cache as stale. One of the two BoA objects, namely `username.cmp`, serves as the manager object for the page, and MPS fetched it as a precondition for the container object. The `precondition_hits` counter indicates how many objects MPS found in its cache that had a precondition associated with them. The `fresh_hits` counter shows that 11 objects were found fresh in the cache. That value includes the precondition BoA object that was found in the cache (after it was fetched from the server) with the usage count equal to one. The `hits` counter reports that 13 cache hits occurred. That value was produced as follows: two hits were due to the container page (all cached objects with preconditions currently result in two hits), one hit was due to the stale Per object, and 10 hits were due to other cached objects. MPS reports only one new cache miss, due to the BoA object that was not used as a precondition.

In addition to the new values for all counters, we also show the request for the precondition object that MPS sent to the origin server, and the response that it received. The request is as follows:

```
GET http://hedgehog/~mikhail/dca/cnn/username.cmp HTTP/1.0
Accept: text/x-dca
VName: index.html
VVersion: 1008104231-1036279902
```

MPS sends the request above to the server upon receiving a request from its client for the container object. MPS includes information about the cached volume in its request, as shown above. MWS replies as follows (we show only the HTTP response headers, not the body of the response):

```
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2002 23:46:41 GMT
Server: Apache/1.3.19 (Unix) (Red-Hat/Linux) mod_perl/1.24_01
Content-Length: 110
Content-Type: text/x-dca
CCC: cmd=NC
Version: 996109452
VName: index.html
VVersion: 1008104231-1036280801
```

### 5.6.6 Object Invalidation

We now update one of the objects composing our sample page, namely the RDyn object `top_story.cmp`, and then request the page again using Galeon. Note that the third retrieval takes place a few hours after the second one, and the two Per objects should have already expired in the cache.

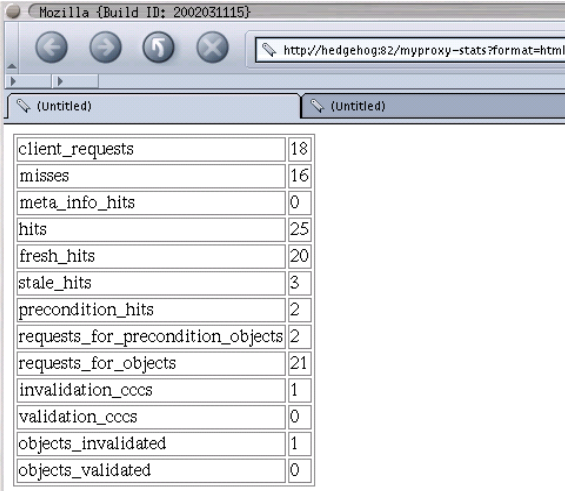
Figure 5.10 shows the state of the counters at the proxy server after the third retrieval of the page. MPS received six more requests from the browser, starting with the request for the container object. MPS found the main container in the cache and proceeded to satisfy the precondition. MPS incremented both precondition-related counters by one. The response from the server for the precondition object carried an invalidation for the updated `top_story.cmp` object, and MPS immediately removed that object from its cache. Invalidation-related counters indicate that there was only one invalidation CCC command and one object was invalidated. The number of cache misses increased by two since the previous request, due to one BoA object not being in the cache and the invalidated object not being in the cache as well. The number of cache hits increased by one less than after the second retrieval of the page. The decrease in the increment is due to the invalidated object being removed from the cache. The number of stale hits increased by two, since two Per objects have expired by the time of the third retrieval of the page. MPS issues a total of five requests to the server on this page retrieval: one request to fetch the precondition object, one request to fetch the missing BoA object, two requests to get the expired Per objects, and one more request to get the invalidated RDyn object.

We once again show the request/response exchange between MPS and MWS. The request for the precondition object is as follows:

```
GET http://hedgehog/~mikhail/dca/cnn/username.cmp HTTP/1.0
Accept: text/x-dca
VName: index.html
VVersion: 1008104231-1036280801
```

The response from MWS is as follows:





client_requests	18
misses	16
meta_info_hits	0
hits	25
fresh_hits	20
stale_hits	3
precondition_hits	2
requests_for_precondition_objects	2
requests_for_objects	21
invalidation_cccs	1
validation_cccs	0
objects_invalidated	1
objects_validated	0

Figure 5.10: Proxy Statistics after the Third Retrieval

```

HTTP/1.1 200 OK
Date: Sun, 03 Nov 2002 17:27:04 GMT
Server: Apache/1.3.19 (Unix) (Red-Hat/Linux) mod_perl/1.24_01
Content-Length: 110
Content-Type: text/x-dca
CCC: cmd=INV; objs="top_story.cmp^1036343784"
CCC: cmd=NC
Version: 996109452
VName: index.html
VVersion: 1008104231-1036344424

```

The response from the server contains two CCC commands, one of which carries object invalidation. In this case, the invalidation is only for one object, `top_story.cmp`. In addition to the name of the invalidated object, the CCC also provides the latest version identifier for that object.

## 5.7 Summary

In this chapter we discussed the design and implementation of all components of the prototype system implementing the MONARCH approach to object management. We pointed out the design issues that we faced while building the system, discussed design alterna-

tives, and described the rationale behind the choices that we made. We provided a detailed example of a real Web page, composed of a few objects, and showed how these objects are managed by the origin and proxy servers. We provided partial traces of the interactions between the proxy and the origin server and presented statistics gathered by the proxy server.

Having a working prototype system provides a validation of the approach, indicating that implementation of the proposed approach is feasible. In addition, the process of building a system allows one to discover and iron out various design issues that may not be obvious otherwise. In this chapter we showed that it is feasible to implement MONARCH and to manage a set of Web objects using object change characteristics and relationships between objects in the set.

## Chapter 6

# Evaluation of MONARCH

In the previous chapter, we showed that it is possible to implement MONARCH and presented design and implementation details of our prototype system. In this chapter, we evaluate the performance of MONARCH and compare it to the performance of existing and proposed cache consistency policies using simulations with the snapshots of content collected from popular Web sites. We also use results from a previous study by Krishnamurthy et al. [51] to estimate the relative differences in user-perceived response time between various policies.

### 6.1 Collection Methodology

In an ideal world, we would be able to obtain current information about the content dynamics and access patterns at busy Web sites. We could then use this information in a trace-driven simulation to evaluate our policy against others. However, most of the available server logs are from small or research-oriented sites, they tend to be dated and incomplete, and may contain only records of client accesses and not of object updates. Obtaining the required information from a single popular site is difficult, and obtaining it from a variety of sites is not realistic.

This section describes an alternate to this ideal, a methodology that we developed

and used for collecting content from Web sites. This content is converted into a format appropriate for a simulator we use to evaluate MONARCH against current and proposed consistency policies. The methodology must address a number of important issues: 1) what is the set of Web sites from which to collect content; 2) what content is collected from each site; and 3) how frequently is it collected. This section addresses all three issues. In the following section, we discuss how accesses to this content are generated.

### 6.1.1 Source Web Sites

The number of existing Web sites is large and is growing continuously. Evaluation of a newly proposed approach that improves some aspect of the Web cannot possibly be carried out on the content of the entire Web. However, a relatively small number of recognizable Web sites are responsible for much of the Web traffic. Thus, to evaluate the usefulness of a new proposal it is only necessary to investigate whether it offers improvements for a sampling of such sites. We also argue that sites with semantically different types of content—news site vs. educational site vs. corporate site—may use different page construction mechanisms and have different content update patterns. It is thus important to ensure that the sites selected for a study cover a range of content characteristics. Given these site selection guidelines, our approach was to pick *recognizable* Web sites that offer semantically different types of content. Table 6.1 lists the eleven Web sites that we selected for this study. The Web sites in our set vary widely in the number of embedded objects that their pages have, in the frequency of updates, and in the use of the HTTP directives related to caching. More information about the dynamics of these sites is provided in subsequent sections.

### 6.1.2 Content to Collect

Having identified a set of sites to study, we needed to decide on the set of objects to study at each site. While we could perform an exhaustive study of a site, we did not want to turn the study into a denial of service attack. Therefore, we focused on collecting the dynamics

Table 6.1: Web Sites Used in Study

Web Site	Type of Site
<a href="http://amazon.com">amazon.com</a>	large e-commerce site
<a href="http://boston.com">boston.com</a>	international/national/local news
<a href="http://cisco.com">cisco.com</a>	corporate site
<a href="http://cnn.com">cnn.com</a>	international/national news site
<a href="http://espn.com">espn.com</a>	sports scores/news
<a href="http://ora.com">ora.com</a>	corporate/publishing site
<a href="http://photo.net">photo.net</a>	graphics heavy discussion site
<a href="http://slashdot.org">slashdot.org</a>	discussion site
<a href="http://usenix.org">usenix.org</a>	technical/scientific association
<a href="http://wpi.edu">wpi.edu</a>	educational site
<a href="http://yahoo.com">yahoo.com</a>	all inclusive portal

for a subset of content at a site.

We used the home page for a site as the starting point for our collection. While it is possible to access a specific page within a site directly, by finding the link using a search engine or receiving a pointer via e-mail, many users “enter” a site and search engines navigate from the home page. Home pages of popular sites are also likely to change frequently as sites add more information, add pointers to new resources, or simply rotate existing content to create the feeling of frequent updates so users return often.

We also wanted to collect a sample of content that could be accessed via the home page. Rather than follow all links on the page or a random subset of links, we took a two-pronged approach. We first identified links on the home page that are always present. We label these links *static*. These links represent aspects of the site that are constant features such as the world news for a news site or admissions information for an academic site. Some users may frequently visit the site because they monitor this aspect of the site.

We also identified links on the page that change over time. We label these links *transient*. These links are of interest to repeat visitors to a site because they do change. They include breaking news stories or new corporate press releases.

While examining Web sites for this study, we realized that home pages of sites known

to be *portals*, such as [www.yahoo.com](http://www.yahoo.com), often provide little content and serve as aggregators for links to sites devoted to different categories of content, such as [finance.yahoo.com](http://finance.yahoo.com) and [photos.yahoo.com](http://photos.yahoo.com). To make sure that static and transient links contain content related to that of the site home page, we required these URLs to have the same hostname as the site itself.

In our methodology, we explicitly divided the links on each home page retrieval for a Web site into static and transient. We then needed to decide how many of each type of link to follow for content collection. We believed that following only a single link was too little and that following all links was too much, in addition to potentially causing denial of service issues if collection was too frequent. For the study we decided to use up to three links of each type (not all sites had three transient links). The reason we chose this number is because we believe it allows us to track the dynamics of a subset of popular pages at a site while not overwhelming the site with requests nor our Content Collector with data. An obvious direction for future work is to examine the effect of alternate criteria for picking the number and type of pages to study at a site.

### 6.1.3 Content Collection Methodology

To collect content for this study we used our Content Collector with the `GetFullPages` configuration, discussed in Section 3.1. We started the Content Collector on June 20, 2002 and it collected content every 15 minutes from 9 am EST until 9 pm EST daily for 14 days, until July 3, 2002. We focused on the daytime hours as the primary time for user and server activity. The 15-minute interval was deliberately chosen to be small enough to capture site dynamics and large enough to avoid any appearance of a denial of service attack. We extended the Content Collector to save the 9 am versions of all home pages at the beginning of each daily retrieval cycle so that the next day it could decide on transient links. We also pre-fetched and stored all home pages on June 19—one day before we started data gathering.

We further extended the Content Collector to also retrieve objects that it has seen within the last hour, even if these objects were no longer embedded on any of the pages in our sets. Having information about an object’s updates for one hour after that object was first accessed allows us to model server invalidation with the length of a volume lease of up to one hour.

#### 6.1.4 Content Conversion

We wrote the Content Converter software to convert collected content into the format required by our simulator. The Content Converter first detects object updates by comparing MD5 checksums of the successive retrievals, and uses the value of the `Last-Modified` HTTP response header, if it was present, or the retrieval timestamp as the time of the update. We are aware that the latter approach provides only an estimate of the exact update time and underestimates the number of updates to an object, but it matches the granularity of our study. The Content Converter creates a list of updates for each object, keeping track of update time, new size, and the set of added and deleted embedded objects.

The Content Converter assigns appropriate change characteristics to collected objects using the following rules: 1) an object is BoA if it changes on every retrieval; 2) an object is St if it does not change over the course of the retrievals *and* has an `Expires` HTTP response header with a value of one year or more; 3) an object is RSt if the median time between object updates is 24 hours or more; 4) an object is RDyn if the median time between object updates is less than 24 hours.

We did not observe any periodic objects in our data sets. The Cumulative Distribution Function (CDF) of the median times between object updates for all ND objects across all sites is shown in Figure 6.1. The graph shows that in our data about 15% of all non-deterministic objects are classified as RDyn, with the remaining 85% classified as RSt. The smallest time between object updates that we were able to detect for objects that did not have last modification timestamps was 15 minutes. The graph thus shows virtually no

objects with update intervals smaller than 15 minutes. Also, we did not have enough information to determine the median time between updates for those ND objects that did not change during the course of our retrievals. For these ND objects, we set the time between updates to one year for the purposes of including these objects in the CDF; that value has no effect on the simulations. As Figure 6.1 shows, about 85% of all ND objects in our data set did not change over the two-week period of our study.

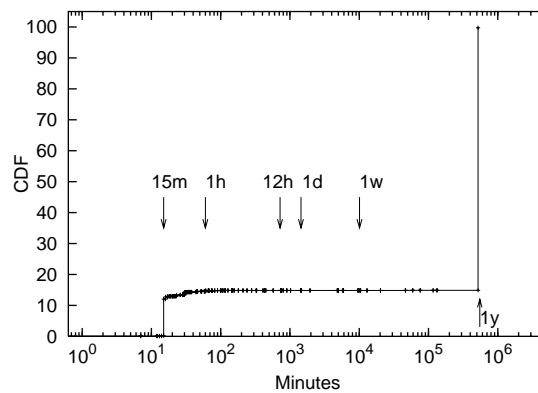


Figure 6.1: CDF of the Median Time between Non-Deterministic Object Updates

Table 6.2 shows the total number of objects collected from each site along with the number of objects in each category of change characteristic. Numbers in parenthesis indicate how many of these objects appeared on more than one page at the site. The Content Converter marks shared objects as *global*. If an object is shared between multiple pages via the `FRAME`, `IFRAME`, or `LAYER` HTML tag, the Content Converter marks all objects embedded in such a container as *global*.

The Content Collector encountered redirects pointing to the same location on every retrieval and encountered redirects pointing to different locations. We modeled the former by creating a permanent mapping between the original URL and the new URL. We modeled the latter by introducing a zero-size object containing an HTML `LAYER` tag (`FRAME` or `IFRAME` tag would also work) pointing to a different embedded object on every access.

Objects composing Web pages may reside on servers other than the origin server. For



Table 6.2: Dynamics of the Collected Objects

Site	Total	local and global objects (global objects)			
		BoA	RDyn	RSt	St
amazon	2642	1071	23 (2)	1548 (942)	0
boston	3987	1788 (5)	687 (10)	1342 (443)	172 (163)
cisco	76	2	5	69 (5)	0
cnn	1448	58 (20)	87 (3)	1303 (378)	0
espn	2095	742 (14)	71 (9)	1222 (529)	60 (30)
ora	180	17	13 (2)	150 (35)	0
photonet	5256	321	141	4794 (319)	0
slashdot	8830	358 (8)	151 (5)	8321 (907)	0
usenix	56	0	0	56 (18)	0
wpi	86	0	3	83 (26)	0
yahoo	1890	231	30 (7)	423 (88)	1206 (444)

this work, we treat all objects composing a page as if they came from the same server. We believe this approach is justified because content served by other servers, such as image or CDN servers, is often under the same control as that from the origin server.

## 6.2 Performance Evaluation

Our goal in this work is to evaluate and compare the performance of the cache consistency policies that are currently deployed and that were proposed in research literature. This section first describes our simulation methodology, the cache consistency policies that we studied, and the performance metrics used.

### 6.2.1 Web Object Management Simulator

In order to evaluate the performance of MONARCH and compare it to that of other object management policies, we developed the Web Object Management Simulator (WOMS). WOMS is a discrete event-based simulator that handles request arrival and object update events. WOMS assumes an infinite capacity cache and no cache replacement policy. Object

management policies are implemented as plug-ins—as long as policies use the simulator’s programming interface, new policies can be written and added to the simulator. The simulator simply feeds each occurring event to all available plug-ins.

Once the collected content is converted, it is presented to WOMS in the form of a (large) configuration file that contains all the necessary information about objects at a site, their relationships, sizes, HTTP response headers, and updates. WOMS is also presented with information about what URLs the Content Collector fetched and when. Additional input parameters specify which of the collected pages to simulate requests for (container and embedded objects) and at what times to make the requests.

### 6.2.2 Simulation Methodology

Values for most of the parameters affecting the outcome of the simulations, such as object sizes and times of object updates, are determined by the collected content. To decide on which access patterns to simulate, one could examine specific access patterns at a Web site. However, in the absence of server logs, an alternate approach is needed.

The approach we used was to investigate a range of possibilities for what content was retrieved on each access and the frequency of these accesses. We simulated the following sets of pages for a site:

- home page only to represent the minimal set,
- home page and the static links to represent a user interested in regular features of the site, and
- home page, the static links and the transient links for that retrieval time to represent the maximal set of the collected content.

In addition, we simulated the following retrieval times based on our collection period of every 15 minutes from 9am to 9pm each day:

- every 15 minutes, the maximum rate possible with the granularity of our collection data, which could simulate requests from a proxy server for a pooled set of users,
- once a day at 9am representing a regular, but relatively infrequent, visitor to the site and
- multiple times a day at 9am, noon, 4pm, and 8pm representing a frequent visitor to the site.

These content and frequency patterns yield a total of nine combinations that we simulated for the collected content of each site.

### 6.2.3 Cache Consistency Policies

As the simulator processes the requests, it simulates all implemented cache consistency policies. All object management policies studied in this dissertation faithfully obey object expiration times and do not cache objects marked as uncacheable. In addition to **MONARCH (M)**, we simulated eight other policies. One policy is the **No Cache (NC)** policy that mimics a non-caching proxy positioned between a client and a server counting messages and bytes transferred. Another policy is the **Optimal (Opt)** policy that has the perfect knowledge of object updates, maintains strong consistency, and contacts the server only when necessary. The next policy is the **Never Validate (NV)** policy that never validates cached objects. Another policy is the **Always Validate (AV)** policy that validates cached non-deterministically changing objects on every access.

We studied the de facto standard **Heuristic** policy, with 5% (**H5**) and 10% (**H10**) of the object's age used as an adaptive expiration time for non-deterministic objects. In addition, we examined the **Current Practice (CP)** policy, which is identical to the H5 policy, except the CP policy also faithfully obeys the HTTP directives related to caching, such as **Cache-Control**, **Expires**, and **Last-Modified**. The CP policy exemplifies the behavior of a caching device deployed on the Internet today. The CP policy is the *only*

policy in our study that is aware of the HTTP response headers. All other policies use only change characteristics identified by the Content Converter.

We also studied a form of server invalidation—the **Object and Volume Leases (OVL)** policy [98], where servers maintain per-client volume and object leases. Clients must hold valid volume and object leases to reuse a cached copy of an object and to receive object updates from the server. We used one hour as the volume lease length and set the object lease length to be longer than the duration of the simulation. The server sends out updates only for non-deterministic objects. Our simulation assumes reliable and timely delivery of invalidation messages to client caches. We do not account for the details of how to handle updates in the face of slow or unavailable clients [101, 99].

#### 6.2.4 Performance Metrics

In order to evaluate the performance of each cache consistency policy and to compare the policies, we used the following performance metrics. For each policy we computed the number of stale objects served from the cache, the number of requests that the cache sent to the server, and the number of bytes served by the server. For MONARCH and server invalidation policies, we computed the number of separate invalidation messages, number of invalidation messages piggybacked onto server responses, and average number of objects invalidated in a piggybacked invalidation message. For MONARCH and server invalidation policies, we also computed the amount of server state that must be maintained. We discuss these state-related metrics in more detail in Section 6.3.3.

## 6.3 Results

For each of the eleven sites (shown in Table 6.1) we performed simulations with all scenarios discussed in Section 6.2.2. For the purpose of the discussion, unless indicated otherwise, all results in this section are from the scenario where the home page, static links, and transient links are retrieved from a site four times each day. This scenario was chosen from the

nine described in Section 6.2.2 because it represents the maximal amount of content at an intermediate access frequency. The relative performance of different policies for the other scenarios is generally consistent in tone with those shown. More frequent accesses result in more reusable cache content, while less frequent accesses result in more content that must be retrieved from the server. In general, the content on site home pages is more dynamic than the content of linked pages.

We first discuss the effectiveness of the policy that models the behavior of modern caches. Then, we compare the performance of different policies in terms of the amount of generated traffic and staleness. After that, we discuss the amount of overhead that the two stateful policies (MONARCH and OVL) incur at the server. Finally, we examine the extent to which cache consistency policies affect end user response time.

### 6.3.1 Effectiveness of the Current Practice Policy

One of the performance goals in this work is to evaluate the effectiveness of the cache consistency policy that reflects the current practice. Performance of the CP policy across all eleven sites in terms of the average number of requests that the server received and the average number of KBytes that the server served per page retrieval is shown in Table 6.3. For comparison, the table also shows the best (Opt) and the worst (NC) case policies. The results indicate that the CP policy avoids transferring 50–60% of bytes (up to 96% for the `usenix` site) as compared to the NC policy. For seven sites in our set, the CP policy also transfers only marginally larger number of bytes than the Opt policy. For the other four sites, however, the CP policy transfers 1.2–7 times more bytes than the Opt policy. We investigated the reason for such a discrepancy and discovered that sites often mark objects that change infrequently as uncacheable or generate such objects upon request and provide no information that caches can use to subsequently validate these objects. The results further indicate that the CP policy issues 1.8–5.4 times more requests to the server than the Opt policy. In terms of staleness, the CP policy serves stale objects for eight sites in

at least one simulation scenario. For two sites (`usenix` and `ora`) the CP policy serves stale objects under all simulation scenarios.

Table 6.3: Performance of the Current Practice Policy (\* indicates stale content served in at least one simulation scenario)

Site	Requests and KB served by Server					
	Opt		CP		NC	
amazon*	3.2	45.1	5.9	45.7	35.2	107.5
boston*	3.6	50.6	19.3	54.4	25.5	113.5
cisco	1.9	2.9	3.5	19.6	19.2	55.4
cnn*	6.3	56.1	16.4	77.6	31.4	190.8
espn*	4.3	75.3	19.4	85.4	38.7	159.5
ora*	0.9	13.8	2.7	14.2	19.9	97.1
photonet	3.1	33.4	3.7	34.5	8.5	55.5
slashdot*	3.1	38.4	7.8	39.5	15.0	70.5
usenix*	0.3	0.8	0.9	0.9	21.1	28.4
wpi*	0.5	2.9	2.3	15.3	25.8	61.2
yahoo	3.7	34.1	6.5	39.2	15.6	71.0

### 6.3.2 Comparison of Cache Consistency Policies

We examined performance of policies other than CP on all sites and present results for three sites—`cnn`, `espn`, and `cisco`—in Figures 6.2–6.4. These sites represent a range of policy results. The horizontal and vertical axes are expressed in percentages relative to the NC policy. On the horizontal axis, we plot the percentage of requests that each policy sent to the server per page retrieval, and on the vertical axis we plot the percentage of bytes that the server served under each policy. Policies that served a non-zero number of stale objects to clients in these simulation are marked with asterisks. The number of requests and bytes under the NC policy is shown in the figure captions.

The graphs indicate that caching of content using any of the policies shown, including the AV policy, offers substantial (at least 50–60%) byte savings. The two heuristic policies H5 and H10 outperform the CP policy both in terms of requests and bytes. The results

indicate that in terms of the traffic between the cache and the server, both MONARCH and the OVL policies provide indistinguishable performance from the Opt policy.

Staleness results across all sites indicate that under at least one simulation scenario, both H5 and H10 policies on average served a small number of stale objects per page retrieval. The amount of stale content served is substantially smaller than the upper bound on staleness provided by the NV policy. Across all sites and all simulation scenarios the NV policy served on average 0.4–3.0 stale objects per page retrieval, and for the simulation scenario used in this discussion it served 0.4–1.5 stale objects.

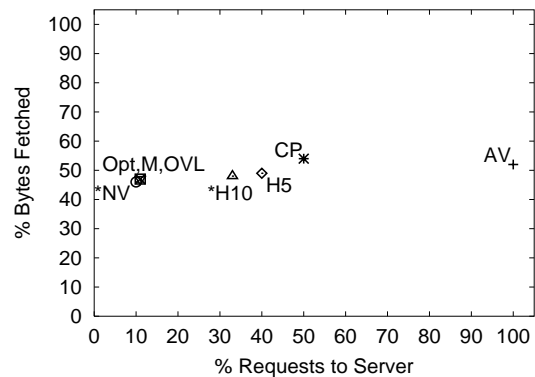
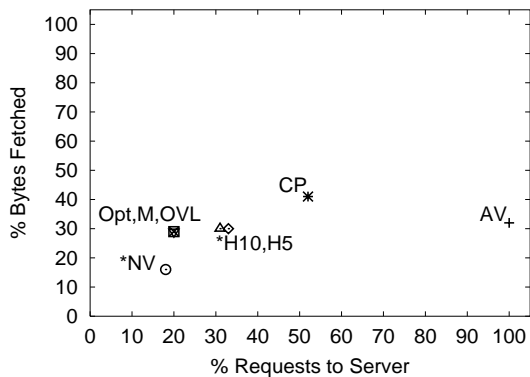


Figure 6.2: CNN, 31.4 Requests, 190.8 KB. Figure 6.3: ESPN, 38.7 Requests, 159.5 KB.

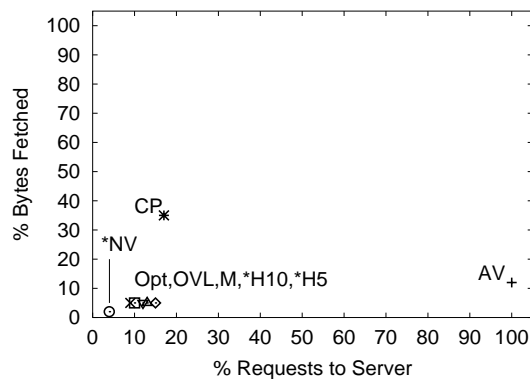


Figure 6.4: Cisco, 19.2 Requests, 55.4 KB.

### 6.3.3 Server Overhead

The MONARCH and OVL policies provide strong cache consistency and also exhibit similar performance in terms of the generated traffic. However, compared to other policies studied, the MONARCH and OVL policies incur overhead at the servers because they require servers to maintain state and perform additional processing to achieve strong cache consistency. In this section we describe the metrics that we use to evaluate server overhead imposed by each of the two policies and present values for these metrics obtained from the simulations.

Both the MONARCH and OVL policies must keep track of updates to non-deterministically changing objects in order to maintain volume information and notify clients of such updates. As a measure of the overhead associated with these updates, we compute the average number of daily updates to non-deterministic objects (**NDU**). The NDU results for each site are shown in Table 6.4.

Table 6.4: Server Overhead

Site	NDU	MONARCH		OVL (AOL)	
		VOL	VR	avg	max
amazon	4	1271	159	464	706
boston	300	138	218	409	657
cisco	125	4	121	67	70
cnn	109	84	198	580	941
espn	67	63	167	525	806
ora	29	22	13	122	151
photonet	220	434	211	432	770
slashdot	167	33	330	404	731
usenix	0.3	5	0	55	56
wpi	2	6	0	83	86
yahoo	110	51	187	145	248

MONARCH maintains per-page local volumes and one global volume that incorporates objects shared between pages. MONARCH increments a volume version when volume membership changes. We capture the overhead associated with volume maintenance using the average number of unique volume revisions (**VR**) created daily. In computing the



VR numbers, we take into account original volume versions created at the start of the simulation. The average daily number of volume revisions and the total number of local volumes created (**VOL**) are shown in Table 6.4.

The OVL policy maintains a list of per-client volume leases and per-client object leases. We focus only on object leases, as the overhead associated with volume leases is likely to be significantly smaller than that of object leases. We measure the overhead of object leases by recording the number of active object leases (**AOL**) held for one client after it makes requests. Table 6.4 shows the average and maximum number of active volume leases held for one client at each site.

As we examine the metrics shown in Table 6.4, we see that the rate of updates to non-deterministic objects varies from 0.3 to 300 per day. We further investigated the NDU results and discovered that for two sites, including **boston**, only a handful of objects (10 or fewer) are responsible for over 50% of all NDU updates. These frequently changing objects could be marked as BoA instead of RDyn to reduce the overhead associated with changing objects, albeit with diminished cached content reuse.

The highest daily number of volume revisions in our simulations is 330 and was observed for only one site. For seven other sites, the daily VR increase is under or slightly over 200. The number of volume revisions for less frequently changing sites, such as **ora**, **wpi** and **usenix**, is either small or zero. Our results show that for six sites the OVL policy must maintain over 400 active object leases per-client, and must also maintain leases even for sites that do not have many object changes. We also investigated the effect that frequency of request arrivals has on the overhead of the two policies. For four sites—**wpi**, **usenix**, **ora**, and **cisco**—the overhead of the OVL policy remains unchanged as the request arrival rate increases from 4 times a day to every 15 minutes. For the other seven sites, the number of active object leases maintained by the OVL policy increases as follows. For five of the seven sites, the average AOL grows by 27–58% and the maximum AOL grows by 29–48%. For the other two sites—**photonet** and **slashdot**—the increase is especially large. For the former

site, the average and the maximum AOL increase by 5.6 and by 5.7 times respectively. For the latter site, the average and the maximum AOL increase by 9.6 and by 10.2 times respectively. The overhead of MONARCH is not affected by fluctuations in the request arrivals or the number of clients.

We also examined the overhead associated with the invalidation activity of MONARCH and the OVL policies. The invalidation behavior of the two policies is different and cannot be compared directly. MONARCH always piggybacks invalidations onto its responses to clients, while the OVL policy sends out invalidations both piggybacked onto other messages and as separate messages. Our results indicate, however, that these differences are not that important. Invalidation traffic in terms of separate messages, piggybacked messages, and objects invalidated in one message is negligible for both policies across all sites and all simulation scenarios.

#### 6.3.4 Response Time Implications for Different Policies

This study allows us to determine the performance of different policies in terms of the requests and byte traffic between caches and servers. It is less clear how this performance impacts the end user. As a means to study this issue, we use performance data that was recently gathered by Krishnamurthy et al. [51].

Krishnamurthy et al. [51] characterized pages based on the amount of content on a page, which they defined as the number of bytes in the container object, the number of embedded objects and the total number of bytes for the embedded objects. Using proxy logs of a large manufacturing company, popular URLs containing one or more embedded objects were successfully retrieved and the 33% and 67% percentile values were used to create a small, medium, and large value range for each characteristic. Using these three ranges for each of the three characteristics defines a total of 27 “buckets” for the classification of an individual page. The cutoffs for container bytes in small, medium, and large were less than 12K, less than 30K bytes, and more than 30K bytes respectively. Similarly, for embedded

objects it was less than 7, 22, and more than 22 and for embedded bytes 20K, 55K, and more than 55K bytes. The authors identified test pages that spanned the space of these characteristics and created a test site of content. They installed the test site on unloaded servers on both coasts of the U.S. and used *httperf* [66] from six other client sites to make automated retrievals to each test server for each test page.

In this work, we use the results from [51] as benchmark response time performance measures for different types of clients and amounts of content. We focus on results from retrievals using up to four parallel TCP connections and HTTP/1.0 requests. Persistent TCP connections with pipelining are expected to produce better results, but pipelining is not commonly used by real clients and proxies. Persistent connections with serialized HTTP requests have been shown to perform no better than four parallel connections [51]. Our methodology is to map the requests and amount of content served under each policy in this work to a corresponding bucket from [51]. We then use the benchmark performance of different clients tested in [51] as an estimate of the relative performance of the various policies.

The buckets in [51] are only coarse classifications and, not surprisingly, in many cases policy traffic performance maps to the same bucket. For example, the Opt, MONARCH, and OVL policies invariably map to the same bucket across different Web sites and retrieval patterns. This convergence is realistic as only significant differences between policies for the number of objects or number of bytes is going to translate into significant response time differences between the policies. The heuristic policies sometimes map to the same bucket as the Opt, MONARCH, and OVL policies and in other cases map to the same bucket as CP. The AV and NC policies generally map to distinct buckets. Given these observations, we show results for the MONARCH, CP, AV, and NC policies for a commercial and modem client.

Figure 6.5a shows results for two sets of sites in our study. The response time results, obtained in [51], are from a commercial client on the East Coast of the U.S. to the West

Coast server. Results for the `boston`, `cnn`, and `espn` sites are mapped to the same buckets and are shown together. Response time for the MONARCH policy is improved relative to current practice and is much better than no cache, although the absolute differences are smaller because the client is well connected. Figure 6.5a also shows that for pages on Web sites with less content, there is less difference between the performance of different policies. The AV policy generally yields worse response time than current practice, although pipelining of responses can reduce the difference.

We also used results in [51] from a modem client on the East Coast to the East Coast server. These results for the various policies and Web site pages are shown in Figure 6.5b. Due to the reduced bandwidth of the client, the absolute differences between the policies is greater, particularly for Web site pages with more content.

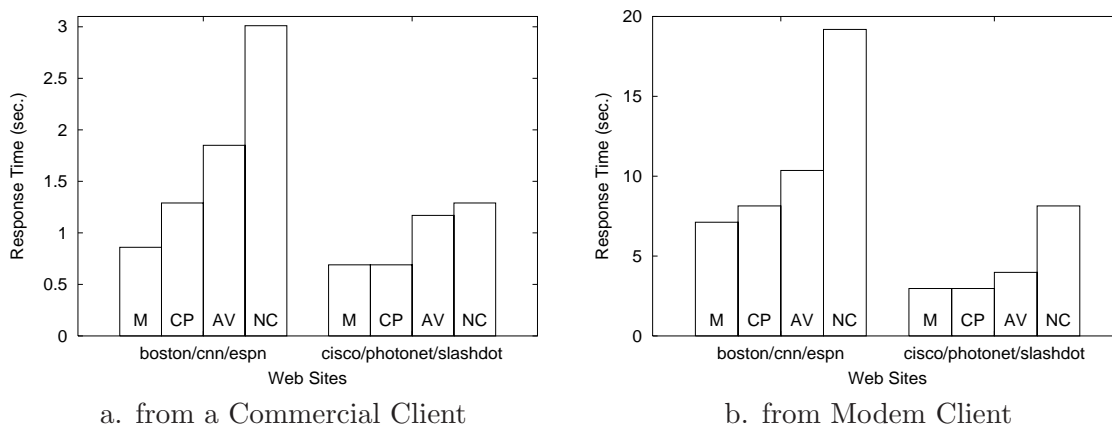


Figure 6.5: Estimated Response Time Performance for Different Policies for Web Site Pages Using Para-1.0 Results in [51]

The results above are averages for all pages at each Web site. We also examined the relative response time differences when retrieving just the home page at multiple times each day. For this analysis, four of the sites showed some response time difference between the MONARCH and CP policies. Overall, the results show that better cache consistency policies can improve expected response time relative to current practice for larger, more dynamic, pages.

## 6.4 Content Reuse

The fact that MONARCH manages objects deterministically rather than heuristically, and gives content providers more control over their content, leads us to believe that content providers may finally be willing to expose internal structure of their pages to clients. In light of this possibility, we need to better understand the impact that exposing page structure to clients could have on performance. In Section 3.3 we presented our study that evaluated potential byte savings if HTML objects are decomposed into chunks and found that about 75% of cached HTML bytes can be reused. We also found that delta encoding can yield substantial reuse. Both page decomposition and delta encoding approaches show significant savings due to better reuse of HTML content that changes between accesses. We need to understand whether HTML content accessed more than once in our simulation amounts to a substantial fraction of all retrieved content. To study this issue, we examined the number of requests and bytes contributed by objects that the cache has not seen before (new objects) and objects that changed since the last time the cache saw them (changed objects) under the Opt policy. We further broke down the two categories of content based on content type.

We show the average number of requests and bytes served by the server per page retrieval under the Opt policy in Table 6.5. The two columns showing the total number of requests and bytes have the same values as the column for the Opt policy in Table 6.3. Our results indicate that for many sites half or more bytes that the cache retrieved from the server are for cached HTML objects that changed at the server. If we focus on the sites for which the cache obtained the most content from the server, such as `espn`, `cnn`, and `boston`, we see that changed HTML objects amount to 70%, 47%, and 55% respectively of the overall bytes retrieved. These numbers are significant as they indicate that reusing content within cached HTML objects would be a substantial improvement.

Different approaches have been proposed for better reuse of cached content, such as delta encoding, templates with dynamic bindings, and breaking pages into components. To better understand how these approaches compare, we need to examine factors other than just byte

Table 6.5: Requests and Bytes Served by the Server under the Optimal Policy due to Retrieval of New and Changed Objects

Site	Requests			KBytes		
	Total	Changed Objs	Changed HTML	Total	Changed Objs	Changed HTML
amazon	3.2	0.8	0.8	45.1	30.5	30.5
boston	3.6	1.0	0.9	50.6	28.2	28.0
cisco	1.9	1.6	1.0	2.9	1.7	1.4
cnn	6.3	3.5	3.4	56.1	26.5	26.2
espn	4.3	1.5	1.1	75.3	53.5	52.9
ora	0.9	0.3	0.4	13.8	9.7	9.7
photonet	3.1	0.7	0.8	33.4	9.8	9.7
slashdot	3.1	0.9	0.9	38.4	22.1	22.1
usenix	0.3	0.0	0.0	0.8	0.3	0.3
wpi	0.5	0.1	0.1	2.9	1.6	1.6
yahoo	3.7	0.9	0.8	34.1	20.4	20.2

savings. One factor to consider is the number of requests generated by caches. The left side of Table 6.5 shows that for nine sites the majority of requests served by the server are due to the cache retrieving objects it has not seen before. If HTML pages are constructed from components, caches would need to issue even more requests to obtain all objects. The number of requests for changed objects would also increase if multiple components change between retrievals. These findings suggest that if content providers expose constituent page components to clients, we need to devise a mechanism that reduces the number of requests that are due to the use of components. Comparing the response time results in Figure 6.5 with the traffic results in Figures 6.2—6.4 we see that requests influence the response time of policies more than bytes do. Thus, reducing the number of requests is important. One approach to reducing the number of requests issued by caches would be to bundle a set of components, such as components with the same change characteristic, together. Another approach might be to treat a set of components as a delta between the current and the updated versions of the page. Investigating these approaches from the feasibility and performance points of view is a clear direction for future work.

## 6.5 Summary

In this chapter, we first described a novel methodology for evaluating a wide range of cache consistency policies over a range of access patterns and then used it to examine the performance of MONARCH and compared it to the performance of current practice and other cache consistency mechanisms. In addition, we used prior results obtained by others to study whether and how traffic variations between various policies affect the user-perceived response time.

The essence of our methodology is to actively gather selected content from sites of interest and then use the collected content as input to a simulator. Traditional use of proxy and server logs for evaluation of cache consistency policies has its limitations. While proxy logs contain real client request patterns, they do not contain the complete request stream to a particular site and provide no indication of when resources change. Logs from popular server sites are not generally available to the research community and do not contain a record of object modification events. Neither server nor proxy logs contain HTTP cache directives, making it impossible to evaluate the effectiveness of policy reflecting current practice. Our content collection methodology is a step towards filling these gaps and obtaining data for any site of interest that is not otherwise available for study.

The results show that for all sites the Current Practice policy yields significantly better response time than if no caching is used. For some sites, the Current Practice policy also yields close to optimal cache performance, but for larger, more dynamic, sites it generates more request and more byte traffic than is necessary. We discovered that sites often either do not provide any cache control information or unnecessarily mark objects as uncacheable. Our results also indicate that the Current Practice policy serves stale objects to clients for a variety of simulation scenarios. Overall, caching objects using the Current Practice policy is beneficial, but there is certainly room for improvement.

Our results show that the MONARCH approach to cache consistency provides substantial improvement over heuristic policies, including the Current Practice policy. MONARCH

---

provides strong cache consistency across all simulation scenarios and outperforms all heuristic policies in terms of the request and byte traffic between caches and servers. Our results also show that for the sites studied, MONARCH generates the same amount of byte traffic and little more request traffic than an Optimal cache consistency policy. Simulations with synthetic content yield similar results. Our results further indicate that the performance of MONARCH is indistinguishable from the performance of the Optimal policy in terms of response time. As compared to the Current Practice policy, for some sites MONARCH showed no improvement in the estimated response time, but for Web site pages for which many validation requests are needed MONARCH did show moderate improvement.

Our results indicate that the performance of MONARCH in terms of staleness, request and byte traffic, and response time is closely matched by the OVL policy. These two policies are the only ones in our study that maintain state at the server. The OVL policy maintains per-client leases and the MONARCH policy keeps track of volume membership changes. Results for the OVL policy show that for many of the sites studied the number of per-client object leases is in the hundreds even though few of the objects actually change. The amount of state for the MONARCH policy is up to a couple hundred volume membership revisions per day. While the overhead of these two policies is different and cannot be compared directly, the overhead of the OVL policy increases with the number of clients and we showed that it also increases as requests arrive more frequently. MONARCH is not affected by fluctuations in the request arrivals or the number of clients.

We believe that content providers may be willing to expose internal structure of their pages to clients if servers and caches use MONARCH because MONARCH manages objects deterministically rather than heuristically and gives content providers more control over their content. We evaluated potential byte savings if parts of repeatedly accessed HTML objects are reused by the cache across retrievals and found that substantial savings are possible. Our findings also indicate that if pages are constructed from components servers are likely to receive more requests than if pages are monolithic. Investigating approaches



for better reuse of cached content is a direction for future work.

We conclude that the MONARCH approach to cache consistency is a substantial improvement over heuristic object management policies in terms of object staleness and request and byte traffic. MONARCH is also an improvement over the OVL policy because the server state maintained by MONARCH is not affected by fluctuations in the request arrivals or the number of clients. The evaluation presented in this chapter complements the design and implementation of the prototype system presented in the previous chapter in validating the hypothesis of this dissertation.

## Chapter 7

# Conclusions and Future Work

We hypothesized in this dissertation that we can improve upon existing heuristic- and invalidation-based object management techniques so that a group of related objects in a distributed system can be managed with both consistency and efficiency. Our goal in this dissertation was to eliminate consistency failures and per-client server overhead while minimizing unnecessary requests. In the previous chapters, we have presented our approach, evaluated it, and compared it to other existing and proposed approaches. This chapter summarizes the contributions of this dissertation and presents ideas for future work.

### 7.1 Contributions

In pursuing the objectives of this dissertation, our research has made the following contributions:

- We have proposed and successfully used a novel methodology for studying Web resources and understanding how they change, which we discussed in Chapter 3. Instead of examining proxy and server logs or packet traces of real user request/responses, as was done by previous studies, we identify a set of Web resources to study and then actively retrieve them over a period of time. Our methodology allows one to

study any desired set of resources, instead of being constrained to the data available in logs or traces. In addition, our methodology allows one to obtain server-supplied cache-control information, which usually is not available in logs. Having cache-control information associated with the resources studied permits one to evaluate cache consistency mechanisms and study the potential improvement of caching schemes.

- We have used the methodology discussed above to perform a study characterizing Web resources. The study and its results provide an important contribution as they better explain how Web resources change and how they are related to each other. We have performed that study twice, with four years between the two studies. The results obtained both times agree with each other. That is also a contribution as it shows that types of changes in Web resources and the relationships between Web objects have remained stable over time. We use these findings in the dissertation.
- We have classified object changes based on their frequency and predictability and provided a taxonomy of object change characteristics. We discussed the implications that object change characteristics have on object management at origin servers and intermediaries. Understanding how objects change is important because the presence of objects with different update schedules inherently calls for different invalidation/update strategies. Armed with this information we can improve upon existing object management mechanisms. We use these change characteristics in this dissertation.
- We have examined possible combinations of the composition relationship between two objects and object change characteristics and identified three important patterns for deterministic object management. We have also extended our finding from two objects to any number of objects. These patterns are important because identifying a pattern that describes a given set of related objects leads to the management strategy for that set of objects.
- We have devised a novel object management mechanism, called MONARCH, that

chooses the most appropriate management strategies for a given set of objects. MONARCH exploits relationships between objects in a set in conjunction with object change characteristics—information that all other object management mechanisms ignore. MONARCH achieves strong cache consistency for all objects and reduces traffic overhead, as compared to heuristic-based mechanisms. MONARCH maintains no per-client state at the server, unlike mechanisms based on server-initiated invalidation. Even though we evaluated the MONARCH approach to object management using Web content designed for traditional Web browsers, the approach is not specific to HTML or even to the Web. The MONARCH approach can be applied to managing any set of related objects in a distributed system, such as objects in an on-line computer game, distributed simulation, or CAD project.

- As part of MONARCH, we have proposed providing caches with concise and unambiguous instructions on how to manage each object. Current object management mechanisms are heuristic. With our Content Control Commands, caches no longer need to use heuristics to estimate object freshness lifetimes. We believe the use of CCCs leaves control over content in the hands of content providers while allowing caches to cache content and serve it from the edge of the network.
- We have designed and built a prototype system implementing MONARCH. The prototype system is a substantial contribution because it validates the proposed approach and shows that such a system can be built. While working on the system, we encountered a number of issues, such as keeping track of volume revisions or deciding on the set of objects to invalidate. Documentation and discussion of these issues, as well as our solutions to them, is also a contribution. Other researchers wishing to implement similar functionality can build on our experience.
- We estimated the potential reduction in the number of unnecessary validation requests offered by our approach. also studied a number of real Web pages to estimate

the amount of byte savings if pages were constructed from components and caches were allowed to cache these components. These evaluations show the potential for substantial improvement offered by MONARCH. The methodologies that we used in these studies are also contributions as we are the first to use such evaluation methodologies to study the potential of a new object management mechanism.

- We are also the first to propose using snapshots of content actively collected from real Web sites to evaluate cache consistency policies. Previous studies used proxy, server, and object update [101, 99] logs for such evaluation. It is well-known that such logs are hard to find and they tend to be from smaller or research-oriented sites. They also tend to be dated. It is difficult to obtain (recent) traces from large and popular commercial Web sites. Yet, evaluation of new proposals on precisely these types of sites is of most interest. We applied this methodology for collecting snapshots of content and for using it in simulations.
- While evaluating MONARCH and comparing its performance to that of existing object management policies, we also evaluated the policy that reflects the behavior of currently deployed caching proxies. We were able to do that due to our collected content methodology discussed above. We are not aware of other studies that evaluated the performance of the Current Practice policy taking into account the HTTP cache-control directives used by real Web sites. Such evaluation sheds more light on the effectiveness of caching mechanisms in HTTP as used by real sites.

## 7.2 Lifetime of Contributions

Any research project, especially as large and lengthy one as a Ph.D. Dissertation, causes concerns regarding the relevancy of work when the project is completed. A problem that is acute at the start of the project may become irrelevant by the end of the project.

The problem addressed in this dissertation is about an efficient mechanism for providing

strong cache consistency for a set of related objects in a distributed system. We discussed the importance of caching for distributed systems, especially large ones, in Chapter 1. The Internet and the Web continue to grow and thus caching continues to be important for these systems. As these systems become more entrenched in our society, we more rely on them for not only casual but also mission-critical tasks. We place new requirements on the systems and require new guarantees, such as ensuring strong cache consistency. The issue of improving efficiency of distributed systems is always important, especially as systems grow in size and popularity. The problem addressed in this dissertation is as important or perhaps even more important today than it was when we started this work. We believe the ideas described in this dissertation are applicable not only to HTML pages and objects embedded in them, but also to non-HTML content and to domains other than the Web and have the potential to fuel the next wave of research activity. We discuss our ideas for future research directions next.

### 7.3 Future Work

Any work must be limited in scope to maintain focus and ensure timely completion. While devising and evaluating the MONARCH approach, we encountered a number of directions for future exploration. We have compiled a list of these directions for future work and present them in this section.

One set of directions for future work is on content assembly. We discussed content assembly and described how it is carried out in the MONARCH prototype system in Section 5.5. Akamai has also proposed caching of page components and their assembly at the edge of the network using their ESI technology [29]. We have already started extending our basic content assembly approach with partial content assembly and with the integration of content assembly and personalization. We are interested in evaluating how these additions may affect our approach to Web object management. We elaborate on these two extensions below.

### 7.3.1 Selective Content Assembly

Servers and caches could choose to perform *partial* (or *selective*), instead of full, content assembly. A decision to assemble all components of a composite object or only selected ones could depend on a number of conditions. An overloaded server could skip the assembly or assemble only those components that are present in the cache and are fresh. In a caching hierarchy where caches have assembly capability, only servers at the client-side edge of the network may be allowed to assemble content. Servers can further exploit relationships between components and take into account object change characteristics in deciding whether to perform full or partial assembly. For example, a server may assemble all related objects that have the same change characteristic, cache the result of the assembly, and reuse it on the subsequent requests, thus amortizing the cost of the assembly over multiple requests. We plan to explore these issues in future work.

### 7.3.2 Assembling Customized Content

In this dissertation, we treated all objects that must be generated upon request as BoA, as discussed in Section 4.5. Some objects are treated as BoA because origin servers generate them based on the information in the client request, such as a cookie identifying a particular user, parameters that follow a ? in the requested URL, or an HTTP request header. Objects that depend on the information in the client request could be separated into their own special category of personalized, or *Input Dependent* (*InpDep*), objects, which is separate from the BoA category. Currently, we treat personalized objects as BoA. Objects in the InpDep category also belong to one of the categories depicted in Figure 4.2. For example, an object can be both RSt and InpDep. That object should be treated as an RSt object as long as requests supply the same input parameter. This property of InpDep objects allows us to remove the dependency on input parameters at the origin server, cache the resulting non-InpDep object, and later on have the cache re-introduce the dependency on input during the content assembly process. The three examples that follow illustrate this approach. The

details of assembling customized content, however, still need to be researched.

### Personalization

Consider the personalized greeting component *CMP1*, shown in Figure 5.7, that is part of the test page used to demonstrate the operation on the MONARCH prototype system. Instead of having the server generate this component on every request, based on the input provided by the user, content designers could remove the dependency on input and make the new *CMP1'* component of the category RSt and cacheable by changing `username.cmp` to:

```
Welcome, <GI SRC=userprofile/<GI ISRC=userid DEFAULT=new-user-id>#Name>!
```

As a cache assembles a page with the *CMP1'* component, it replaces the inner GI construct with the value of the `userid` object that the current request supplies in the cookie. For example, if a request contains the `Cookie: userid=ID1` HTTP request header, the assembler replaces the inner GI tag with `ID1`. Trusted caches and origin servers manage the resulting `userprofile/ID1` as any other object. If the client request contains no `userid` object, then the origin server provides the cache with an id for a new user.

User profiles may contain a number of distinct fields, not all of which may be required for a particular component. To access only the required field, we could use a mechanism similar to that used by Web browsers to navigate to a specific named part of a page. We append `#` and the name of the required field to the name of the object. After obtaining `userprofile/ID1` from the origin server, the assembler replaces the outer GI tag with the content of the `userprofile/ID1` that is located between the `Name` and the next field, finishing the assembly. To prevent input parameters from masking those objects that should be retrieved from the origin server, we use the `ISRC` attribute of the GI tag, instead of `SRC`, to explicitly inform assemblers that the included object may be available in the request.

The real strength of treating an entire user profile as any other object is that caches can avoid contacting the origin server when the same client requests pages that depend on other



fields in the profile, such as address, e-mail or company name. Unless the profile changes at the origin server, caches continue using it to personalize pages that depend on it.

### URL Re-Writing

Many Web sites wish to identify unique visitors and track paths that they follow through the site. Cookies do not work when users turn them off in their browsers, and Web crawlers do not always support cookies. A more robust technique of differentiating between clients is URL re-writing, where for each client the server generates a unique identifier and dynamically appends it to each traversal link on each page that it serves to that client. Pages with re-written URLs are uncacheable even though their content does not change on every request.

Caches enhanced with our content assembly mechanism can re-write traversal links in cached pages by performing simple substitutions and can propagate unique IDs between cached pages without contacting the origin server. Content designers change traversal links in their pages from

```
<a href="link.html">
```

to

```
<a href="link.html?sessionId=<GI ISRC=sessionId">"> ,
```

effectively decoupling the InpDep part from the rest of the page. The modified page now belongs to one of the categories in Figure 4.2, and can be cached if the category is not BoA. Upon receiving a request for such a page, a cache replaces the entire GI tag with the value of the `sessionId` found in the client's request. For example, a request for `page.html?sessionId=ID1` results in the re-written `page.html` containing the link `<a href="link.html?sessionId=ID1">`. If the client follows re-written links within that page, the cache re-writes those pages as well, assuming it has them cached, using the same `sessionId`. If another client requests the same pages, the cache re-writes them with the `sessionId` taken from that client's request. If a request does not contain a `sessionId`

object, the cache obtains a new ID from the origin server. The cache can also prefetch a block of new IDs from the origin server in advance.

The reason sites deploy URL-re-writing is to log all requests and differentiate between unique visitors. Caches with the content assembly capability annul the usefulness of the URL-re-writing to servers by shielding them from client requests. We propose to decouple serving cached content from propagating requests to the origin servers. Servers inform caches via a CCC command whether they want to see each request for a given object right away, at some later point in time or never. If real-time feedback is not required, caches can aggregate requests for a given object and notify the server later, perhaps during off peak hours.

### Input-Based Object Selection

When a Web site is offering content in more than one language encoding, the server decides on the correct encoding at the time of the access by examining the `Accept-Language` HTTP request header. The default installation of the Apache Web server [3], for example, comes with the default home page in a few languages `index.html.de`, `index.html.en`, `index.html.fr`, etc. Currently, such language-specific server responses can be cached with the addition of the `Vary` field that a subsequent client request must satisfy to receive the cached page.

Using our content assembly mechanism, caches can cache all objects that may result from a server making a selection from a finite set of choices, while the `Vary` field supports caching of only one object. Content providers add another object, `index.html`, that contains a single line:

```
<GI SRC=index.html.<GI ISRC=Accept-Language DEFAULT=en>>
```

Upon receiving a request for `index.html`, caches either use the language preference of the browser or the default value of `en` and construct the name of the object with the language-specific content.

### 7.3.3 Content Reuse

We found that better reusing content within HTML objects can offer substantial byte savings. We plan to study different approaches for better reuse of HTML objects, such as delta encoding, breaking pages into components, and HPP, and compare these approaches along dimensions other than just byte savings. We believe there is a number of categories of dynamically generated content and each content reuse approach may be more suited for one category of dynamic content than for another. For example, URL re-writing might be more easily accomplished with HPP than with components. We plan to study dynamic content categories and investigate combining different approaches to content reuse.

As suggested in Section 6.4, the use of components is likely to increase the number of requests that servers receive. We plan to examine and compare techniques for reducing the number of requests contributed by components. One possibility would be to identify components with the same change characteristic and fetch them using a single request. Another possibility would be to have servers provide deltas for all updated components. Combining different approaches is also a direction for investigation.

### 7.3.4 Dynamic Change Characteristics

One interesting question that arises with respect to object change characteristics is whether an object's change characteristic is a function of time. Some objects may never change their change characteristic, while others may switch between change characteristics at different time scales. In this dissertation we assumed that each object has only one change characteristic. Since in reality that may not be the case, we plan to explore this issue further and evaluate how the dynamics of object change characteristics influence object management in general and the MONARCH approach in particular.

### 7.3.5 Deploying MONARCH at an Experimental Site

An ideal way to evaluate MONARCH and other object management policies would be to obtain client access traces and object update records from a variety of sites and replay them using a simulator. While instrumenting sites that are not under our control may not be possible, we could build our own site, with real content, and instrument it to keep track of client accesses and object updates. For example, a set of Web pages with course material could be treated as a small site. Students accessing course information, syllabus, and project description pages generate real client accesses. We could also add MONARCH functionality to the Web server and co-locate a MONARCH cache with the Web server. The MONARCH cache keeps a detailed record of client accesses, cache hits and misses, and communications with the MONARCH Web server, similarly to how our prototype MPS kept a record of its activity, as described in Section 5.4.5.

Traces of client accesses and object updates taken at a small site with few users and little offered content may not be representative of larger sites with large amounts of content and millions of users. Still, we believe it would be valuable to evaluate MONARCH and other object management policies on such a realistic workload. However small, new knowledge and better understanding do reduce the amount of the unknown. Testing MONARCH on a live system would also be interesting and valuable since live deployment may uncover unexpected behavior, which may lead to better understanding of the issues involved and new research directions.

### 7.3.6 Coupling MONARCH with Existing Templating

#### Mechanisms

In addition to deploying MONARCH at an experimental site, as discussed in Section 7.3.5, we are interested in examining the issues of coupling MONARCH with existing templating mechanisms, such as PHP [75] and Mason [59]. Templating mechanisms are widely used to construct Web sites, and have mechanisms for building pages from components. We could

convert our experimental Web site, discussed in Section 7.3.5, into a PHP- or Mason-based site, and study the interaction between MONARCH and PHP or Mason. Understanding the issues involved and finding ways to address them is important to the gradual deployment of MONARCH on the Web.

### 7.3.7 Deployment Issues

Considering a newly proposed approach or a protocol for deployment in an existing distributed system, especially as diverse and open as the Internet, raises a host of additional questions that may not even be applicable to a controlled laboratory environment. Consider one such issue—authentication. In the MONARCH prototype system, discussed in Chapter 5, client authentication is quite simplistic—both MPS and MWS examine the `Accept` HTTP request header in the client request to decide whether the client is allowed to receive constituent page components or not. Deployment scenarios where MPS and MWS communicate over an open channel require more sophisticated authentication schemes. A possible future direction is studying such issues as authentication and security and subsequently formally describing the MONARCH approach and these issues in an Internet Draft document.

### 7.3.8 Enabling End Client Nodes with the MONARCH Capability

We plan to explore the possibility of enhancing Web clients with the MONARCH capabilities. Extending the request and byte savings offered by MONARCH all the way to the client would provide an even greater scalability and efficiency than offering MONARCH functionality up to the edge of the network. The benefits should be especially significant where client connectivity to the network is inferior, latency or bandwidth-wise, compared to the cache-server connectivity.

There could be different approaches to extending the MONARCH mechanism to the

client nodes. One approach would be to produce a version of the MPS that could run on a client machine and work in conjunction with a standard Web browser. A Web browser can be explicitly configured to access the Web through a proxy server. The proxy server carries out all the communication with the origin server, caches constituent page components, and embedded objects and assembles pages before handing them to the Web browser.

Having a separate proxy server, however, requires users to download, install, and configure another piece of software. In addition, it requires users to configure their browsers to use the newly installed proxy server. One could argue that our `GI` tags can be replaced with an existing `IFRAME` tag, which is supported by recent versions of popular Web browsers, such as Mozilla, Galeon, and Internet Explorer. Components included in the page with the `IFRAME` tags are fetched and cached by these browsers separately from the other page components. Unfortunately, rendering of `IFRAME` tags by these browsers is not quite seamless. Not only can one tell where the `IFRAME` components are on the page, scrolling and navigation sometimes breaks when `IFRAME` tags are used. In addition, `IFRAME` tags can be used to include only deterministically changing objects, those that can be explicitly marked as uncacheable or assigned an explicit expiration time, because browsers currently do not understand CCC commands.

One could also suggest attaching JavaScript code that performs page assembly at the browser to each MONARCH-enabled page. Once a JavaScript-enabled browser loads a Web page with the `GI` tags, the attached JavaScript code runs, traverses the Document Object Model (DOM) of the page, and finds all occurrences of the `GI` tags. The security model of JavaScript prohibits such code from opening connections to remote servers. A way around this limitation would be for the JavaScript code to create a hidden `IFRAME` for each `GI` tag and assign the source of the `GI` tag to the `src` attribute of the `IFRAME`, effectively forcing the browser to load the respective components into the proper hidden `IFRAME`. The JavaScript code can then manipulate the nodes in the DOM tree and remove all `IFRAME` nodes, making their children part of the main document. The described scheme sounds fairly complicated

and requires each page to include JavaScript code capable of such assembly. Even worse, JavaScript cannot handle caching of these components. However, given that the source code for the Mozilla Web browser is available to the general public, we plan on investigating the possibility of building the MONARCH functionality directly into that browser.

### 7.3.9 Applying Ideas in MONARCH to non-HTML Content

The ideas used by the MONARCH approach are not specific to the Web. However, our discussions focused on the application of these ideas to the Web domain and to HTML pages in particular. We plan on broadening our experimentation by applying our ideas to other types of content. For example, one possible direction of research is applying the ideas in MONARCH to the wireless Web. Since wireless devices are often hand-held and thus are small in size, and the nature of wireless access to the network is quite different from wired access, the language used to mark up wireless content, WML, offers capabilities that differ from those offered by HTML. We plan to study these differences and examine how they can be exploited in conjunction with the ideas in MONARCH for improved object management in wireless environments. One such difference between WML and HTML is the notion of card elements, introduced by WML [97]. Card elements specify fragments of the document body, allowing a larger page to be broken up into smaller parts. These cards could be treated as components, support for which is readily available in wireless browsers. We also plan to examine a new multimedia standard for Television and Web environments, MPEG-4 [45], which is designed to integrate several types of objects to create multimedia. It allows a composition of natural content, such as recordings of people or still objects, and synthetic content, such as synthesized voice and animated 3D models.

The ideas in MONARCH can also be applied to large scale distributed simulations. One example of such simulations is on-line computer games. Modern computer games involve sophisticated virtual worlds with complex interaction between numerous simulated objects—buildings, people, wizards, monsters, weapons. These objects have different cha-

racteristics, including change characteristics. Maintaining strong consistency in distributed computer games is important, as successes and failures of game players, and even realism of the game itself, depend on consistency. A player locates and shoots a virtual opponent just to discover that her view of the virtual world was inconsistent with the master representation at the time, and the opponent actually killed her. We plan on investigating how ideas in MONARCH can be applied to maintaining consistency for objects in distributed computer games.

### 7.3.10 Methodology for Active Content Collection

In this dissertation, we collected snapshots of content from real Web sites for evaluation of various cache consistency policies. We collected the home page, up to three static links, and up to three transient links from each Web site in our set of sites. We plan to further explore, refine, and validate our methodology for active content collection. One direction is to investigate the effect of alternate criteria for picking the number and type of pages at a site for study. We are also interested in determining which pages at a site are actually requested by users. Studying such pages would be most useful. We could examine available proxy logs and select those URLs that are pointing to the sites that we intend to study. We also plan to apply our methodology to sites for which server logs are available and study the effect of content collection intervals and the accuracy of the collected data.

## 7.4 Summary

In this chapter, we summarized many contributions produced by this dissertation, pointed out why we believe our contributions are important, and discussed their lifetime. We also listed and discussed a few ideas for future work. Some of the ideas, such as the selective content assembly and assembling customized content, extend MONARCH's functionality. Other ideas, such as adding MONARCH to end clients, extend MONARCH's reach. We also proposed investigating how the ideas used by MONARCH can be applied to non-HTML



content and other domains, such as multimedia. We believe exploring these directions may result in new extensions to MONARCH and may lead to new applications of the ideas in MONARCH.

## Bibliography

- [1] 100hot.com. <http://www.100hot.com>.
- [2] Akamai. <http://www.akamai.com>.
- [3] The Apache Server Project. <http://www.apache.org>.
- [4] Martin Arlitt and Tai Jin. A Workload Characterization Study of the 1998 World Cup Web Site. *IEEE Network*, May/June 2000.
- [5] Gaurav Banga, Fred Douglass, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *USENIX Technical Conference*, pages 289–303, Anaheim, California, USA, January 1997.
- [6] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. Hypertext Transfer Protocol—HTTP/1.0. RFC 1945, May 1996.
- [7] Manish Bhide, Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy. Adaptive Push-Pull: Disseminating Dynamic Web Data. *IEEE Transactions on Computers*, 51(6):652–668, June 2002.
- [8] Bills.com. <http://www.bills.com>.
- [9] Adam D. Bradley and Azer Bestavros. Basis Token Consistency: Extending and Evaluating a Novel Web Consistency Algorithm. In *Proceedings of the Workshop on Caching, Coherence, and Consistency*, New York, NY, June 2002.
- [10] Adam D. Bradley and Azer Bestavros. Basis Token Consistency: Supporting Strong Web Cache Consistency. In *Proceedings of the IEEE Globecom*, Taipei, Taiwan, November 2002.
- [11] Thomas P. Brisco. DNS support for load balancing. RFC 1794, April 1995.
- [12] CacheFlow Inc. <http://www.cacheflow.com>.
- [13] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching dynamic contents (objects) on the Web. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, England, September 1998.

- [14] Vincent Cate. Alex - a Global Filesystem. In *Proceedings of the USENIX File Systems Workshop*, pages 1–12, May 1992.
- [15] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable and highly available system for serving dynamic data at frequently accessed web sites. In *ACM/IEEE Supercomputing*, Orlando, Florida, USA, November 1998. ACM/IEEE.
- [16] Jim Challenger and Arun Iyengar. Distributed cache manager and API. Technical Report RC 21004, IBM Research Division, October 1997.  
<http://www.research.ibm.com/people/i/iyengar/cache-api.ps>.
- [17] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of the IEEE Infocom '99 Conference*, New York, NY, March 1999. IEEE.
- [18] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic web content. In *Proceedings of the IEEE Infocom 2000 Conference*, Tel Aviv, Israel, March 2000. IEEE.
- [19] Martin Cieslak and David Forster. Web Cache Coordination Protocol v1.0. Expired Internet Draft draft-ietf-wrec-web-pro-00.txt, June 1999.  
<http://www.ietf.org/internet-drafts/draft-ietf-wrec-web-pro-00.txt>.
- [20] Edith Cohen and Haim Kaplan. Refreshment policies for web content caches. *Computer Networks*, 38(6):795–808, April 2002.
- [21] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. In *ACM SIGCOMM'98 Conference*, September 1998.
- [22] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Efficient Algorithms for Predicting Requests to Web Servers. In *Proceedings of the IEEE Infocom '99 Conference*, New York, NY, March 1999. IEEE.
- [23] Ian Cooper, Ingrid Melve, and Gary Tomlinson. Internet web replication and caching taxonomy. Internet Draft draft-ietf-wrec-taxonomy-05.txt, July 2000.  
<http://www.wrec.org/Drafts/draft-ietf-wrec-taxonomy-05.txt>.
- [24] John Dille. The Effect of Consistency on Cache Response Time. *IEEE Network*, May/June 2000.
- [25] John Dille, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally Distributed Content Delivery. *IEEE Internet Computing*, 6(5), September/October 2002.
- [26] Fred Douglass, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the World Wide Web. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

- [27] Fred Douglass, Antonio Haro, and Michael Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, USA, December 1997.
- [28] Venkata Duvvuri, Prashant Shenoy, and Renu Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of the IEEE Infocom 2000 Conference*, Tel Aviv, Israel, March 2000.
- [29] Edge Side Includes. <http://esi.org/>.
- [30] Alan Emtage and Peter Deutsch.archie—an electronic directory service for the Internet. In *Proceedings of the Winter USENIX Conference*, pages 93–110, January 1992.
- [31] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. RFC 2616, June 1999.
- [32] Galeon Web Browser. <http://galeon.sourceforge.net/>.
- [33] Paul Gauthier, Josh Cohen, Martin Dunsmuir, and Charles Perkins. Web Proxy Auto-Discovery Protocol. Expired Internet Draft draft-ietf-wrec-wpad-01.txt, July 1999. <http://www.wrec.org/Drafts/draft-ietf-wrec-wpad-01.txt>.
- [34] Steven Glassman. A caching relay for the World Wide Web. In *Proceedings of the First International World Wide Web Conference*, Geneva, Switzerland, May 1994.
- [35] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 202–210, December 1989.
- [36] James Gwertzman and Margo Seltzer. World-Wide Web cache consistency. In *Proceedings of the USENIX Technical Conference*, pages 141–152. USENIX Association, January 1996.
- [37] Martin Hamilton, Alex Rousskov, and Duane Wessels. Cache digest specification - version 5. Online Publication, December 1998. <http://squid.nlanr.net/CacheDigest/cache-digest-v5.txt>.
- [38] Barron C. Housel and David B. Lindquist. WebExpress: a system for optimizing Web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking (MOBICOM)*, pages 108–116, November 1996.
- [39] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

- [40] Arun Iyengar and Jim Challenger. Improving web server performance by caching dynamic data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, USA, December 1997.
- [41] Arun Iyengar and Jim Challenger. Data Update Propagation: A method for determining how changes to underlying data affect cached objects on the Web. Technical Report RC 21093(94368), IBM Research Division, February 1998.  
<http://www.research.ibm.com/people/i/iyengar/dup.ps>.
- [42] Van Jacobson. How to Kill the Internet (viewgraphs). In *ACM SIGCOMM'95 Middleware Workshop*, Cambridge MA, USA, August 1995.  
<ftp://ftp.ee.lbl.gov/talks/vj-webflame.pdf>.
- [43] Brewster Kahle and Art Medlar. An Information System for Corporate Users: Wide Area Information Servers. *ConneXions—The Interoperability Report*, 5(11), November 1991.
- [44] Keynote Business 40 Internet Performance Index.  
<http://www.keynote.com/measures/business/business40.html>.
- [45] Rob Koenen. MPEG-4: a Powerful Standard for Use in Web and Television Environments. In *Workshop on the Television and the Web*, Sophia-Antipolis, France, June 1998. World Wide Web Consortium.  
<http://www.w3.org/Architecture/1998/06/Workshop/paper26/>.
- [46] D.G. Korn and K.-P. Vo. Engineering a Differencing and Compression Data Format. In *Proceedings of Usenix'2002*. USENIX, 2002.
- [47] Balachander Krishnamurthy and Jennifer Rexford. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching and Traffic Measurement*. Addison Wesley, 2001.
- [48] Balachander Krishnamurthy and Craig E. Wills. Study of piggyback cache validation for proxy caches in the World Wide Web. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [49] Balachander Krishnamurthy and Craig E. Wills. Piggyback server invalidation for proxy cache coherency. In *Proceedings of the Seventh International World Wide Web Conference*, pages 185–193, Brisbane, Australia, April 1998.
- [50] Balachander Krishnamurthy and Craig E. Wills. Analyzing Factors That Influence End-to-End Web Performance. In *Proceedings of the Ninth International World Wide Web Conference*, Amsterdam, Netherlands, April 2000.
- [51] Balachander Krishnamurthy, Craig E. Wills, and Yin Zhang. Preliminary Measurements on the Effect of Server Adaptation for Web Content Delivery. In *Proceedings of the Internet Measurement Workshop, Short abstract*, Marseille, France, November 2002.

- [52] Thomas M. Kroeger, Darrel D.E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [53] Steven Levy. The new digital galaxy. *Newsweek*, pages 56–62, May 1999.
- [54] Dan Li, Pei Cao, and Mike Dahlin. WCIP: Web Cache Invalidation Protocol. Internet Draft. <http://www.wrec.org/Drafts/draft-danli-wrec-wcip-00.txt>.
- [55] Jussara Almeida Li Fan, Pei Cao and Andrei Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *ACM SIGCOMM'98 Conference*, September 1998.
- [56] Chengjie Liu and Pei Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997.
- [57] Ari Luotonen and Kevin Altis. World Wide Web proxies. In *Proceedings of the First International World Wide Web Conference*, Geneva, Switzerland, May 1994.
- [58] Peter Marshall. WAIS:The Wide Area Information Server or Anonymous What??? In *Proceedings of the Net Conference*, May 1992.  
<ftp://ftp.uwo.ca/usr/doc/wais/paper.ps.Z>.
- [59] Mason. <http://www.masonhq.com>.
- [60] Media Metrix. <http://www.mediametrix.com>.
- [61] Paul Mockapetris. Domain names—concepts and facilities. RFC 1034, November 1987.
- [62] Paul Mockapetris. Domain names—implementation and specification. RFC 1035, November 1987.
- [63] Jeffrey C. Mogul. Errors in timestamp-based HTTP header values. Technical Report 99/3, Compaq Western Research Laboratory, December 1999.  
<http://www.research.digital.com/wrl/techreports/abstracts/99.3.html>.
- [64] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *ACM SIGCOMM'97 Conference*, September 1997.
- [65] Jeffrey C. Mogul, Balachander Krishnamurthy, Fred Douglis, Anja Feldmann, Yaron Y. Golland, Arthur van Hoff, and Daniel M. Hellerstein. Delta encoding in HTTP. RFC 3229, January 2002.
- [66] David Mosberger and Tai Jin. httpperf – a tool for measuring web server performance. In *Workshop on Internet Server Performance*, Madison, Wisconsin USA, June 1998.

- [67] Mozilla Web Browser. <http://www.mozilla.org/>.
- [68] ftp.mozilla.org mirrors. <http://www.mozilla.org/mirrors.html>.
- [69] Microsoft Ad in U.S.News and World Report, Special Issue, Volume 127, Number 25, December 1999.
- [70] Erich M. Nahum. WWW Workload characterization work at IBM Research. In *Web Characterization Workshop*, Cambridge, MA, November 1998. World Wide Web Consortium.
- [71] Netscape. Navigator proxy auto-config file format, March 1996.  
<http://www.netscape.com/eng/mozilla/2.0/relnotes/demo/proxy-live.html>.
- [72] Anoop Ninan, Purushottam Kulkarni, Prashant Shenoy, Krithi Ramamritham, and Renu Tewari. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. In *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, May 2002.
- [73] NLANR Proxy Server Logs. <http://www.ircache.net/Traces/>.
- [74] Katia Obraczka, Peter B. Danzig, and Shih-Hao Li. Internet resource discovery services. *IEEE Computer*, September 1993.
- [75] PHP Hypertext Preprocessor. <http://www.php.net>.
- [76] Guillaume Pierre, Maarten van Steen, and Andrew S. Tanenbaum. Dynamically Selecting Optimal Distribution Strategies for Web Documents. *IEEE Transactions on Computers*, 51(6):637–651, June 2002.
- [77] Michael Rabinovich and Oliver Spatscheck. *Web Caching and Replication*. Addison Wesley, 2002.
- [78] Pablo Rodriguez, Christian Spanner, and Ernst W. Biersack. Web caching architectures: hierarchical and distributed caching. In *Proceedings of the 4th International Web Caching Workshop*, San Diego, CA, March/April 1999.
- [79] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Conference*, pages 119–130, June 1985.
- [80] World Wide Web Consortium, Synchronized Multimedia.  
<http://www.w3.org/AudioVideo/>.
- [81] Ben Smith, Anurag Acharya, Tao Yang, and Huican Zhu. Exploiting Result Equivalence in Caching Dynamic Web Content. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, USA, October 1999. USENIX Association.

- [82] Streamline.com. <http://www.streamline.com>.
- [83] Renu Tewari, Thirumale Niranjan, and Srikanth Ramamurthy. WCDP: A Protocol for Web Cache Consistency. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*, Boulder, CO, August 2002.
- [84] Vinod Valloppillil and Keith W. Ross. Cache Array Routing Protocol v1.0. Expired Internet Draft draft-vinod-carp-v1-03.txt, February 1998.  
<http://ircache.nlanr.net/Cache/ICP/carp.txt>.
- [85] Fred Vogelstein and William J. Holstein. Adventures in e-shopping. *U.S. News and World Report, Special Issue*, pages 34–35, December 1999.
- [86] Abigail Walch. The way we live now: In figures. *Newsweek, Special Edition*, page 67, December–February 1999–2000.
- [87] Jia Wang. A survey of Web caching schemes for the Internet. *Computer Communication Review*, 29(5):36–46, October 1999.
- [88] Webvan.com—The World’s Market at Your Doorstep. <http://www.webvan.com>.
- [89] Duane Wessels. Squid Internet Object Cache. <http://squid.nlanr.net/Squid/>.
- [90] Duane Wessels. ICP home page, July 1999.  
<http://ircache.nlanr.net/Cache/ICP/>.
- [91] Duane Wessels and K. Claffy. Application of Internet Cache Protocol (ICP), version 2. RFC 2187, September 1997.
- [92] Duane Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2. RFC 2186, September 1997.
- [93] Stephen Williams, Marc Abrams, Charles R. Standbridge, Ghaleb Abdulla, and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the ACM SIGCOMM Conference*, pages 293–305, August 1996.
- [94] Craig E. Wills and Mikhail Mikhailov. Examining the cacheability of user-requested Web resources. In *Proceedings of the 4th International Web Caching Workshop*, San Diego, CA, March/April 1999.
- [95] Craig E. Wills and Mikhail Mikhailov. Towards a better understanding of Web resources and server responses for improved caching. In *Proceedings of the Eighth International World Wide Web Conference*, Toronto, Canada, May 1999.
- [96] Craig E. Wills, Mikhail Mikhailov, and Hao Shang. *N* for the Price of 1: Bundling Web Objects for More Efficient Content Delivery. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001.
- [97] Wireless Markup Language Version 2 Specification. White Paper, Wap Forum, September 2001. <http://www.wapforum.org>.



- 
- [98] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Using leases to support server-driven consistency in large-scale systems. In *Proceedings of the 18th International Conference on Distributed Systems*. IEEE, May 1998.
- [99] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Arun Iyengar. Engineering web cache consistency. *ACM Transactions on Internet Technology*, 2(3):224–259, August 2002.
- [100] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, USA, October 1999. USENIX Association.
- [101] Jian Yin, Mike Dahlin, Lorenzo Alvisi, Calvin Lin, and Arun Iyengar. Engineering server driven consistency for large scale dynamic web services. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001.