CONTENT CONSISTENCY FOR WEB-BASED INFORMATION RETRIEVAL

CHUA CHOON KENG

NATIONAL UNIVERSITY OF SINGAPORE

2005

CONTENT CONSISTENCY FOR

WEB-BASED INFORMATION RETRIEVAL

CHUA CHOON KENG

(B.Sc (Hons.), UTM)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgments

I would like to express sincere appreciation to my supervisor, Associate Professor Dr. Chi Chi Hung for his guidance throughout my research study. Without his dedication, patience and precious advices, my research would not have completed smoothly. Not only did he offered me academic advices, he also enlightened me on the true meaning of life and that one must always strive for the highest – "think big" in everything we do.

In addition, special thanks to my colleagues especially Hong Guang, Su Mu, Henry and Jun Li for their friendship and help in my research. They have made my days in NUS memorable.

Finally, I wish to thank my wife, parents and family for their support and for accompanying me through my ups and downs in life. Without them, I would not have made this far. Thank you.

Table of Contents

List of Tables

List of Figures

Summary

In this thesis, we study the inconsistency problems in web-based information retrieval. We then propose a novel content consistency model and a possible solution to the problem.

In traditional data consistency, 2 pieces of data are considered consistent if and only if they are bit-by-bit equivalent. However, due to the unique operating environment of the web, data consistency cannot adequately address consistency of web contents. Particularly, we would like to address the problems of correctness of content delivery functions, and reuse of pervasive content.

Firstly, we redefine content as entity that consists of object and attributes. Later, we propose a novel content consistency model and introduce 4 content consistency classes. We also show the relationship and implications of content consistency to web-based information retrieval. In contrast to data consistency, "weak" consistency in our model is not necessarily a bad sign.

To support our content consistency model, we present 4 case studies of inconsistency in the present internet.

The first case study examines the inconsistency of replicas and CDN. Replicas and CDN are usually managed by the same organization, making consistency maintenance easy to perform. In contrast to common beliefs, we found that they suffer severe inconsistency problems, which results in consequences such as unpredictable caching behaviour, performance loss, and content presentation errors.

In the second case study, we investigate the inconsistency of web mirrors. Even though mirrored contents represent an avenue for reuse, our results show that many mirrors suffer inconsistency in terms of content attributes and/or objects.

The third case study analyzes the inconsistency problem of web proxies. We found that some web proxies cripple users' internet experience, as they do not comply to HTTP/1.1.

In the forth case study, we investigate the relationship between contents' time-to-live (TTL) and their actual lifetime. Results show that most of the time, TTL does not reflect the actual content lifetime. This leads to either content staleness or performance loss due to unnecessary revalidations.

Lastly, to solve the consistency problems in web mirrors and P2P, we propose a solution to answer "where to get the right content" based on a new ownership concept. The ownership scheme clearly defines the roles of each entity participating in content delivery. This makes it easy to identify the owner of content whom users can check consistency with. Protocol extensions have also been developed and implemented to support ownership in HTTP/1.1 and Gnutella.

Chapter 1

INTRODUCTION

1.1 Background and Problems

Web caching is a mature technology to improve the performance of web content delivery. To reuse a cached content, the content must be *bit-by-bit equivalent* to the origin (known as data consistency). However, since the internet is getting heterogeneous in terms of user devices and preferences, we argue that traditional data consistency cannot efficiently support pervasive access. 2 primary problems are yet to be addressed: 1) correctness of functions, and 2) reuse of pervasive content. In this thesis, we study a new concept termed *content consistency* and show how it helps to maintain the correctness of functions and improve the performance of pervasive content delivery.

Firstly, there lies a fundamental difference between "data" and "content". Data usually refers to entity that contains a single value, for example, in computer architecture each memory location contains a word value. On the other hand, content (such as a web page) contains more than just data; it also encapsulates *attributes* to administrate various functions of content delivery.

Unfortunately, present content delivery only considers the consistency of data but not attributes. Web caching, for instance, is an important function for improving performance and scalability. It relies on caching information such as expiry time, modification time and other caching directives, which are included in attributes of web contents (HTTP headers) to function correctly. However, since content may traverse through intermediaries such as caching proxies, application proxies, replicas and mirrors, the HTTP headers users receive may not be the original. Therefore, instead of using HTTP headers as-is, we question about the *consistency of attributes*. This is a valid concern because the attributes directly determine whether the functions will work properly and they may also affect the performance and efficiency of content delivery. Besides web caching, attributes are also used for controlling the presentation of content and to support extended features such as rsync in HTTP [1], server-directed transcoding [2], WEBDEV [3], OPES [4], privacy & preferences [5], Content-Addressable Web [6] and many other extensions. Hence, the magnitude of this problem should not be overlooked.

Secondly, in pervasive environments, contents are delivered to users in their best-fit presentations (also called variants or versions) for display on heterogeneous devices [7, 8, 9, 10, 11, 12, 13, 2]. As a result, users may get presentations that are *not* bit-by-bit equivalent to each other, yet all these presentations can be viewed as "consistent" in certain situations. Data consistency, which refers to bit-to-bit equivalence, is too strict and cannot yield effective reuse if applied to pervasive environment. In contrast to data consistency, our proposed content consistency *does not* require objects to be bit-by-bit equivalent. For example, 2 music files of different quality can be considered consistent if the user uses a low-end device for playback.

Likewise, 2 identical images except with different watermarks can be considered as consistent if users are only interested in the primary content of the image. This relaxed notion of consistency increases reuse opportunity, and leads to better performance in pervasive content delivery.

1.2 Examples of Consistency Problems in the Present Internet

1.2.1 Replica/CDN

Many large web sites replicate contents to multiple servers (replicas) to increase availability and scalability. Some maintain their server cluster in-house while others may employ services from Content Delivery Networks (CDN).

When users request for replicated web content, a traffic redirector or load balancer dynamically forwards the request to the best available replica. Subsequent requests from the same user may not be served by the replica initially responded.

No matter how many replica are in used, they are externally and logically viewed as a single entity. Users aspect them to behave like a single server. By creating multiple copies of web content, a significant challenge arises on how to maintain all the replicas so that they are consistent with each other. If content consistency is not addressed appropriately, replication can bring more harm than good.

1.2.2 Web Mirrors

Web mirrors are used to offload the primary server, to increase redundancy and to improve access latency (if mirrors are closer to users). They differ from replication/CDN in that mirrored web contents use name spaces (URLs) that are different from the original.

Mirrors can become inconsistent due to 3 reasons. Firstly, the content may become outdated due to infrequent update or slack maintenance. Secondly, mirrors may modify the content. An example is shown in Figure 1 where a HTML page is stripped off redundant white spaces and comments. From data consistency point of view, the mirrored page has become inconsistent, but what if there is no visual or semantic change? Thirdly, HTTP headers are usually ignored during mirroring, which results in certain functions to fail or work inefficiently.

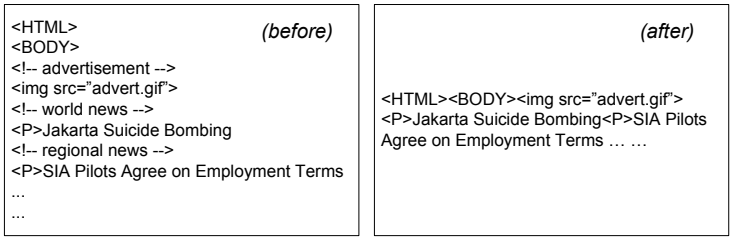

Figure 1: A HTML Page Before and After Removing Extra Spaces and Comments

We see web mirrors as an avenue for content reuse, however content inconsistency remains a major obstacle. Content attributes and data could be modified for both good and bad reasons, making it difficult to decide on reusability. On one hand, we have to offer mirrors incentives to do mirroring, such as by allowing them to include their own advertisements. On the other

hand, inappropriate header or data modification has to be addressed. This problem is similar to that of OPES.

Another notable problem is that there is no clear distinction of the roles between mirror and server. Presently, users treat mirrors as if they are the origin servers, and thus perform some functions inappropriately at mirrors (eg: validation is performed at mirror where it should be performed at origin server instead). The problem arises from the fact that HTTP lacks the concept of content ownership. We will study how ownership can address this problem.

1.2.3 Web Caches

Caching proxies are widely deployed by ISPs and organizations to improve latency and network usage. While it has proved to be an effective solution, there are certain consistency issues about web caches. We shall discuss 3 in this section.

Firstly, there is some mismatch between content lifetime and time-to-live (TTL) settings. Content lifetime refers to the period between the content's generation time and its next modification time. This is the period where the content can be cached and reused without revalidation. Content providers assign TTL values to indicate how long contents can be cached. In the ideal case, TTL should reflect content lifetime, however in most cases it is impossible to known content lifetime in advance. If TTL is set lower than the actual lifetime, cached contents become stale. On the contrary, setting a TTL higher than the actual lifetime causes redundancy in performing cache revalidations.

Secondly, different caching proxies may have conflicting configurations which can results in consistency problems if these proxies are chained together. It is quite common for caching proxies to form a hierarchical structure. For instance, ISP proxies form the upstream of organization proxies which in turn become the upstream of departmental proxies. Wills et al. [14] reveal that more than 85% of web contents do not have explicit expiry dates, which can cause problems for proxies in cache hierarchies. HTTP/1.1 states that proxies can use heuristics to cache contents without explicit expiry dates. However, if proxies in the hierarchy use different heuristics or have different caching policies, transparency of cache semantics is lost. For example, if a departmental proxy was configured to cache these contents for 1 hour, users would not expect them to be stale for more than 1 hour. This expectation would not be met if the upstream proxy was configured to cache these contents for a longer duration.

Thirdly, compliance of web proxy caches is an issue of concern. For many users behind firewall, proxies are the only way to access the internet. Since there is no alternative access to the internet (such as direct access) that bypasses inconsistent proxies, it becomes critical that the proxies comply to HTTP/1.1. Proxies should ensure that they serve contents that are consistent with origins. We will further study this issue in this thesis.

1.2.4 OPES

OPES is a framework for deploying application intermediaries in the network [4]. It is viewed as an important infrastructural component to support pervasive access and to provide content services. However, since OPES processors can modify requests and contents that pass through

them, they may silently change the semantic of content and affect various content delivery functions.

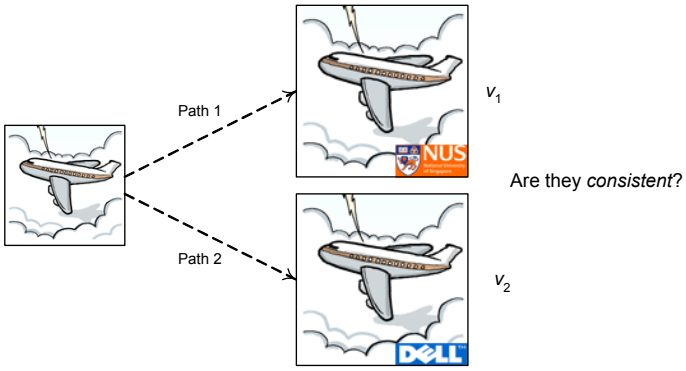

Figure 2: OPES Creates 2 Variants of the Same Image

The existence of OPES processors creates multiple variants for the same content. An example is shown in Figure 2 where an image is delivered to 2 users on different path. On each path, an OPES processor inserts a small logo to the image, resulting in 2 variants of the same image v_1 and v_2. We ask:

- Is v_1 consistent with v_2 (and vice versa)? No - from data consistency's point of view.
- Suppose the caching proxy in path 2 found a cached copy of v_1 (as in the case if there is peering relationship between proxies on path 1 and path 2). Can we use v_1 to serve requests of v_2? If users are only interested in the "main" content of the image, then the system should use v_1 and v_2 alternatively.

OPES requires all operations performed to be logged in an OPES trace and include it in HTTP headers. However, the trace only tells us what has been done on the content, and not how to

reuse different versions or variants of the same content. Challenges in achieving reuse include how and when to treat contents as consistent (content consistency), and the necessary protocol/language support to realize the performance improvement.

1.3 Contributions

The author has made contributions in the following 3 aspects.

Content Consistency Model – Due to the unique operating environment of the web, we redefine the meaning of content as entity that consists of object and attributes. With the new definition of content, we propose a novel content consistency model and introduce 4 content consistency classes. We also show the relationship and implications of content consistency to web-based information retrieval.

Comprehensive Study of Inconsistency Problems in the Present Internet - To support our model, we highlight inconsistency problems in the present internet with 4 comprehensive case studies. The first examines the prevalence of inconsistency problem in replicas of server farms and CDN. The second studies the inconsistency of mirrored web contents. The third analyzes the inconsistency problem of web proxies while the forth studies the relationship between contents' time-to-live (TTL) and their actual lifetime. Results from the 4 case studies show that consistency should not only base on data; attributes are of equal importance too.

An Ownership-based Solution to Consistency Problem - To solve the consistency problems in web mirrors and P2P, we propose a solution to answer "where to get the right

content" based on a new ownership concept. The ownership scheme clearly defines the roles of each entity participating in content delivery and makes it easy to identify the source or owner of content. Protocol extensions have been developed and implemented to support ownership in HTTP/1.1 and Gnutella/0.6.

1.4 Organization

The rest of the thesis is organized as follows. Chapter 2 reviews existing web and P2P consistency models. We also survey some work related on HTTP headers. In chapter 3, we present the content consistency model and show its implication to web-based information retrieval. Chapters 4 to 7 examine in details the inconsistency problems of replica/CDN, web mirrors, web proxies and content TTL/lifetime respectively. To address the content consistency problem in web mirrors and P2P, an ownership-based solution is proposed in chapter 8. Chapter 9 describes the protocol extensions and system implementation of ownership in web and Gnutella. Finally, Chapter 10 concludes the thesis with a summary and some proposals for future work.

Chapter 2

RELATED WORK

2.1 Web Cache Consistency

2.1.1 TTL

HTTP/1.1 [15] supports basic consistency management using TTL (time-to-live) mechanism. Each content is assigned a TTL value by the server. When TTL time has elapsed, the content is marked as invalid and clients must check with the origin server for an updated copy. This method works best if the next update time of content is known a priori (good for news website). However, this is not the case for most other contents; content providers simply do not know when contents will be updated. As a result, TTL values are usually assigned conservatively (by setting a low TTL) or arbitrarily, creating unnecessary polling with origin servers or staleness.

To overcome these limitations, two variations of TTL have been proposed. Gwertzman et al. [16] proposed the adaptive TTL which is based on the Alex file system [17]. In this approach, the validity duration of a content is the product of its age and an update threshold (expressed in

percentage). The authors show that good results can be obtained by fine-tuning the update threshold by analyzing content modification log. Performing this tuning manually will only result in suboptimal performance. Another approach to improve the basic TTL mechanism is to use an average TTL. The content modification log is analyzed to determine the average age of contents. The new TTL value is set to be the product of content average age and an update threshold (expressed in percentage). Both methods improve the performance of the basic TTL scheme, but do not overcome the fundamental limitation: unnecessary polling or staleness.

2.1.2 Server-Driven Invalidation

Weak consistency guarantee offered by TTL may not be sufficient for certain applications, such as websites with many dynamic or frequently changing objects. As a result, server-driven approach was proposed to offer strong consistency guarantee [18]. Server-driven approach works as follows. Clients cache all response received from server. For each new object (object that has not be requested before) delivered to a client, the server send an "object lease" which will expire some time in the future. The clients can safely use an object as long as the associated object lease is valid. If the object is later modified, the server will notify all clients who hold a valid object lease. This requires the server to maintain states such as which client has which object leases. The number of states grows with the number of objects and connecting clients. An important issue that determines the feasibility of server-driven approach is its scalability. Much of further research has focused on this direction.

2.1.3 Adaptive Lease

An important parameter for the lease algorithm is the lease duration. Two overheads imposed by leases are state maintained by server and control message overhead. Having short lease duration reduces the server state overhead but increases control message overhead and vice versa. Duvvuri et al. [19] proposed adaptive lease which intelligently compute the optimal duration of leases to balances these tradeoffs. By using either the state space at the server or the control messages overhead as the constraining factor, the optimal lease duration can be computed. If the lease duration is computed dynamically using the current load, this approach can react to load fluctuations.

2.1.4 Volume Lease

Yin et al. [20] proposed volume lease as a way to further reduce the overhead associated with leases. A problem observed in the basic lease approach is the high overhead in lease renewals. To counter this problem, the authors proposed to group related objects into volumes. Besides an object lease, each object is also associated with a volume lease. A cached object can be used only if both the object lease and the corresponding volume lease have not expired. The duration of volume lease is configured to be much lower than that of object leases. This has the effect of amortizing volume lease renewal overheads over many objects in a volume.

2.1.5 ESI

The Edge Side Include (ESI) is an open standard specification for aggregating, assembling, and delivering web pages at the network edge, enabling greater levels of dynamic content caching [21]. It is observed that for most dynamic web pages, only portions of the pages are really dynamic, the other parts of the pages are relatively static. Thus, in ESI, each web page is decomposed into a page template and several page fragments. Each template or fragment is treated as independent entity; they can be tagged with different caching properties. ESI defines a simple set of markup language that allows edge servers to assemble page templates and fragments into a complete web page before delivering to end users. ESI's server invalidation allows origin servers to invalidate cache entries at CDN surrogates. This allows for tight coherence between origin servers and surrogates. ESI has been endorsed and implemented by many vendors and products including, Akamai, Oracle 9i Application Server and BEA WebLogic.

2.1.6 Data Update Propagation

Many web pages are dynamically generated upon request and are usually marked as uncachable. This causes clients to retrieve them upon every request, increasing server and network resource usage. Challenger et al. [22] proposed the Data Update Propagation (DUP) technique, which maintains data dependence information between cached objects and the underlying data (eg. database) which affect their values in a graph. In this approach, response for dynamic web pages is cached and used to satisfy subsequent requests. This eliminates the need to invoke

server programs to generate the web page. When the underlying data changes, their dependent cache entries are invalidated or updated.

2.1.7 MONARCH

MONARCH is proposed to offer strong consistency without having servers to maintain per-client state [23]. Majority of web pages consist of multiple objects and retrieval of all objects is required for proper page rendering. The authors argue that ignoring relationship between page container and page objects is a lost opportunity. The approach achieves strong consistency by examining the objects composing a web page, selecting the most frequently changing object on that page and having the cache request or validate that object on every access. The goal of this approach is to offer strong consistency for non-deterministic objects (objects that change at unpredictable rate). Traditional TTL approach forces publishers to set conservative TTL in order to achieve high consistency at the cost of high revalidation overhead. With MONARCH, these objects can be safely cached by exploiting the relationship and change pattern of page container and objects.

2.1.8 Discussion

All web cache consistency mechanisms attempt to solve the same problem – to ensure cached objects are consistent with the origin. Broadly, existing approaches can be categorized into pull-based solutions which provide weak consistency guarantees, and server-based invalidation/update solutions which provide strong consistency guarantees.

Existing approaches only concern in whether users get the most updated object. They ignore the fact that many other functions rely on HTTP headers to work correctly. A consistency model is incomplete if content attributes (HTTP headers) are not considered. For example, suppose a cached object is consistent with the origin but the headers are not, which results in caching and presentation errors. In this case, do we still consider them as consistent?

The web differs from other distributed systems in that it does not have a predefined set of content attributes. Even though HTTP/1.1 has well defined headers, it is extensible that many new headers have been proposed or implemented to support new features. The set of headers will only grow with time, thus consistency of content attributes should not be overlooked.

It might be tempting to think why not just extend the existing consistency models to treat each object and their attributes as a single content. This way we can ensure that attributes are also consistent with the origin. The problem is that even in HTTP/1.1, there are many constraints or logics governing headers, some headers must be maintained end-to-end, some hop-by-hop, while some maybe calculated based on certain formula. In some cases, the headers of 2 contents maybe different but the contents are still consistent. Even if we can incorporate all the constraints in HTTP/1.1 into the consistency model, we still have problem supporting present and future HTTP extensions, each having its own constraints and logics.

2.2 Consistency Management for CDN, P2P and other Distributed Systems

Caching and replication creates multiple copies of content, therefore consistency must be maintained. This problem is not limited to the web, many other distributed computing systems also cache or replicate content. Saito et al. [29] did an excellent survey of consistency management in various distributed systems.

Solutions for consistency management in distributed systems share similar objective, but differ in their design and implementation. They make use of their specific system characteristics to make consistency management more efficient. For example, Ninan et al. [24] extended the lease approach for use in CDN, by introducing the cooperative lease approach. Another solution for consistency management in CDN is [30]. On the other hand, solutions available for the web or CDN are inappropriate for P2P as peers can join and leave unexpectedly. Solutions specifically designed for P2P environments include [31, 32].

2.2.1 Discussion

Similar to existing web cache consistency approaches, solutions available for distributed systems treat each object as having an atomic value. They are less appropriate for web content delivery where various functions heavily depend on content attributes to work correctly. In pervasive environments, web contents are served in multiple presentations, which invalidate the assumption that each content contains an atomic value.

2.3 Web Mirrors

Though there are some works related to mirrors, none has focused on consistency issues.

Makpangou et al. developed a system called Relais [26], which is a replicated directory service that connects a distributed set of caches and mirrors, providing the abstraction of a single consistent, shared cache. Even though it mentioned about reusing mirrors, it did not explain how mirrors are checked for consistency or how they are added to the cache directory. We assume that either the mirrors are hosted internally by the organization (thereby assumed consistent) or they can be any mirror as long as their consistency has been manually checked upon. Furthermore, the mirrors URLs might be manually associated with the origin URLs.

Other work related to mirrors are [25] which examines the performance of mirror servers to aid the design of protocols for choosing among mirror servers, [33, 34] which propose algorithms to access mirror sites in parallel to increase download throughput, and [35, 36] which propose techniques for detecting mirrors to improve search engine results or to avoid crawling mirrored web contents.

2.3.1 Discussion

Many web sites are replicated by third party mirror sites. These mirror sites represent a good opportunity for reuse, but consistency must be adequately addressed first. Unlike CDN, mirrors are operated by many different organizations, thus it would not be easy to make them consistent. Instead of making all the mirrors consistent, we can probably use mirrors as a non-

authoritative download source and provide users with links to the owner (authoritative source) if they want like to check for consistency.

2.4 Studies on Web Resources and Server Responses

Wills et al. [27, 28] study on characterizing information about web resources and server responses that is relevant to web caching. Their data sets include the popular web sites from 100hot.com as well as URLs in NLANR proxy traces. Besides gathering statistics about the rate and nature of changes, they also study the response header information reported by servers. Their results indicate that there is potential to reuse more cached resources than is currently being realized due to inaccurate and nonexistent cache directives.

2.4.1 Discussion

Even though the objective of their study is to understand web resources and server responses to improve caching, they have pointed out some inconsistency problems in server response headers. For example, they noted some web sites with multiple servers have inconsistent ETag or Last-Modified header values. However, their results on the header inconsistency problem are very limited, which motivate us to study this subject in more details.

2.5 Aliasing

Aliasing occurs in web transactions when different request URLs yield replies containing identical data payloads [48]. Existing browsers and proxies perform cache lookups using URLs,

and aliasing can cause redundant payload transfers when the reply payload that satisfies the current request has been previously received but is not cached under the current URL.

The authors found that aliasing accounts for over 36% of bytes transferred in a proxy trace. To counter this problem, they propose to index each cache entry using payload digest, in addition to URL. Before downloading the payload from any server, verify via digest lookup that the cache does not already have it. This ensures only compulsory misses occur; misses due to the namespace cannot.

2.5.1 Discussion

By definition, mirrored web contents are aliased. A problem that has not been addressed yet is that among a set of aliased URL (origin and mirrors), which URL is the origin, or is the "authoritative" copy. In Chapter 8, we propose an ownership approach to solve this problem.

Chapter 3

CONTENT CONSISTENCY MODEL

3.1 System Architecture

Our vision of the future content consistency is depicted in Figure 3.

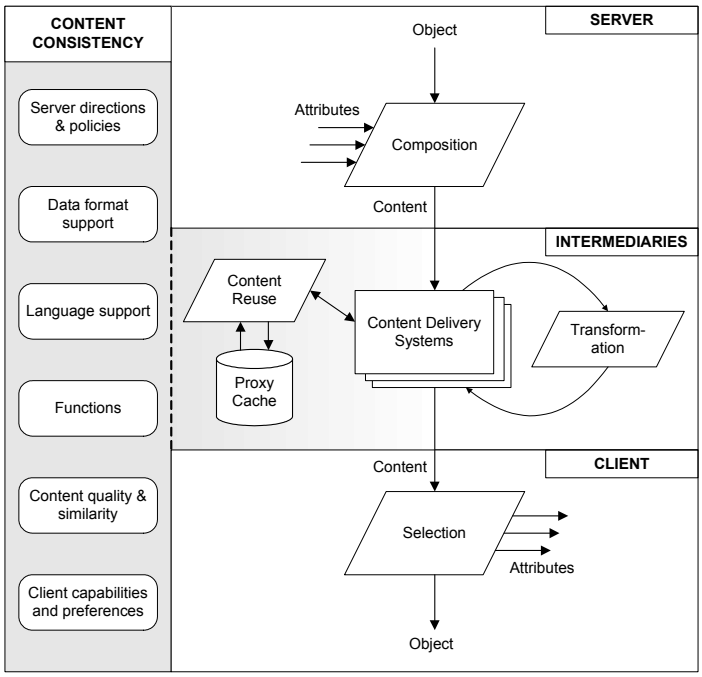

Figure 3: System Architecture for Content Consistency

We begin by describing the pervasive content delivery process in 3 stages: server, intermediaries and client. In stage 1, server *composes* content by associating an object with a set of attributes. *Content* is the unit of information in the content delivery system where object refers to the main data such as image and HTML while attributes are metadata required to perform *functions* such as caching, transcoding, presentation, validation, etc. The number of functions available is infinite, and functions evolve over the time as new requirements emerge.

In stage 2, content travels through zero or more intermediaries. Each intermediary may perform transformations on content to create new *variants* or *versions*. Transformations such as transcoding, translation, watermarking and insertion of advertisements, are operations that change object and/or attributes. To improve the performance of content delivery, intermediaries may cache contents (original and/or variants) to achieve full or partial reuse.

In stage 3, content is received by client. *Selection* is performed to select object and the required attributes from content. The object is then used by user-agents (for display or playback), and functions associated with content are performed. Two extreme cases of selection is to select all available attributes, or to only select attributes of a specific function.

Content consistency is a measurement of how coherent, equivalent, compatible or similar are 2 content. Typically, cached/replicated contents are compared against the original content (at server) to determine whether they can be reused. Content consistency is important for 2 reasons. Firstly, we depend on content consistency to ensure content delivery functions (such as caching & transcoding) perform correctly. Functions are as important as the object itself as they

can affect the performance of content delivery, presentation of content, etc. Secondly, we exploit on content consistency to improve content reuse, especially at intermediaries where significant performance gain can be achieved. To reuse content in pervasive environment, our architecture demands on the necessary data format support (such as scalable data model [37]) and language support. Content consistency also requires information from both ends of content delivery: server and client. For instance, server shall provide its directions and policies on consistency for its contents while client needs to indicate its capabilities and preferences.

3.2 Content Model

Functions require certain knowledge in order to work correctly. We consider a simple representation of knowledge by using attributes. (Although we use the term knowledge, we do not require complex knowledge representation as used in AI/knowledge representation). There exists a many-to-many relationship between function and attribute, that is, each function may require a set of attributes while each attribute may associate with many functions.

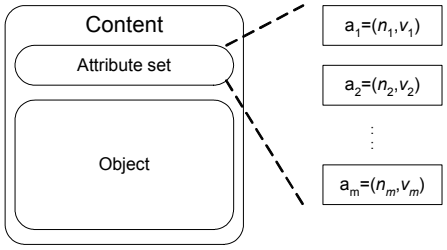

Figure 4: Decomposition of Content

Each object is bundled with relevant attributes to form what we called "content", as shown in Figure 4. Formally, a content C is defined as $C = \{O, A\}$, where O denotes object and A

denotes attribute set. We further divide content into 2 types: *primitive content* and *composite content*, which will be formally defined in section 3.4.

3.2.1 Object

O denotes object of any size. Objects we consider here are application-level data such as text (HTML, plain text), images (JPEG, GIF), movies (AVI, MPEG), etc. Objects are also known as resources, data, body and sometimes files. Our model does not assume or require any format or syntax for the data. We treat data only as an opaque sequence of bytes; no understanding of the semantics of objects is required.

3.2.2 Attribute Set

A denotes the attribute set, where $A = \{a_1, a_2, a_3, ...\}$ and $a_x = (n_x, v_x)$. The attribute set is a set of zero or more attributes, a_x. Each attribute describes a unique concept and is presented in the form of a (n, v) pair. n refers to the name of the attribute (the concept it refers to) while v the value of the attribute. Eg: ("Date", "12 June 2004"), ("Content-Type", "text/html").

We assume that there is no collision of attribute names. That is there will be no 2 attributes having the same name, but describing about different concepts. This is a reasonable assumption and can be achieved in practice by adopting proper naming conventions.

Some application-level object may embed attributes within their payload; for example, many image formats store metadata in the object headers. These internal attributes may be considered for content consistency depending on the application's requirements.

Since an attribute set A may associate with a few functions, we denote this set of functions as $F_A = \{f_1,..., f_n\}$. We can also divide attribute set A into smaller sets according to the functions they serve. We call such an attribute set function-specific attribute set, denoted A^f where the superscript f represents the name of function. Consequently, for any content C, the union of all its function-specific attribute sets is the attribute set A.

Suppose $F_A = \{f_1,..., f_n\}$

Then, $\{A^{f_1} \cup ... \cup A^{f_n}\} = A$

3.2.3 Equivalence

In our discussions, we use the equal sign (=) to describe the relation between contents, objects and attribute sets. The meaning of equivalence in the 3 cases is defined as follows.

When we have $O_1 = O_2$, we mean that O_1 is bit-by-bit equivalent to O_2.

When we have $A_1 = A_2$, we mean that:

$\forall a_x \in A_1, a_x \in A_2$ and vice versa (set equivalence) *or*

A_1 is semantically equivalent to A_2. We do not define the semantics of attributes; we assume they are well defined according to some known rules, protocols or specifications. For example, suppose $A_1 = \{\{Time, 10:00GMT\}, \{Type, text\}\}$ and $A_2 = \{\{Time, 2:00PST\}, \{Type, text\}\}$. Even though the values of time attribute in both attribute sets are not bit-by-bit equivalent, they are semantically equivalent according to the time convention.

Finally, when we have $C_1 = C_2$, we mean that $O_1 = O_2$ and $A_1 = A_2$.

3.3 Content Operations

2 operations are defined for content: selection and union. Selection operation is typically performed when a system wishes to perform a specific function on content while union operation is used to compose content. Both operations may be used in content transformation.

3.3.1 Selection

The selection operation, SEL_{FS}, is an operation to filter a content so that it contains only the object and the attributes of a set of selected functions.

Let there be n selected functions, represented by the set $FS = \{fs_1, ..., fs_n\}$. The selection operation SEL_{FS} on content C is defined as:

$$SEL_{Fs}(C) = SEL_{Fs}(\{O, A\})$$

$$= \{O, A^{fs_1} \cup ... \cup A^{fs_n}\}$$

$$= \left\{ O, \bigcup_{i=1}^{n} A^{fs_i} \right\}$$

3.3.2 Union

The union operation, $\cup$, is an operation to combine m contents $C_1,...,C_m$ into a single content, provided all of them have the same object, that is $O_1 = ... = O_m$.

The union operation $\cup$ on content $C_1,...,C_m$ is defined as:

$$C_1 \cup ... \cup C_m$$

$$= \{O_1, A_1\} \cup ... \cup \{O_m, A_m\}$$

$$= \left\{ O_1, \bigcup_{i=1}^{m} A_i \right\}$$

3.4 Primitive and Composite Content

We classify content into 2 types: primitive content and composite content. Our content consistency model operates on primitive content only, thus it is important we clearly define them before we proceed to content consistency.

> A *primitive content* C^f is a content that contains only object and attributes of a function, where the superscript f denotes the name of the function. This condition can be expressed as $F_A = \{f\}$. Primitive content is also called function-specific content.

A primitive content can be obtained by applying a selection operation on any content C, that is, $C^f = SEL_{\{f\}}(C)$.

> A *composite content*, CC is a content that contains attributes of more than 1 function. This condition can be expressed as $|F_A| > 1$.

There are 2 ways to generate a composite content: selection or union. The first method is to apply selection operation on content C with more than 1 function.

$$CC = SEL_{FS}(C) \text{ where } |FS| > 1$$

The second method is to apply union operation on contents $C_1,...,C_r$, if they contain attributes of more than 1 function.

$$CC = \bigcup_{i=1}^{i=r} C_i \text{ where } r > 1 \text{ and } \exists i, j; 1 \le i, j \le r; i \ne j; F_{A_i} \ne F_{A_j}$$

3.5 Content Consistency Model

Content consistency compares 2 primitive contents: a *subject* S and a *reference* R. It measures how consistent, coherent, equivalent, compatible or similar is the subject to the reference.

Let C_S and C_R represents the set of all subject contents and the set of all reference contents respectively. Content consistency is a function that maps the set of all subject content and reference content to a set of consistency classes:

$$is\_consistent : \{C_S, C_R\} \mapsto \{Sc, Oc, Ac, Wc\}$$

The 4 classes of content consistency are strong, object-only, attributes-only and weak consistency.

For any 2 content $S \in C_S$ and $R \in C_R$, to evaluate the consistency of S against R, both S and R must be primitive content, that is they must only contain attributes of a common function; otherwise, content consistency is undefined. This implies that content consistency addresses only 1 function at a time. To check multiple functions, content consistency is repeated with primitive contents under different functions.

Strong consistency is a strict consistency class. In this consistency class, no entity in the content delivery path (if any) shall modify object or attributes.

Strong Consistency (*Sc*):

S is strongly consistent with R if and only if $O_S = O_R$ and $A_S = A_R$.

If we relax the strong consistency by allowing attributes to be modified, we have the *object-only consistency*.

Object-only Consistency (*Oc*):

S is object-only consistent with R if and only if $O_S = O_R$ and $A_S \neq A_R$.

If we relax the strong consistency by allowing object to be modified, we have the *attributes-only consistency*.

Attributes-only Consistency (*Ac*):

S is attributes-only consistent with R if and only if $O_S \neq O_R$ and $A_S = A_R$.

Finally, if we allow both object and attributes to be modified, we have the *weak consistency*.

Weak Consistency (*Wc*):

S is weakly consistent with R if and only if $O_S \neq O_R$ and $A_S \neq A_R$.

The 4 content consistency classes is a non-overlapping classification: given any subject and reference content, they will map to one and only one of the consistency classes. Each consistency class represents an interesting case for research especially since Oc, Ac & Wc are now common in pervasive content delivery context. Each has its unique characteristics, which poses different implications on functions and content reuse

3.6 Content Consistency in Web-based Information Retrieval

The two parameters for content consistency are the reference content R and the subject content S. Typically, S refers to a cached/mirrored/local content you wish to reuse, while R refers to the content you wish to obtain (usually the original content). Thus you check the consistency of S against R to ensure what you reuse is what you wanted.

3.7 Strong Consistency

In web content delivery, strong consistency is encountered when S is an unmodified cached copy or an exact replica of R. In both cases, $S = R$ and we can directly reuse S without any problem. This is the simplest and also the ideal case of content consistency, thus nothing interesting to study about.

3.8 Object-only Consistency

Object-only consistency occurs when $O_S = O_R$ and $A_S \neq A_R$. This is usually observed at web sites using replica or CDN, and also at mirror sites. Even though objects are consistent, inconsistent attributes can cause functions to perform incorrectly due to incorrect or incomplete attributes. There are 2 problems not addressed for this class of consistency.

Correctness of functions is the first problem we would like to address. In particular, we will investigate:

- why content delivery functions are important and what happens when the functions are not performed correctly
- the prevalence of this problem on the internet

Secondly, we observe that mirrors often act on behalf of owners but are they authorized to do so? Who is the owner of content: the origin server or the mirror? To answer these questions, we need to define the ownership of content and the roles played by owners in content delivery.

3.9 Attributes-only Consistency

Attributes-only consistency occurs when $O_S \neq O_R$ and $A_S = A_R$. A representative case is when contents are updated by content provider. Suppose the cache stores a content obtained at time 0. When the content provider updates the content at time *t*, it is likely that the new content is attribute-only consistent with the cached content. This occurs because content providers usually only change the object, but not the attributes.

3.10 Weak Consistency

Weak consistency is a condition where both object and attributes are different. This is a common phenomenon in pervasive content delivery context. Usually, when users request for a content, the pervasive system will detect the capabilities and preferences of users in order to *transcode* the original content into a best-fit representation for users. Therefore, the transcoded content becomes weakly consistent to the original.

In pervasive environment, finding a cached content with the exact required representation is very unlikely. Nevertheless, we can achieve *partial reuse* by transforming a suitable representation into a representation we need. Generally, there are 3 steps involved. Firstly, given a the requested representation R, find all transformed representations S from the cache that share the same URL as R. Next, among the candidates for S, select those that are of higher quality than R (eg. higher dimension, more color bit depth, higher quality metrics etc). Third, select the best S from the candidates according to heuristics and predefined policies [43, 44].

Heuristics are something like "choose the S with the highest quality as it is more likely to generate a good quality R". On the other hand, an example of policy is "S must be resized by more than 50% in order to generate high quality representations."

The pre-requisite for a transcoding system is that clients must indicate their capabilities and preferences, so that the system can figure out the best representations for each client. In general, any content of higher quality can be transcoded to lower quality ones, but the reverse is difficult to achieve. To efficiently rebuild higher quality content from lower quality ones, we need specific data format support such as scalable data model [37], JPEG2000 [38] and MPEG4 [39]. Furthermore, to support arbitrary content transformation in systems such as OPES, additional language support is needed to annotate changes made to content, and to assist systems in efficiently reuse transformed content.

3.11 Challenges

Challenges in web content consistency can be summarized with Figure 5.

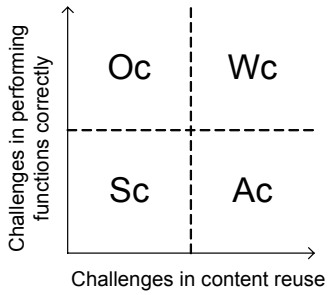

Figure 5: Challenges in Content Consistency

In strong consistency, where the entire content is consistency, there is not much problem. However, when attributes become inconsistency, as in object-only consistency, we face the challenge to ensure content delivery functions are performed correctly. On the other hand, when object becomes inconsistent, there are significant difficulties in reusing content. In particular, we need to achieve efficient partial content reuse which requires data format support, language support, client capabilities & preferences, and server directions and policies.

3.12 Scope of Study

Pervasive content delivery itself is a very big area for research, and many things are not well defined yet. For instance, given 2 representations of a content, which one is more "appropriate" to the user? These issues are still on-going research. This is the reason we point out that content consistency is important for pervasive environment too and it is our future work.

Even though our content consistency model is applicable to both non-pervasive and pervasive content delivery, the remaining of the thesis only focuses on the non-pervasive aspect. Specifically, we only consider the case where each content has 1 representation.

3.13 Case Studies: Motivations and Significance

Our content consistency model defines 4 content consistency classes and we will illustrate the problems of each of the classes with some real world case studies.

Strong consistency is not that applicable to the web environment we are interested in; it is mainly for the traditional systems in which data must be bit-by-bit equivalent. It also presents a very high requirement for the web environment, and not everyone can meet this requirement. Even if we meet this requirement, it just presents a perfect case and there are no implications to study about except the fact that we can reuse the content. In this sense, there is nothing interesting to study in strong consistency. On the other hand, weak consistency is meant for pervasive environment. This area is open for debate and still needs a lot of research to clearly define many concepts (eg. appropriateness of content).

Due to these reasons, we will only focus on object-only and attribute-only consistency. The 4 case studies we performed are shown in Table 1.

Consistency Class	**Case Study**
Object-only consistency	Chapter 4: Replica/CDN
	Chapter 5: Web Mirrors
	Chapter 6: Web Proxies
Attribute-only consistency	Chapter 7: Content TTL/Lifetime

Table 1: Case Studies and Their Corresponding Consistency Class

We selected three case studies to illustrate the object-only consistency and one for the attribute-only consistency. The case studies are chosen because they are the typical representatives of the web infrastructure. They do not represent all the consistency problems in the web today, but are sufficient to illustrate the consistency classes in our model. We also use the case studies to highlight some new insights and surprising results that were not previously reported in the literature.

Chapter 4 to 6 study replica/CDN, web mirrors and web proxies. These are special purpose networks that attempt to replicate contents in order to offload the server. As we go from replica/CDN to web proxies, we go from an environment that is tightly coupled to content server to one that is less coupled. Specifically, in replica/CDN (chapter 4), content server takes the responsibility to push updates to replicas using some private protocols for synchronization. This is so called the server-driven approach. On the other hand, web proxies (chapter 6) are less coupled with server and only rely on the standard protocol (HTTP/1.1). Servers play minimum involvement in this situation; they only provide TTL to proxies and let proxies perform consistency maintenance according to HTTP/1.1. Replica/CDN and web proxies represent both ends of the spectrum. By comparison, web mirrors (chapter 5) are somewhere in the middle of the spectrum. They are in an ambiguous region as they employ neither server-driven nor client-driven approach. Lastly, chapter 7 tries to find out whether TTL defined by servers are accurate as inaccuracy can lead to performance loss or staleness.

Chapter 4

CASE STUDY 1: REPLICA / CDN

4.1 Objective

Large web sites usually replicate content to multiple servers (replicas) and dynamically distribute the load among them. Replicas/CDN are tightly coupled with the content server. There is usually some automatic replication mechanism to ensure updates are properly propagated to all replicas. Replicas are transparent to users; from users' point of view, they are seen as a single server and are expected to behave like one.

The purpose of replica/CDN is to deliver content on behalf of the origin server. The question is whether it achieves this purpose? There are 2 situations when this purpose cannot be fulfilled. Firstly, if a replicated object is not the same as the origin, it means the replica holds an outdated copy. Obviously, this is bad as users are only interested in the most current copy. This is the subject of replica placement and update propagation, which have been explored by many researchers. Secondly, if the replicated attributes are not the same as the origin, then it can cause some misfunctions outside of the replica/CDN network. For example, if caching headers are

not consistent, it can cause caching errors on proxy and browser caches. By contrast, this issue has not received much attention from the research community. We argue that this is an important topic that has been overlooked in the past; therefore it is our focus in this case study.

In the replica/CDN environment, the responsibility lies mainly on content server to actively push updates to replicas. Since server knows when contents are updated, it is the best party to notify the replicas. We do expect the least consistency problem for this environment. With this case study, we try to find out if this is the case. If there are any problems, what are they? We anticipate for some interesting results from this study as no one did a similar study in the past.

4.2 Methodology

4.2.1 Experiment Setup

The input of our experiment is the NLANR traces gathered on Sep 21, 2004. For each request, we extract the site (Fully Qualified Domain Name) from the URL and perform DNS queries using the Linux "dig" command to list the A records of the site. Usually, each site translates to only 1 IP address. However, if a site has more than 1 IP address, it indicates the use of DNS-based round-robin load-balancing. This usually means that the content of the website is replicated to multiple servers. An example is shown in Table 2.

Site	www.cnn.com
Replica	64.236.16.116, 64.236.16.20, 64.236.16.52, 64.236.16.84, 64.236.24.12, 64.236.24.20, 64.236.24.28

Table 2: An Example of Site with Replicas

In reality, there maybe more than 1 server behind each IP address. However, it is technically infeasible to find out how many servers are there behind each IP address and there is no way to access them directly. Therefore, in our study, we only consider each IP address as a replica.

The traces originally contain 5,136,325 requests, but not all of them are used in our study. We perform 2 levels of pre-processing. Firstly, we filter out URLs not using replica (3,617,571 of 5,136,325). Secondly, we filter out URLs with query string (those containing the ? character) because NLANR traces are sanitized by replacing each query string with a MD5 hash. URLs with query strings become invalid and we can longer fetch them for study (227,604 of 1,518,754). After pre-processing, we have 1,291,150 requests to be studied, as shown in Table 3. Fully analysis of extent of replication is shown in Appendix A.

Input traces	NLANR Sep 21, 2004
Request studied	1,291,150
URL studied	255,831
Site studied	5,175

Table 3: Statistics of Input Traces

4.2.2 Evaluating Consistency of Headers

For each URL, we request the content from each of the replicas using HTTP GET method. The HTTP response headers are stored and compared for inconsistency.

For each header H, we search for replicas with these types of inconsistency:

- Missing – the header appears in some but not all of the replicas

- Multiple – at least one the replica have multiple header H. This category is only applicable when the header being studied cannot be combined into a single, meaningful comma-separated list as defined in HTTP/1.1. Examples are the Expires and Last-Modified header.
- ConflictOK – at least 2 of the replica have conflicting, but acceptable values. For example, if the Expires value is relative to the Date value, then 2 replica may show different Expires values if accessed at different time.
- ConflictKO – at least 2 of the replica have conflicting and unacceptable values.

When comparing header values, we allow +/-5 minutes tolerance for date types headers (Last-Modified & Expires) and numeric types headers (max-age & smax-age Cache-Control directive). All other headers must be bitwise equivalent to be considered consistent.

For Cache-Control header, multiple header values (if exist) are combined into one. The order of directives is not important as we check whether Cache-Control headers are "semantically consistent" and not bitwise consistent.

4.3 Caching Headers

4.3.1 Overall Statistics

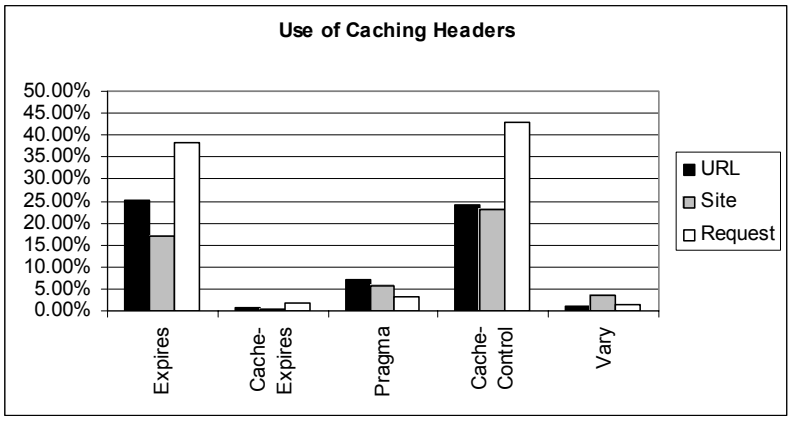

Figure 6: Use of Caching Headers

As shown in Figure 6, Expires and Cache-Control are the most widely used caching headers. If clients are HTTP/1.1 complaint, these 2 headers are sufficient. Pragma header is used for backward compatibility with HTTP/1.0 and is not so widely used. Vary header is only used under certain situations such as for content negotiation or compressed content. Interestingly, we found Cache-Expires in used by some sites even though it is not a standard header.

4.3.2 Expires

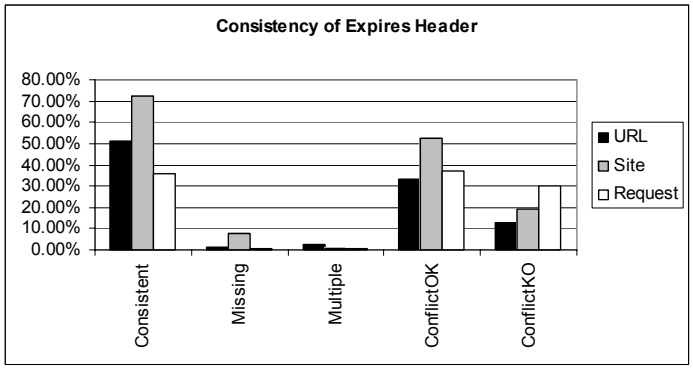

Figure 7: Consistency of Expires Header

Figure 7 shows that while 50.91% of URLs with Expires header are consistent, 1.54% are missing, 2.64% have multiple headers, 33.36% are conflictOK and 12.75% are conflictKO.

4.3.2.1 Missing Expires Value

Site	Number of Replica without Expires	
1. a1055.g.akamai.net (150 URL)	1 of 2	(50%)
2. pictures.mls.ca (89 URL)	4 of 8	(50%)
3. badsol.bianas.com (82 URL)	1 of 2	(50%)
4. sc.msn.com (60 URL)	1 of 2	(50%)
5. images.sohu.com (51 URL)	1 of 4	(25%)
6. jcontent.bns1.net (51 URL)	1 of 2	(50%)
7. gallery.cbpays.com (51 URL)	1 of 2	(50%)
8. graphics.hotmail.com (38 URL)	1 of 2	(50%)
9. photo.sohu.com (38 URL)	1-2 of 3	(33.3-66.7%)
10. static.itrack.it (35 URL)	1 of 2	(50%)

Table 4: Top 10 Sites with Missing Expires Header

The top 10 sites with missing Expires header are shown in Table 4. Expires time of these web contents are not precisely defined because only some of the replicas return the Expires header

while some do not. Clients who do not receive the Expires header do not know the exact expiry time of content and may cache the content longer than intended. This affects freshness of content and may also affect performance as revalidation maybe performed unnecessarily.

4.3.2.2 Multiple Expires Values

Site	Occurrence of Expires headers
1. ak.imgfarm.com (1069 URL, 2 replicas)	2-4
2. smileys.smileycentral.com (610 URL, 2 replicas)	2-16
3. promos.smileycentral.com (13 URL, 2 replicas)	2-8
4. ak.imgserving.com (1 URL, 2 replicas)	2

Table 5: Sites with Multiple Expires Headers

Table 5 shows all the sites with multiple Expires headers, all of them are operated by Akamai. The Expires header of each response is repeated between 2 to 16 times, but all the repeated headers carry the same expiry value. Thus, it doesn't cause much problem.

It is unnecessary to send multiple headers of the same value (a waste of bandwidth). Strictly speaking, multiple Expires headers is not allowed by HTTP1/1.1 since the multiple values cannot be concatenated into a meaningful comma-separated list as required by HTTP/1.1. We know that Expires value is supposed to be singular.

4.3.2.3 Conflicting but Acceptable Expires Values

Site	Expires Value
1. thumbs.ebaystatic.com (8953 URL, 4 replicas)	Expires = Date + 1 week
2. pics.ebaystatic.com (1865 URL, 2 replicas)	Expires = Date + 2 week
3. a.as-us.falkag.net (806 URL, 2 replicas)	Expires = Date + *x* min
4. a1040.g.akamai.net (410 URL, 2 replicas)	Expires = Date + *x* min
5. ar.atwola.com (406 URL, 12 replicas)	Expires = Date + *x* days
6. pics.ebay.com (383 URL, 2 replicas)	Expires = Date + 2 week
7. sc.groups.msn.com (348 URL, 2 replicas)	Expires = Date + 2 week
8. a1216.g.akamai.net (342 URL, 2 replicas)	Expires = Date + 6 hour
9. spe.atdmt.com (301 URL, 2 replicas)	Expires = Date + *x* hour
10. images.bestbuy.com (286 URL , 2 replicas)	Expires = Date + *x* min

Table 6: Top 10 Sites with Conflicting but Acceptable Expires Header

The top 10 sites with conflicting but acceptable Expires header are shown in Table 6. It is a common practice to set Expires time relative to Date. In Apache, this is done with the mod_expires module. Since our requests may reach replicas at different time, the Expires values returned could be different. In this case, the inconsistency of the Expires value is acceptable.

4.3.2.4 Conflicting and Unacceptable Expires Values

Site
1. thumbs.ebaystatic.com (1865 URL, 4 replicas)
2. spe.atdmt.com (1352 URL, 2 replicas)
3. i.walmart.com (812 URL, 2 replicas)
4. pics.ebaystatic.com (793 URL, 2 replicas)
5. a.as-us.falkag.net (580 URL, 2 replicas)
6. sc.groups.msn.com (194 URL, 2 replicas)
7. www.manutd.com (179 URL, 2 replicas)
8. cdn-channels.aimtoday.com (172 URL, 6 replicas)
9. a.as-eu.falkag.net (158 URL, 2 replicas)
10. include.ebaystatic.com (120 URL, 2 replicas)

Table 7: Top 10 Sites with Conflicting and Unacceptable Expires Header

Table 7 shows the top 10 sites with conflicting and unacceptable Expires header. It is intuitively wrong for Expires values to be inconsistent across all replicas. Replication should not cause different Expires values to be returned for the same content. This is bad for content provider as the exact expiry time of content cannot be precisely estimated.

4.3.2.5 Cache-Expires

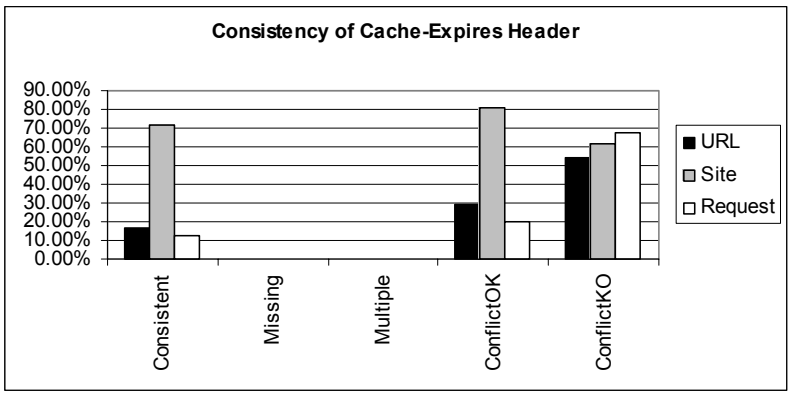

Figure 8: Consistency of Cache-Expires Header

In about 0.6% of URL, we saw a rather strange header – Content-Expires. It is not defined in HTTP/1.1, yet some sites use them. Most clients and caches would not recognize them and we are unsure why it is used.

As shown in Figure 8, there is no missing or multiple Content-Expires header, similar to that of Expires header in Figure 7. There is also similar percentage of URL that fall in the ConflictOK group, but there are significantly more URL in the ConflictKO group (54.27%) than that for the standard Expires header (12.75%).

4.3.3 Pragma

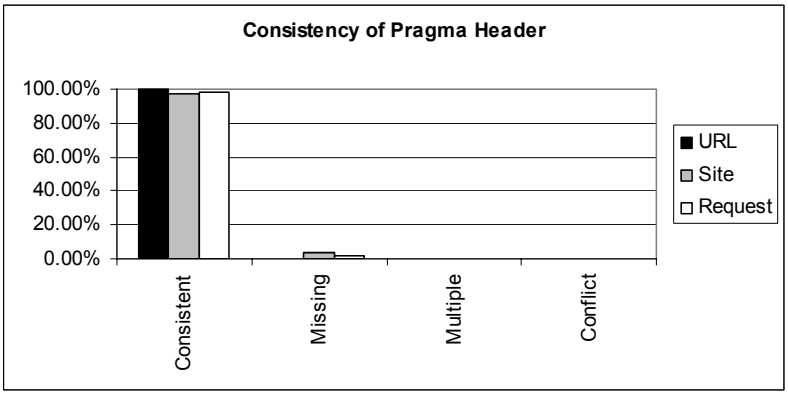

Figure 9: Consistency of Pragma Header

As shown in Figure 9, most of the sites with Pragma header are consistent while only some has missing header.

Site	Number of Replica with Missing Pragma	Scenario
1. ads5.canoe.ca (57 URL)	1 of 8 (12.5%)	1
2. www.bravenet.com (5 URL)	1-2 of 7 (14.2-28.5%)	1
3. www.nytimes.com (4 URL)	2 of 4 (50%)	1
4. msn.foxsports.com (3 URL)	1 of 2 (50%)	2
5. www.peugeot.com (3 URL)	1 of 2 (50%)	1
6. www.mtv.com (3 URL)	1 of 2 (50%)	2.
7. rad.msn.com (1 URL)	2 of 3 (66.7%)	1
8. rs.homestore.com (1 URL)	1 of 4 (25%)	1
9. www.trinpikes.com (1 URL)	4 of 5 (80%)	1
10. ibelgique.ifrance.com (1 URL)	8 of 12 (66.7%)	1

Table 8: Top 10 Sites with Missing Pragma Header

There are 80 URL (0.43%) with missing Pragma header. The top 10 sites are shown in Table 8.

Among these sites we found 2 common scenarios.

In the first scenario, the response with missing Pragma header has no other equivalent cache directives (eg. "Cache-Control: no-cache"). This causes serious inconsistency in caching behaviour. These contents are supposed to be uncacheable, but some of the replicas allow them to be cached due to the missing header.

In the second scenario, the response with missing Pragma header has a compensating "Cache-Control: max-age=x" header. However, for some replicas the max-age value is not 0 and differs from replica to replica. At first glance, this seems conflicting as uncacheable responses should have a max-age=0. We deduce that these responses could in fact be cached for a short duration. However, as Cache-Control is only defined in HTTP/1.1, content providers use the more restrictive Pragma: no-cache header for HTTP/1.0 caches. Nevertheless, due to the inconsistent max-age values, each content of these sites is cached for different durations at different caches, which is unfavourable to content providers.

4.3.4 Cache-Control

Multiple Cache-Control headers can exist in a HTTP response. In our experiment, we combine multiple Cache-Control headers into a single header according to HTTP/1.1 specification.

Table 9 shows the statistics of URL containing Cache-Control header. 94.36% of the URLs have consistent Cache-Control, while 5.64% do not. Furthermore, among all the URLs, there are 1.54% with missing Cache-Control header.

	URL		Site		Request	
Total with Cache-Control	61,618	100.00%	1,196	100.00%	554,191	100.00%
Missing	951	1.54%	70	5.85%	3,732	0.67%
Semantically con.	58,140	94.36%	1,185	99.08%	429,799	77.55%
Semantically incon.	3,478	5.64%	108	9.03%	124,392	22.45%
- public missing	7	0.01%	3	0.25%	30	0.01%
- private missing	4	0.01%	1	0.08%	22	0.00%
- no-cache missing	6	0.01%	2	0.17%	56	0.01%
- no-store missing	6	0.01%	2	0.17%	56	0.01%
- must-revalidate missing	2	0.00%	1	0.08%	6	0.00%
- max-age missing	2	0.00%	1	0.08%	4	0.00%
- max-age inconsistent	3,468	5.63%	107	8.95%	124,310	22.43%
- no-transform missing	0	0.00%	0	0.00%	0	0.00%
- proxy-revalidate missing	0	0.00%	0	0.00%	0	0.00%
- smax-age missing	0	0.00%	0	0.00%	0	0.00%
- smax-age inconsistent	0	0.00%	0	0.00%	0	0.00%

Table 9: Statistics of URL Containing Cache-Control Header

4.3.4.1 Missing Cache-Control Header

Site	No. of Replica without Cache-Control	Replica with Cache-Control
1. a1055.g.akamai.net (150 URL)	1 of 2 (50%)	Cache-Control: max-age=3xxx
2. pictures.mls.ca (89 URL)	4 of 8 (50%)	Cache-Control: max-age=432000
3. sc.msn.com (60 URL)	1 of 2 (50%)	Cache-Control: max-age=1209600
4. ads5.canoe.ca (57 URL)	1 of 8 (12.5%)	Cache-Control: private, max-age=0, no-cache
5. images.sohu.com (51 URL)	1 of 4 (25%)	Cache-Control: max-age=5184000
6. gallery.cbpays.com (51 URL)	1 of 2 (50%)	Cache-Control: max-age=86400
7. badsol.bianas.com (39 URL)	1 of 2 (50%)	Cache-Control: max-age=1728000
8. graphics.hotmail.com (38 URL)	1 of 2 (50%)	Cache-Control: max-age=2592000
9. photo.sohu.com (38 URL)	1-2 of 3 (33.3-66.7%)	Cache-Control: max-age=5184000
10. static.itrack.it (35 URL)	1 of 2 (50%)	Cache-Control: post-check=900,pre-check=3600, public

Table 10: Top 10 Sites with Missing Cache-Control Header

Table 10 shows the top 10 sites with missing Cache-Control header. The effect of missing Cache-Control header depends on the header value given by other replicas.

If other replicas have a Cache-Control: private, max-age=0, no-cache, it indicates the response is uncacheable. Thus, the replica with missing Cache-Control will make the response cacheable which seriously affects the freshness of content.

On the other hand, if other replicas have a max-age=x, then the replicas with missing Cache-Control simply do not provide implicit instruction to caches as to how long the cached response can be used without revalidation. Most caches will cache and use the content based on heuristics, different from the intention content provider. This means the same content is cached for different duration at different caches.

4.3.4.2 Semantically Inconsistent Cache-Control Header

11% of URL has Cache-Control headers that are not bitwise consistent. However, this does not necessarily mean that they are inconsistent because Cache-Control can appear in multiple headers and the order of directives is not important. It is possible for 2 Cache-Control headers to be bitwise inconsistent but having the same semantics (in our traces, this accounts for 0.49% of URLs with Cache-Control header). Thus, it is more meaningful to examine the semantics of Cache-Control headers to determine their true consistency.

The implications of inconsistent cache directives can be explained according to the purpose of those directives:

- When the "public" directive is missing from some replicas, it is not a big problem because responses are public (cacheable) by default.
- When the "private" directive is missing, the contents will be erroneously cached by public caches.
- Missing the "no-cache" or "no-store" also results in the contents being erroneously cached by both public and private caches.
- Missing the "must-revalidate" or "proxy-revalidate" directives can cause contents to be served even if stale which would not happen if these directives were present.
- Most of the semantically inconsistent Cache-Control headers have conflicting max-age values. The top 10 sites are shown in Table 11.

Site	**Countdown for Expires**	**Remarks**
1. pics.ebaystatic.com (793 URL, 2 replicas)	Yes	
2. i.a.cnn.net (369 URL, 8 replicas)	No	
3. www.oup.com (254 URL, 4 replicas)	No	
4. sc.groups.msn.com (194 URL, 2 replicas)	Yes	
5. mlb.mlb.com (166 URL, 2 replicas)	No	
6. include.ebaystatic.com (120 URL, 2 replicas)	Yes	
7. img.kelkoo.com (109 URL, 2 replicas)	No	max-age are negative values but Expires values are in future (conflict). No other cache directives present.
8. smileys.smileycentral.com (90 URL, 2 replicas)	No	
9. www.los40.com (85 URL, 2 replicas)	No	
10. a.sc.msn.com (84 URL, 2 replicas)	Yes	

Table 11: Top 10 sites with Inconsistent max-age Values

HTTP/1.1 section 14.9.3 states that both Expires header and max-age directive can be used to indicate the time when the content becomes stale. Intuitively, Expires header and max-age

should match, that is max-age should be the number of seconds towards the Expires time. However, HTTP/1.1 states that if both Expires and max-age exist, max-age takes precedence.

Some sites' max-age values are dependent on Expires value; it is a countdown towards the Expires time. Theoretically, this is supposed to be acceptable. However, in our input traces, these replicas' Expires values are inconsistent, which cause the max-age values to become inconsistent too. It is still puzzling why the same content has different Expires values when accessed through different replicas.

There are some other sites which simply provide random max-age values and we are unable to deduce any reasoning for this observation.

Inconsistent max-age values can cause problems, especially as content providers cannot precisely estimate when cached content will expire or revalidated. This may affect their ability to carefully implement content updates.

4.3.5 Vary

The consistency of Vary header is shown in Figure 10. While 70.75% of URL have consistent Vary header, 29.21% and 0.04% have missing and conflicting Vary header respectively.

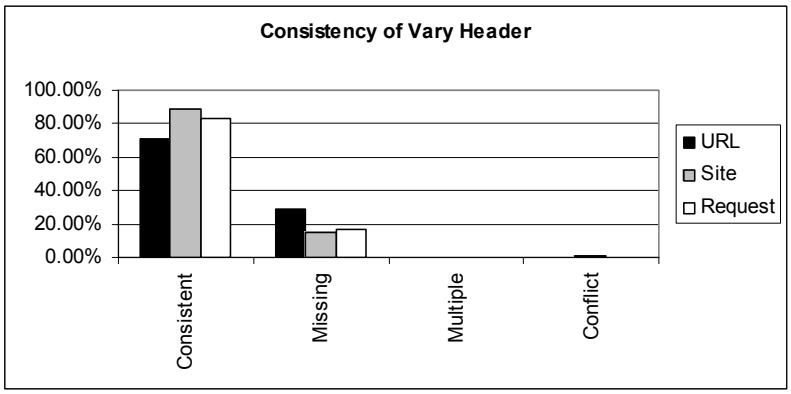

Figure 10: Consistency of Vary Header

Site	Replicas with Vary Header	Replicas without Vary Header (Compensating)	Remarks
1. img.123greetings.com	Vary: Accept-Encoding		OK
2. www.jawapos.com	Vary: Accept-Encoding,User-Agent		BAD
3. www.jawapos.co.id	Vary: Accept-Encoding,User-Agent		BAD
4. jawapos.com	Vary: Accept-Encoding,User-Agent		BAD
5. www.harris.com	Vary: *	Cache-Control: no-cache	OK
6. jawapos.co.id	Vary: Accept-Encoding,User-Agent		BAD
7. capojogja.board.dk3.com	Vary: Accept-Encoding		OK
8. sthumbnails.match.com	Vary: * (with no-cache directive)	No Cache-Control header	BAD
9. sports.espn.go.com	Vary: Accept-Encoding, User-Agent		BAD
10. mymail01.mail.lycos.com	Vary: *	No Cache-Control header	BAD

Table 12: Top 10 sites with Missing Vary Header

The top 10 sites with missing Vary header is shown in Table 12. Vary is a header to tell caches which request-headers are used to determine how the content is represented. It also instructs caches to store different content versions for each URL according to headers specified in Vary.

"Vary: Accept-Encoding" is commonly used to separate the cache for plaintext versus compressed content. It avoids caches from accidentally sending compressed content to clients who do no understand them. However, it is only a precaution measure because if a caching proxy is "intelligent" enough to differentiate plaintext and compressed content, it would not make this mistake. Therefore, if some replicas do not supply this Vary header, it may not be disastrous.

"Vary: Accept-Encoding,User-Agent" means that the content represented could be customized to the User-Agent. Responses without this header will not be cached correctly. To illustrate the seriousness of this issue, consider a site that customizes its contents to 3 browser types: Internet Explorer, Netscape, and others. If the Vary: User-Agent header is missing from some replicas, then proxy caches may erroneously respond to Netscape browsers with contents designated for Internet Explorer. This can seriously affect the accessibility to those sites as the customized content may not be rendered properly on other browsers.

"Vary: *" implicitly means the response is uncacheable. If the vary header is not present in other replicas' response, an equivalent Cache-Control: no-cache or Pragma: no-cache header should be present. Otherwise, responses from these replicas would be erroneously cached.

The top 10 sites suffer a mixture of problems mentioned above.

Our results show that there is only 1 site with conflicting vary headers - www.netfilia.com (1 URL, 6 replicas). This site returns 2 conflicting Vary values:

- Vary: Accept-Encoding, User
- Vary: Accept-Encoding

In practice, clients seldom send User header as it is not a recognized HTTP/1.1 header. Thus, even though the 2 Vary values are different, they may achieve similar results in most cases (unless the site aspects some users to use certain browsers that send the User header).

4.4 Revalidation Headers

4.4.1 Overall Statistics

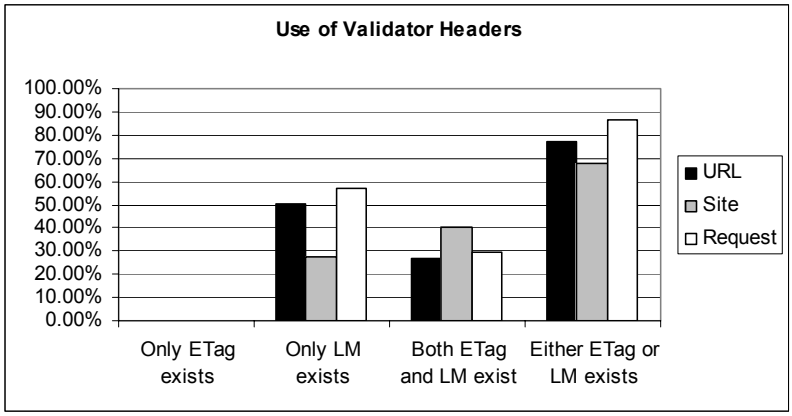

Figure 11: Use of Validator Headers

From Figure 11, we can see that majority of URL (77.50%) in our study has a cache validator (either ETag or Last-Modified header). The existence of a cache validator allows these URL to

be revalidated using conditional request when the content TTL expired. Clients revalidate by sending conditional If-Modified-Since (for Last-Modified) or If-None-Match (for ETag) requests to the server.

4.4.2 URLs with only ETag available

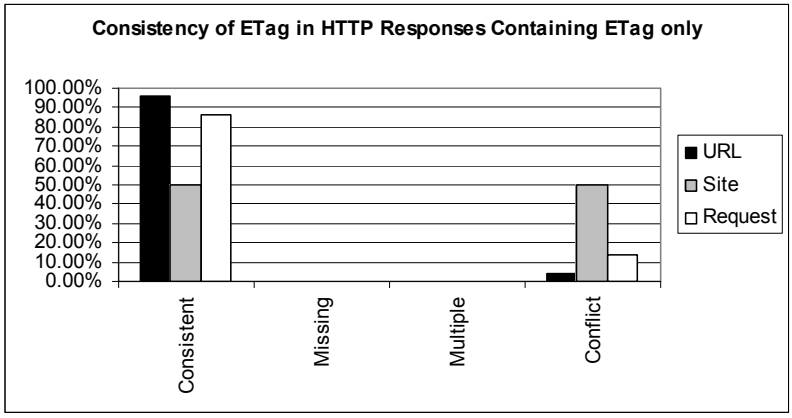

Figure 12: Consistency of ETag in HTTP Responses Containing ETag only

Figure 12 shows the consistency of ETag in HTTP responses containing ETag only. Majority of URL (96.17%) are consistent, but there are 3.83% with conflicting ETag.

Site
1. css.usatoday.com (6 URL, 2 speedera.net CDN replica)
2. i.walmart.com (1 URL, 2 replica)
3. www.wieonline.nl (1 URL, 3 replica)

Table 13: Sites with Conflicting ETag Header

All the sites with conflicting ETag header are shown in Table 13. Replicas in this class do not provide Last-Modified header, so using ETag is the only way to revalidate. However, revalidation may fail if users revalidate at a later time with a different replica than the one

initially contacted. This results in unnecessary full body retrieval even though the content might not have changed. If there are n replicas and each of them gives a different ETag, the probability of revalidation failure is $\frac{n-1}{n}$. This means revalidation failure rate increases with the number of replica.

4.4.3 URLs with only Last-Modified available

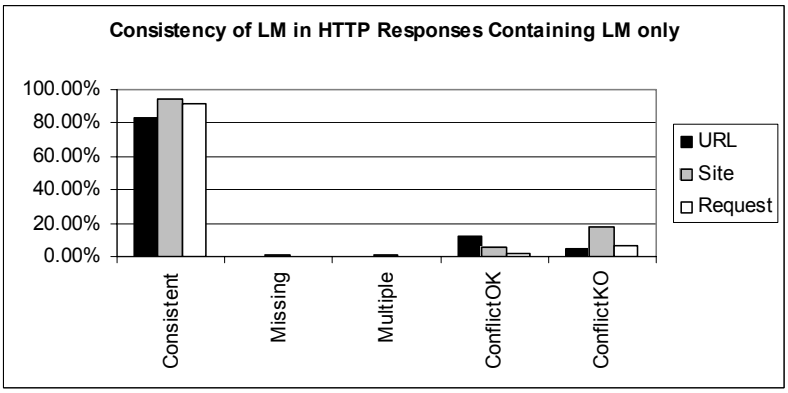

Figure 13: Consistency of Last-Modified in HTTP Responses Containing Last-Modified only

Figure 13 shows the consistency of Last-Modified in HTTP responses containing Last-Modified only. Most URLs (83.35%) provide consistent Last-Modified, except that there are 0.04% missing, 0.05% having multiple headers, 11.94% in ConflictOK and 5.11% in ConflictKO.

4.4.3.1 Missing Last-Modified Value

Site	Number of Replica without Last-Modified
1. bilder.bild.t-online.de (12 URL)	1 of 2 (50%)
2. sthumbnails.match.com (9 URL)	1 of 2 (50%)
3. www.sport1.de (7 URL)	1 of 3 (33.3%)
4. a.abclocal.go.com (4 URL)	1 of 2 (50%)
5. a1568.g.akamai.net (3 URL)	1 of 2 (50%)
6. En.wikipedia.org (2 URL)	2 of 10 (20%)
7. www.cnn.com (2 URL)	1-2 of 8 (12.5-25%)
8. fr.wikipedia.org (2 URL)	2 of 10 (20%)
9. id.wikipedia.org (1 URL)	2 of 10 (20%)
10. thanks4today.blogdrive.com (1 URL)	1 of 2 (50%)

Table 14: Top 10 Sites with Missing Last-Modified Header

The top 10 sites with missing Last-Modified header are shown in Table 14. Last-Modified is provided to clients so that they can revalidate using If-Modified-Since requests and retrieve the full body only when content changes. However, if not all replica of a site give Last-Modified, then some users lose the opportunity to revalidate. Inability to perform revalidation means that users need to download the full body even when they not changed. This is a waste of bandwidth.

4.4.3.2 Multiple Last-Modified Values

The top 10 sites with multiple Last-Modified headers are shown in Table 15. All the sites in this category provide 2 Last-Modified headers in the response. For most sites (except for www.timeforaol.com), the first Last-Modified value is the same as Date while the second Last-

Modified value is consistent across all the replicas. The second Last-Modified seems to be the real value as it is some time in the past. A sample response is shown in Table 16.

Site
1. www.timeinc.net (26 URL, 2 replica)
2. www.time.com (24 URL, 2 replica)
3. people.aol.com (4 URL, 2 replica)
4. www.health.com (3 URL, 2 replica)
5. www.business2.com (3 URL, 2 replica)
6. subs.timeinc.net (2 URL, 2 replica)
7. www.golfonline.com (2 URL, 2 replica)
8. www.cookinglight.com (1 URL, 2 replica)
9. www.timeforaol.com (1 URL, 2 replica)
10. www.life.com (1 URL, 2 replica)

Table 15: Top 10 Sites with Multiple Last-Modified Headers

Response headers from replica 205.188.238.110	**Response headers from replica 205.188.238.179**
HTTP/1.1 200 OK Date: Sun, 26 Sep 2004 13:41:37 GMT Last-modified: Sun, 26 Sep 2004 13:41:37 GMT Last-modified: Thu, 01 Jul 2004 19:47:05 GMT …	HTTP/1.1 200 OK Date: Sun, 26 Sep 2004 13:41:37 GMT Last-modified: Sun, 26 Sep 2004 13:41:37 GMT Last-modified: Thu, 01 Jul 2004 19:47:05 GMT …

Table 16: A Sample Response with Multiple Last-Modified Headers

For both replicas, sending If-Modified-Since requests using the second (real) Last-Modified returns "304 Unmodified". Repeating the conditional request with the first Last-Modified (same as Date) also returns 304, because the first date is later than the real Last-Modified, therefore using it causes no problem.

Let us review how HTTP treats multiple headers. HTTP Section 4.2 states that multiple headers with same field name may exist if and only if the field-value can be combined into a comma-separated list. However, common sense tells us Last-Modified is supposed to consist of 1 value, so we can't combine them in a meaningful way. Thus, multiple Last-Modified is unacceptable. Even though multiple Last-Modified is semantically undefined, many browsers still handle them well as they only read the first value. If clients revalidate using multiple Last-Modified values (by combining multiple Last-Modified into a single value or by specifying multiple If-Modified-Since for each value), servers usually read the first value too.

4.4.3.3 Conflicting but Acceptable Last-Modified Values

Site	Last-Modified Value
1. thumbs.ebaystatic.com (14945 URL, 4 replicas)	LM = Date
2. www.newsru.com (99 URL, 3 replicas)	LM = Date
3. thumbs.ebay.com (74 URL, 2 replicas)	LM = Date
4. www.microsoft.com (58 URL, 8 replicas)	LM = Date
5. www.cnn.com (53 URL, 8 replicas)	LM = Date
6. money.cnn.com (14 URL, 4 replicas)	LM = Date
7. www.time.com (14 URL, 2 replicas)	LM = Date
8. udn.com (16 URL, 8 replicas)	LM = Date
9. www.face-pic.com (12 URL, 8 replicas)	LM = Date
10. newsru.com (9 URL, 3 replicas)	LM = Date

Table 17: Top 10 Sites with Conflicting but Acceptable Last-Modified Header

Table 17 shows the top 10 sites with conflicting but acceptable Last-Modified header. Many sites define the Last-Modified value relative to the time of response (usually Last-Modified = Date). Since our requests may reach replicas at different times, the Last-Modified values would

be different. This is intentional to control caching, usually to tell clients that the content is freshly generated. They also have explicit TTL defined.

However, by setting Last-Modified = Date, these sites implicitly disable validation as all If-Modified-Since requests will fail, leading to higher network usage. In fact, there is a better approach to achieve high consistency. We can explicitly set a low TTL (eg. 1 min) to contents. Now, instead of using Last-Modified, we use ETag and only change its value when the content body changes. This achieves strong consistency by increasing revalidation, but content providers do not send redundant content bodies.

4.4.3.4 Conflicting Last-Modified Values

Top 10 Sites	**Successful revalidation using**		
	The right LM at the right replica	**Any date >= the earliest LM**	**Any arbitrary date**
1. thumbs.ebaystatic.com	√		
2. i.cnn.net	√		
3. images.amazon.com	√	√	
4. ar.atwola.com	√	√	√ (up to a few weeks earlier than the earliest LM)
5. images.overstock.com	√		
6. www.topgear.com	√		
7. msnbcmedia.msn.com	√		
8. udn.com	√		
9. a1072.g.akamai.net	√		
10. rcm-images.amazon.com	√	√	

Table 18: Top 10 Sites with Conflicting Last-Modified Header

5.11% of URLs suffer from inconsistent Last-Modified header. The top 10 sites with conflicting Last-Modified header are shown in Table 18. Sites that fall into this category include

brand names like Amazon and sites using CDN. This may be caused by improper replication, that is, contents are replicated without the associated Last-Modified information. This can result in revalidation failure if clients revalidate at a replica different from the one initially contacted.

Note that some sites understand that inconsistent Last-Modified can fail revalidations. Instead of making their Last-Modified consistent, they employ special treatments for If-Modified-Since request. It is interesting to see that instead of spending the effort on making Last-Modified consistent, these sites spend efforts on counter-measures which are not 100% error-proof. We saw 2 types of special treatments for If-Modified-Since requests.

The first type works as follows. Suppose the Last-Modified values of a content are in the ordered set of $\{LM_1, LM_2, ..., LM_m\}$ where $LM_1 \le LM_2 \le ... \le LM_m$, the replicas are configured to respond with "304 Unmodified" if users revalidate with any date equal to or later than LM_1.

The second type of sites accepts If-Modified-Since requests with any arbitrary date. For example, ar.atwola.com accepts date earlier than LM_1 up to a few weeks earlier.

These special treatments help to reduce the problem caused by inconsistent LM. However, it only works for clients who directly retrieve from replicas. The inconsistency problem cannot be eliminated if clients use proxy servers to access the sites. For example (see Figure 14), client A fetches content through proxy and the response specifies Last-Modified value LM_1. Later, another client later forces the proxy to fetch the new version (using no-cache or max-age=0

directives), the proxy now get a different LM value LM_2. If $LM_2 \geq LM_1$ and client A now revalidates with the proxy, the proxy will tell the client content is modified where it fact it doesn't.

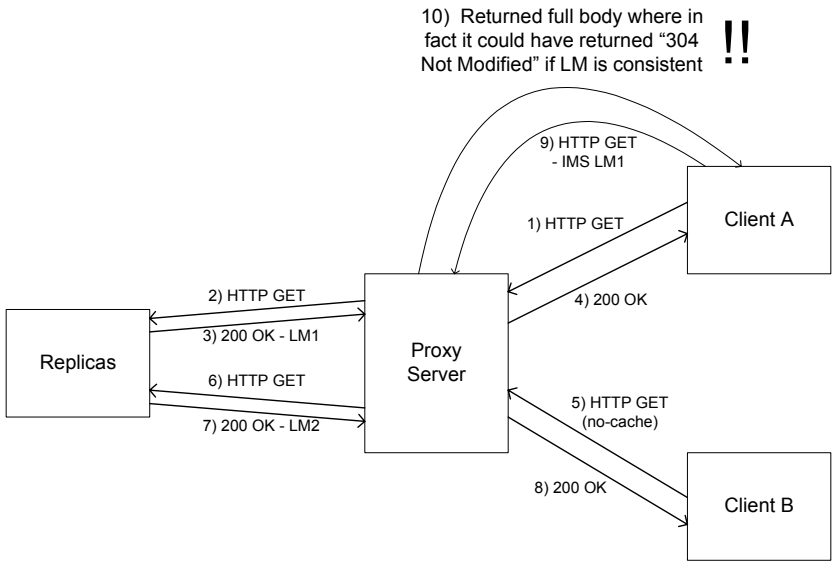

Figure 14: Revalidation Failure with Proxy Using Conflicting Last-Modified Values

4.4.4 URLs with both ETag & Last-Modified available

4.4.4.1 How Clients and Servers Use ETag and Last-Modified

Before we present the results, let us first study how clients and servers use ETag and Last-Modified, if both validator exist.

At client side, HTTP/1.1 section 13.3.4 states that clients must use If-None-Match for conditional requests if ETag is available. Furthermore, both If-Modified-Since and If-None-Match should be used for compatibility with HTTP/1.0 caches (ETag not supported in

HTTP/1.0). However, in reality, Microsoft Internet Explorer (90% browser share [40]) only uses If-Modified-Since even though ETag is available. Another browser, Mozilla/FireFox comply to HTTP/1.1 as it uses both If-Modified-Since and If-None-Match.

At server side, HTTP/1.1 section 13.3.4 states that servers should respond with "304 Not Modified" if and only if the object is consistent with ALL of the conditional header fields in the request. In reality, Apache (67.85% market share [41]) - both version 1.3 and 2.0 ignore If-Modified-Since and use If-None-Match only even though both conditional headers exist (non-compliant). The rational could be that ETag is a strong validator while LM is weak. If strong validator is already consistent, there is no need to check for weak validator. On the other hand, Microsoft IIS server (21.14% market share) version 5.0 returns 304 if any of the If-Modified-Since or If-None-Match conditions is satisfied (non-compliant) while version 6.0 returns 304 if and only if both If-Modified-Since and If-None-Match is satisfied (compliant).

At proxy side, NetCache/NetApp requires that both If-Modified-Since and If-None-Match must be satisfied. Another popular cache server - Squid 2.5 stable, does not support ETag; If-None-Match is ignored completely. Patches for ETag available but not widely used.

Even though not all current browsers and servers are fully HTTP/1.1 compliant with regards to If-Modified-Since and If-None-Match, it is still important that both ETag and Last-Modified be consistent when they are available. This is to avoid problems with browsers/servers that are compliant or when compliance improves over the time.

4.4.4.2 Inconsistency of Revalidation Headers

Type	URL		Site		Request	
1. Both ETag and LM exist	69,070	100.00%	2,071	100.00%	378,153	100.00%
2. ETagMissing	395	0.57%	47	2.27%	4,437	1.17%
3. ETagMultiple	36	0.05%	2	0.10%	697	0.18%
4. ETagConflict	43,518	63.01%	1,250	60.36%	234,647	62.05%
5. LMMissing	96	0.14%	22	1.06%	414	0.11%
6. LMMultiple	0	0.00%	0	0.00%	0	0.00%
7. LMConflictOK	380	0.55%	33	1.59%	6,446	1.70%
8. LMConflictKO	6,968	10.09%	419	20.23%	37,958	10.04%
9. LMConflictKO & ETagConflict	6,821	9.88%	400	19.31%	37,610	9.95%
10. Both ETag & LM incon. (ex LMConflictOK)	7,019	10.16%	419	20.23%	38,285	10.12%
11. Either ETag or LM incon. (ex LMConflictOK)	43,978	63.67%	1,286	62.10%	239,848	63.43%

Table 19: Types of Inconsistency of URL Containing Both ETag and Last-Modified Headers

Table 19 shows the types of Inconsistency of URL containing both ETag and Last-Modified headers. When a response contains both ETag and Last-Modified, inconsistency can occur to one of both of the headers. Whether or not the inconsistent header will cause validation problems depends on how clients revalidate and how servers respond to the request.

Comparing item 8 & 9 in Table 19, we can see that most of the response with conflicting Last-Modified also has conflicting ETag. We can deduce that ETag is likely to be derived from Last-Modified (such as in the default Apache configurations), thus inconsistent Last-Modified directly causes inconsistent ETag.

From item 11, we can see that if all clients and servers are HTTP/1.1 compliant, 63.67% of URLs will not be able to revalidate properly because one of the validators are inconsistent. This indicates a hidden performance loss that has been overlooked.

4.5 Miscellaneous Headers

We also check the consistency of other headers such as Content-Type, Server, Accept-Ranges, P3P, Warning, Set-Cookie, MIME-Version. Majority (>97%) of important headers such as Content-Type and P3P are consistent. Informational headers such as Server, Accept-Ranges, Warning and MIME-Version have varying degree of inconsistency, but these do not cause any significant problem. Lastly, the Set-Cookie header is very inconsistent due to the nature of their usage (which is normal). Intuitively, for each request without a Cookie header, the server should attempt to set a new Cookie using the Set-Cookie header.

4.6 Overall Statistics

We are interested to know the extent of inconsistency in general. We do this by searching for URL with "critical inconsistency" in caching and revalidation headers, according to the types stated in Table 20 (ConflictOK are excluded).

Caching	Expires: missing, ConflictKO
	Pragma: missing
	Cache-Control: missing, or the directives private, no-cache, no-store, must-revalidate, max-age missing, or max-age inconsistent
	Vary: missing or conflict
Revalidation	ETag: missing, multiple or conflict
	Last-Modified: missing, multiple or conflictKO

Table 20: Critical Inconsistency in Caching and Revalidation Headers

Results from Figure 15 show that 22.57% of URLs with replica suffer some form of critical inconsistency. This affects 32.44% of sites and 35.06% of requests. This confirms that replica and CDN suffer severe inconsistency problems that result in revalidation failure (performance loss), caching error, content staleness and presentation errors.

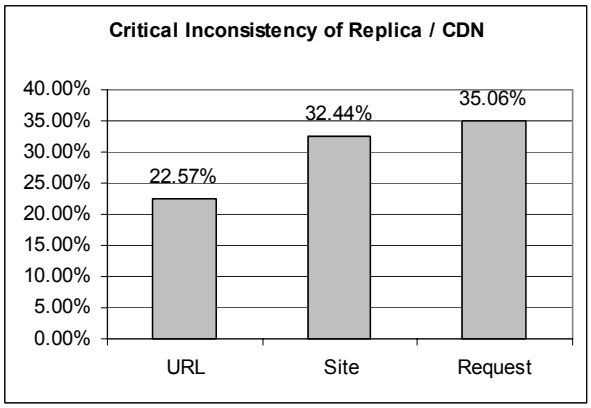

Figure 15: Critical Inconsistency of Replica / CDN

4.7 Discussion

This case study has revealed some interesting results. Even though replication and CDN are well studied areas, our measurement shows that current replica/CDN suffer varying degree of inconsistency in content attributes (headers). If server pushed content updates to replicas (as should be the case), it should not be difficult to push attributes as well. There is really no reason for inconsistency to occur. The reason inconsistency happens is because many replica/CDN are only concerned with properly replicating content body, and forget or do not pay much attention to content attributes.

Even though problems caused by inconsistent attributes may not be immediately "visible", depending on the attributes that are inconsistent, this can lead to various problems. For instance, when the Expires header is inconsistent, it can lead to either of 2 situations. If the Expires value is set higher than the real one, then bandwidth is wasted on revalidations (used inefficiently). Conversely, if the Expires value is set lower than the real one, though no necessarily the case, it is possible for contents to become stale (outdated).

Inconsistency of other headers can also lead to similar disastrous effects. For example, inconsistency in the Pragma, Vary, or Cache-Control header can cause serious caching problems such as caching of uncacheable contents. We also observed that many sites have inconsistent validator headers (ETag and Last-Modified). This causes unnecessary revalidation failures and thus decreasing performance (higher latency, bandwidth and server load).

The only related works we can find are replica placement strategies and CDN cache maintenance. Replica placement strategies [58, 59, 60, 61, 62] focus on achieving optimality under certain performance metrics (latency, minimum hop etc) by properly placing replica copies near users. On the other hand CDN cache maintenance [24, 30] extends traditional proxy cache maintenance approaches to CDN. To the best of our knowledge, our case study is the first to measure consistency of the real world replica/CDN, especially in terms of content attributes.

Chapter 5

CASE STUDY 2: WEB MIRRORS

5.1 Objective

In replica/CDN environment (chapter 4), server pushes updates to replicas while in web proxies (chapter 6) updates are pulled using HTTP/1.1. Web mirrors are in the ambiguous region, somewhere in the middle of the spectrum. It is infeasible for servers to push updates to mirrors and they do not usually have a partnership agreement. On the other hand, pulling regularly from servers taxes on bandwidth consumption and is not preferred by mirrors. As a result, there is no well-defined mechanism or requirements for mirrors to keep updated with server.

The purpose of mirrors is to help server in serving contents, thereby offloading the server. In terms of consistency, mirrors should be "similar" to the origin servers. However, since mirrors are not as tightly coupled with server as in the first case study, we expect the consistency situation to be worse than that of replica/CDN. With this case study, we investigate the inconsistency of mirrored web contents in the Internet today.

5.2 Experiment Setup

Subject of Study	Origin URL	Mirrors' URLs obtained from
A. Main page of Squid website	www.squid-cache.org/index.html	http://www.squid-cache.org/Mirrors/http-mirrors.html
B. Main page of Qmail website	www.qmail.org/top.html	www.qmail.org
C. Microsoft download file	download.microsoft.com/download/9/8/b/98bcfad8-afbc-458f-aaee-b7a52a983f01/WindowsXP-KB823980-x86-ENU.exe	www.filemirrors.com

Table 21: Selected Web Mirrors for Study

We selected 3 subjects for study, as shown in Table 21. For each subject, we identify the origin URL as well as the mirror URLs. Subject A (Squid) & B (Qmail) are relatively-dynamic contents, as they are updated every few days or weeks. In contrast, subject C (Microsoft) is a static object, which is not expected to be updated over the time.

The experiment was conducted on August 20, 2003 4:00PM. The content at each origin URL is retrieved and serves as reference. The same content is then retrieved from each mirror URL for comparison. We compare all HTTP headers and entity bodies. Some of the mirrors are unreachable, thus our discussion is based on working mirrors only.

For each subject of study, we performed 3 separate tests to see whether:

- the mirror sites are *up-to-date* (test #1)
- the mirror sites *modify* the content of the object (test #2)

- the mirror sites *preserve* HTTP headers of the object (test #3)

Operations performed in each test are:

- Test #1: Last-modified date and version number are indicated in the content of A (Squid) & B (Qmail). We use these metrics to identify whether their mirror sites are *up-to-date*. A mirror is *up-to-date* if its last-modified date and version number match the ones at the origin. On the other hand, we do not perform this test on subject C (Microsoft) as it is known to be static.
- Test #2: If a mirror is up-to-date, we perform bitwise comparison of its content with the origin's to see whether the mirror *modifies* the object's content.
- Test #3: For each mirror, we compare its HTTP headers with the origin's to see whether the mirrors preserve the headers.

Test #2 & #3 are performed only if the mirrored content is up-to-date, as we are unable obtain the same outdated versions from the origin server for comparison.

5.3 Results

Results for each subject are shown in Table 22, Table 23 and Table 24. In each table, we divide the mirrors into up-to-date and outdated. Then each up-to-date mirror is further checked to see if it preserved the headers from origin.

Squid Mirrors: 21							
Up-to-date: 17							Out-dated: 4
	Exact Replica: 16			Modified Replica: 1			1) 2 yr 4 mth, 2) 1 yr 8 mth, 3) 5 mth, and 4) 3 mth
Object headers	Preserv ed	Conflict	Missing	Preserv ed	Conflict	Missing	
Last-Modified	13	0	3	0	1	0	
ETag	0	13	3	0	1	0	
Expires	0	0	16	0	1	0	
Cache-Control	0	0	16	0	0	1	
Content-Type	11	5	0	0	0	1	

Table 22: Consistency of Squid Mirrors

Qmail Mirrors: 152							
Up-to-date: 149							Out-dated: 3
	Exact Replica: 146			Modified Replica: 3			1) 26 days, 2) 26 days, and 3) 1 day
Object header	Preserv ed	Conflict	Missing	Preserve d	Conflict	Missing	
Last-Modified	131	5	10	3	0	0	
ETag	2	128	16	0	3	0	
Content-Type	113	33	0	2	1	0	

Table 23: Consistency of Qmail Mirrors

Microsoft Mirrors: 29					
Up-to-date: 29					Out-dated: 0
	Exact Replica: 29			Modified Replica: 0	
Header	Preserv ed	Conflict	Missing		
Last-Modified	1	28	0		
ETag	0	29	0		
Content-Type	25	4	0		

Table 24: Consistency of (Unofficial) Microsoft Mirrors

From Table 22 and Table 23, we can see that 4 of 21 Squid mirrors and 3 of 152 Qmail mirrors are outdated. The duration of mirrors being outdated range from as long as 2 years and 4 months to as short as 1 day. This happens because mirrors are managed autonomously by 3rd parties on best-effort basis. Origin servers have no control over how frequent mirrors are updated.

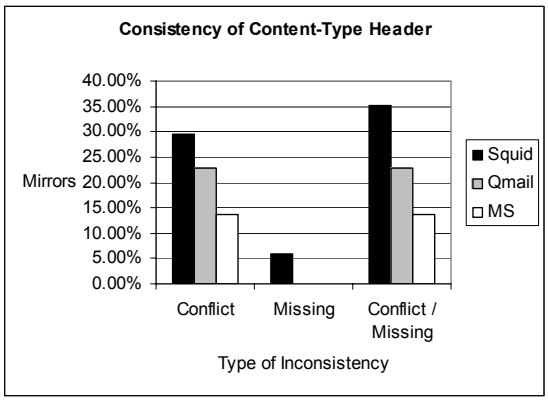

Figure 16: Consistency of Content-Type Header

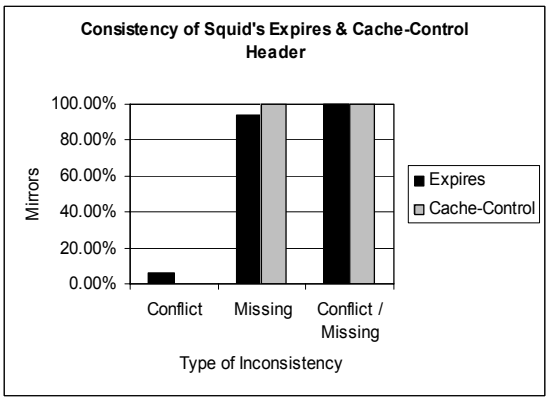

Figure 17: Consistency of Squid's Expires & Cache-Control Header

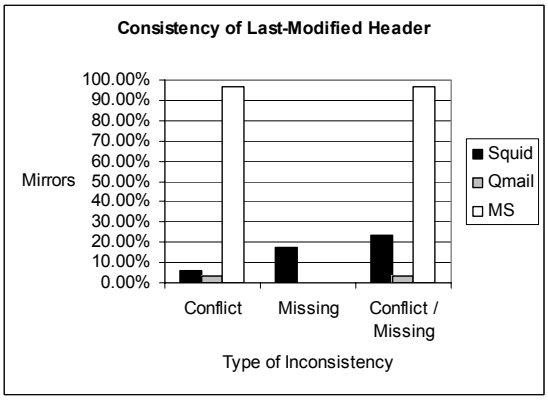

Figure 18: Consistency of Last-Modified Header

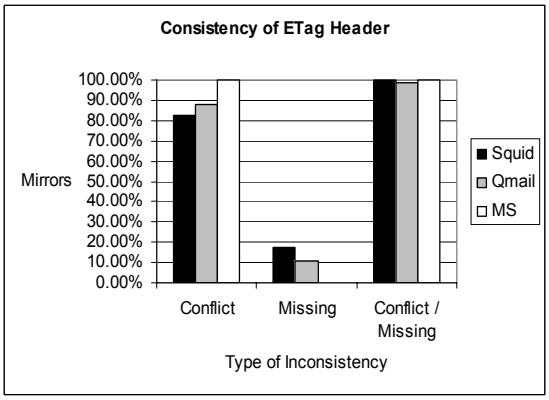

Figure 19: Consistency of ETag Header

We observed that many mirrors do not consistently replicate HTTP headers. Mirroring is usually done using specialized web crawling software such as *wget* and *rsync*. With proper settings, these software can replicate contents and retain their last modification time; however other HTTP headers are discarded. We note that for unofficial mirrors (subject C), contents are replicated in an ad-hoc manner, probably by downloading using browsers which do not preserve last modification time. This is clearly seen from Figure 18 that most of subject C's mirrors do not preserve the "Last-Modified" header

Some mirrors appear to preserve the headers of objects, such as the Content-Type header. However, we believe this is not due to proper header replication, but due to similar web server configurations. For instance, many web server software assign Content-Type: "text/html" to files with .html extension.

From Figure 16, we can see that some mirrors do not preserve Content-Type header. This can cause browsers to display the object differently than the intention of content provider. We noted that many mirrors change Content-Type into another compatible value, for example from "text/html" to "text/html; charset=UTF-8". However, we saw 1 of the Qmail's mirrors changes the original Content-Type from "text/html" to "text/html; charset=big5", which can cause browsers not supporting Big5 to unnecessarily download the Big 5 "language-pack".

From Figure 17, we see that all Squid mirrors do not preserve caching directives (Expires & Cache-Control). This can cause mirrored objects to be cached incorrectly with respect to the intention of content provider.

As we can see from Figure 18 and Figure 19, web mirrors suffer serious inconsistent in validator headers. If validators are missing, clients will not be able to revalidate using If-modified-since or If-none-match conditional requests. On the other hand, if validators are generated by mirrors themselves, then revalidation becomes ambiguous as the mirrors have no mechanism to determine the validity of objects. Figure 18 shows that 23.53% of Squid mirrors, 3.36% of Qmail mirrors and 96.55% of unofficial Microsoft mirrors do not preserve Last-Modified header; while Figure 19 shows that nearly all of Squid, Qmail and unofficial Microsoft mirrors do no preserve ETag header.

5.4 Discussion

Since there is no clear consistency mechanism for web mirrors, it is not surprising to see that the inconsistency of web mirrors is worse than that of replica/CDN.

The first problem is in accuracy or currency of content. Since neither server-driven (push) nor client-driven (pull) are appropriate for mirrors, many mirrors are outdated. From HTTP/1.1 perspective, there are no requirements for mirrors to do anything with regards to consistency.

Some mirrors abuse their "freedom" by changing the contents. Whether or not this is permitted by content provider is a subjective matter. For us, a bit changed is changed, we do not know if it is significant or not. Another significant problem is that ownership of content is not well-defined; there is no way to find out the true "owner" of contents. Most people would equate owner with the distribution site address, so mirrors are treated as the owners. Actually, we can

view web mirrors as temporary servers who try to offload content servers. Since mirrors are loosely coupled with server, suffer varying degree of inconsistency, and some even change contents, we ask whether mirrors should be responsible for revalidation. If not, who shall users revalidate with? The present HTTP/1.1 standard does not address this issue; therefore we will propose an ownership-based solution to address this problem in Chapter 8.

Though there are some works related to mirrors, none has focused on consistency issues. Makpangou et al. developed a system called Relais [26] which can reuse mirrors. However, they do not mention the consistency aspect of mirrors. Other work related on mirrors are [25] which examines the performance of mirror servers to aid the design of protocols for choosing among mirror servers, [33, 34] which propose algorithms to access mirror sites in parallel to increase download throughput, and [35, 36] which propose techniques for detecting mirrors to improve search engine results or to avoid crawling mirrored web contents.

Chapter 6

CASE STUDY 3: WEB PROXY

6.1 Objective

Web proxies pull content updates on demand according to users' requests, in hope for reducing bandwidth consumption. They are expected to be transparent (ie. do not change contents unnecessarily), and comply to HTTP/1.1 specifications discretely. There are rules to be followed if proxy wanted to add, change or drop any headers.

When users retrieve content via proxies, an important question to ask is whether they get the same content as the one on the server? This is what we try to find out with this case study.

Compared to the first 2 case studies in which inconsistencies are resulted from careless replication or mirrors that intentionally modify content; inconsistent web proxies are likely due to improper settings.

6.2 Methodology

We gather more than 1000 open proxies from well known sources shown in Table 25. All the proxies are verified by a script to ensure they work.

http://www.publicproxyservers.com/page1.html
http://www.aliveproxy.com/transparent-proxy-list/
http://www.stayinvisible.com/index.pl/proxy_list?order=&offset=0
http://www.proxy4free.com/page1.html
http://www.atomintersoft.com/products/alive-proxy/proxy-list/?p=100
http://www.proxywhois.com/transparent-proxy-list.htm

Table 25: Sources for Open Web Proxies

```
HTTP/1.1 200 OK
Date: Sun, 31 Oct 2004 14:25:27 GMT
Server: Unknown/Version
X-Powered-By: PHP/4.3.3
Expires: Sat, 01 Jan 2005 12:00:00 GMT
Cache-Expires: Sat, 01 Jan 2005 12:00:00 GMT
Pragma: no-cache
Cache-Control: must-revalidate, max-age=3600
Vary: User-Agent
ETag: "k98vlkn23kj8fkjh229dady"
Last-Modified: Thu, 01 Jan 2004 12:00:00 GMT
Accept-Ranges: bytes
MIME-Version: 1.0
P3P: policyref="http://somewhere/w3c/p3p.xml", CP="CAO DSP COR CUR ADM DEV
TAI PSA PSD IVAi IVDi CONi TELo OTPi OUR DELi SAMi OTRi UNRi PUBi IND
PHY ONL UNI PUR FIN COM NAV INT DEM CNT STA POL HEA PRE GOV"
Set-Cookie: PRODUCT=ABC
X-Extra: extra header
X-SPAM: spam header
Connection: close
Content-Type: text/html
```

Figure 20: Test Case 1 - Resource with Well-known Headers

We use 2 special cases to test each proxy. The first case is a resource that returns many common headers, as shown in Figure 20. The objective of using the first case is to check whether proxies remove or modify well known headers. The second test case is a resource that returns the bare minimum headers, as shown in Figure 21. We use the second test case to check whether proxies insert any default header values when they don't exist.

We request the 2 resources through each of the proxies and store all the responses. For both cases, we check whether proxies add, delete or modify headers.

```
HTTP/1.1 200 OK
Date: Sun, 31 Oct 2004 14:31:48 GMT
Server: Apache/2.0.47 (Unix) PHP/4.3.3
X-Powered-By: PHP/4.3.3
Connection: close
Content-Type: text/html
```

Figure 21: Test Case 2 - Resource with Bare Minimum Headers

6.3 Case 1: Testing with Well-Known Headers

We study the consistency of headers from 3 aspects: modification, addition and removal of headers.

A) Modification of Existing Headers

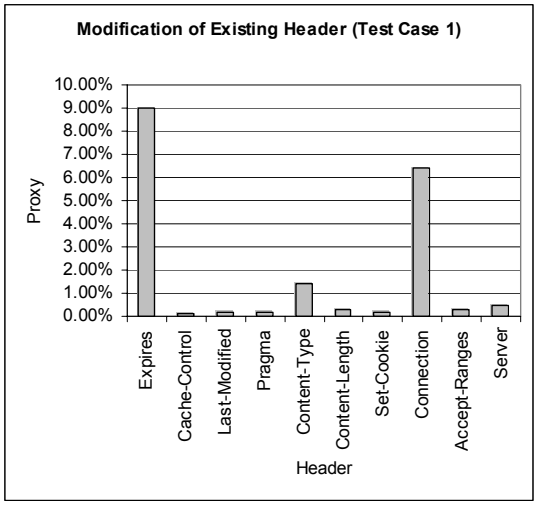

Figure 22: Modification of Existing Header (Test Case 1)

Figure 22 shows modification of existing header for test case 1. It is surprising that 9% of proxies modify the value of Expires header. Actually, these proxies insert new Expires header before the existing headers, so it takes precedence and is used by clients. Most of the proxies that modify Expires header also add/modify Cache-Control headers. This practice may be frowned upon by content providers as the contents' explicit caching headers are overwritten. At the same time, users using these proxies may also experience content staleness.

We found 1 of the proxies (0.1%) changes the original "Cache-Control: must-revalidate, max-age=3600" to Cache-Control: no-cache, must-revalidate, max-age=3600. Since we also specified "Pragma: no-cache", the change is acceptable. Another 2 proxies (0.2%) in our test set modify the Last-Modified header. Modifying the Last-Modified header can cause revalidate problems if users later revalidate with other proxies or the origin server.

1.38% of the proxies modify the value of Content-Type header. Our original Content-Type value is changed from "text/html" to "text/html; charset=iso-8859-1", "text/html; charset=UTF-8" or "text/html; charset=shift_jis". This means the proxies assign default character sets to those contents without explicit character set. This can cause serious problems when the default character set differs from the content's actual one. For instance, Korean web pages that do not provide explicit charset will render incorrectly if assigned default charset=shift_jis.

0.3% of proxies modify the Content-Length because the original content was transcoded or modified to include advertisements. The change of Content-Length value in this case is reasonable.

2 of the proxies (0.2%) append their own cookie to the original, possibly to track clients using the proxy. This does not pose a major issue as the original cookie is still intact.

Connection, Accept-Ranges and Pragma headers are also modified by some proxies. In fact, only the case of the values differs. For example, "Connection: close" is changed to "Connection: Close" and "Pragma: no-cache" to "Pragma: No-Cache". HTTP/1.1

specification defines header values as case-sensitive (only field names are case-insensitive), therefore some clients may not be able to interpret them correctly.

Server header is also modified by some proxies. Since the header is only for informational purposes, there is no major concern.

B) Addition of New Headers

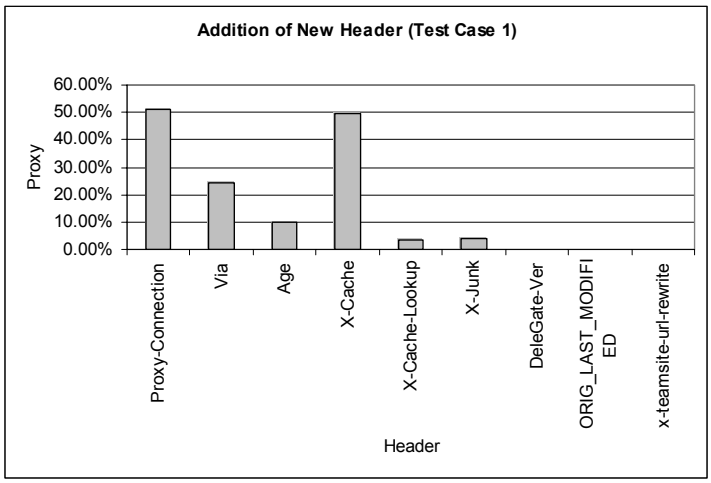

Figure 23: Addition of New Header (Test Case 1)

Addition of new header for test case 1 is shown in Figure 23. It is common for proxies to insert proxy-related headers such as Proxy-Connection, Via and Age. Some other vendor-specific caching proxy headers are also added: X-Cache and X-Cache-Lookup. All those headers are informational or for specific purposes that do not interfere with the normal operation of content delivery.

C) Removal of Existing Headers

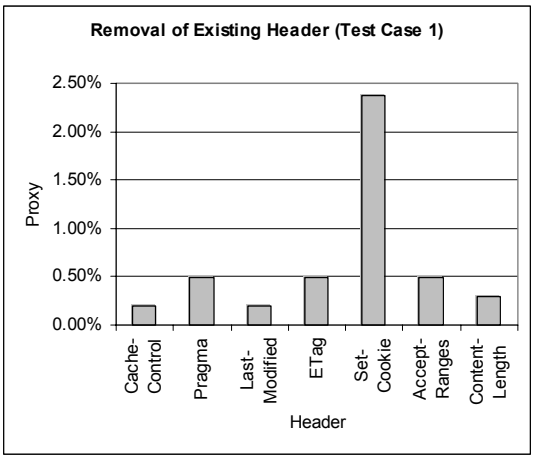

Figure 24: Removal of Existing Header (Test Case 1)

Figure 24 shows that 0.2% and 0.49% of proxies violate HTTP/1.1 by removing Cache-Control and Pragma header respectively. Clients receiving the response may cache the content using heuristics, which affects the freshness of content.

Another 0.2% and 0.49% of proxies remove Last-Modified and ETag headers. This renders clients to perform revalidation incorrectly, which leads to unnecessary network transfers for retrieving full content.

It is surprising to see 2.37% of proxies remove Set-Cookie header. The Cookie specification [42] states that Set-Cookie must not be cached or stored, but must be forwarded to clients. Without cookie, certain sites (such as web-based email) will not function properly.

Finally, some proxies remove Accept-Ranges and Content-Length header, but this has negligible effect.

6.4 Case 2: Testing with Bare Minimum Headers

Again, we check the consistency of headers from modification, addition and removal of headers.

A) Modification of Existing Headers

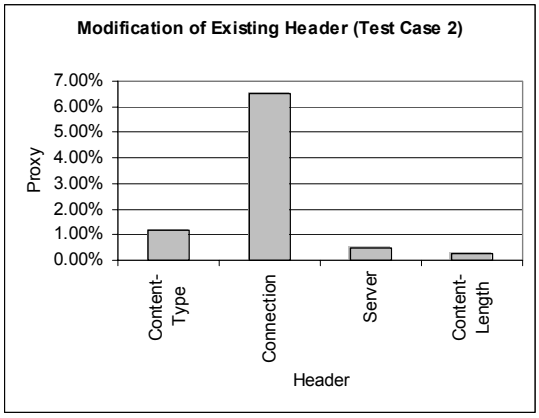

Figure 25: Modification of Existing Header (Test Case 2)

Figure 25 show modification of existing header for test case 2. It shows similar trend as case 1. The effects of the modifying Content-Type, Connection, Server and Content-Length headers are the same as described in case 1.

B) Addition of New Headers

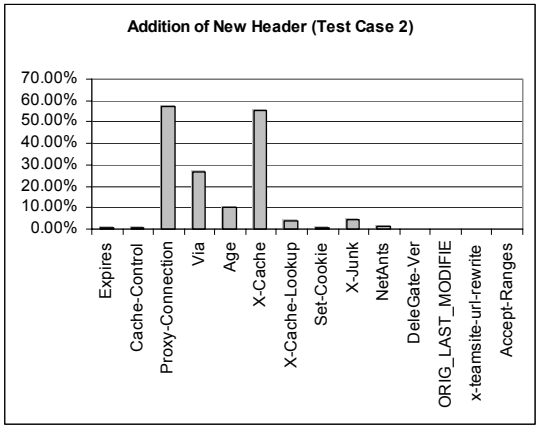

Figure 26: Addition of New Header (Test Case 2)

Figure 26 shows addition of new header for test case 2. 4 proxies (0.39%) calculate their heuristic Expires values and add the header to the response. It is arguable that since proxies cache the resource using heuristics anyway, there is no problem to reveal the heuristic Expires to clients. However, this will cause clients to perceive the Expires value is provided by the origin server. It is recommended not to add this header. The same 4 proxies also added their own Cache-Control header.

Other added headers are unimportant, as described in case 1.

C) Removal of Existing Headers

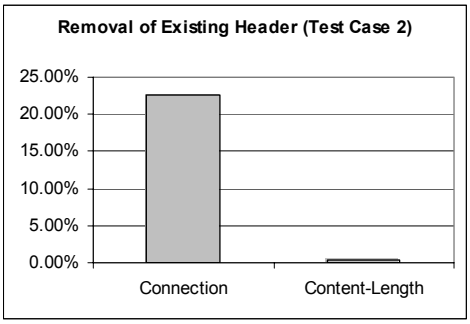

Figure 27: Removal of Existing Header (Test Case 2)

Removal of existing header for test case 2 is shown in Figure 27. Removing Connection and Content-Length headers do not cause any adverse effects.

6.5 Discussion

Results from this case study show that some web proxies are not HTTP/1.1 compliant. It is surprising to find out that proxies do not perform what it is expected to. There are 2 possible reasons for inconsistency to occur.

Firstly, the proxies could have been carelessly or wrongly configured. For example, it is surprising to observe that some proxies modify the Content-Type header to include a default character-set. This is done blindly so it is possible for the default character-set to differ from the actual one. This can lead to improper rendering of web pages on browsers, or to require installation of unnecessary "language packs". Besides that, some proxies even drop the Set-Cookie header. Without Set-Cookie header, users will not be able to access some web-based

applications such as web-based email. Caching error and performance loss also occur when caching or revalidation headers are altered or dropped.

Secondly, some proxies cache contents aggressively, that is by increasing the TTL at the risk of content staleness. This approach was more popular in the past when network bandwidth was scarce and expensive. As network bandwidth and increase tremendously, this approach is no longer popular, but we still observe this in our study.

The only related works we can find are [27, 28, 49, 50, 51, 52]. They study the cacheability of web objects and find out that some contents becomes uncacheable due to missing some header fields such as Last-Modified. In other words, caching functions cannot be correctly determined without those headers. The presence of cookie and the use of CGI scripts also decrease cacheability of objects. Our work differs from them as we investigate the current situation on consistency of web proxies. No previous work has tried to perform a real measurement like us.

Chapter 7

CASE STUDY 4: CONTENT TTL/LIFETIME

7.1 Objective

In chapter 6, we assume attributes given by servers are correct and the rest of the network (proxy caches) is expected to maintain the consistency. By contrast, in this case study, we ask a more fundamental question: "are attributes (TTL) set correctly set by content servers"?

TTL (Time-to-live) is an important attribute content server should give to the network. It defines a period of which the content is considered fresh, which is used by caches. In HTTP/1.1, TTL is expressed using the Expires header (in absolute form) or Cache-Control max-age directive (in relative form).

Unlike the previous 3 case studies, there is no one that tries to change the TTL value. For this case study, we obtain the latest content directly from the servers.

The objective of this case study is to investigate how good are servers in setting the TTL value, that is, whether TTL accurately reflect content lifetime. It is important that TTL is close to the actual content lifetime, otherwise there are 2 possible consequences. If TTL is set too conservatively (too short), then it causes redundancy in performing revalidations. On the other hand, if TTL is set to aggressively (too long), then contents become stale.

Note that we are not looking into prediction of TTL, our focus is on providing measurements of the current situation.

7.2 Terminology

TTL (Time-to-live) refers to the period of which content is considered fresh. Content can be freely cached and reused without revalidation if TTL has not elapsed. TTL is also called expiry time.

Content lifetime refers to the time between 2 successive content modifications where modification is deemed to have occurred when the content object changes.

7.3 Methodology

We use the NLANR Trace on Sep 21, 2004 as input. The following steps are taken:

1. Firstly, we filter out URL with query strings as they are sanitized and are unusable for our study.

2. Since not all URL provide explicit TTL and content modification information, the second step filters out URL without explicit TTL and content modification information.
3. For each valid URL, we obtain the content header with HTTP HEAD requests.
4. From all the responses, we examine the Expires header and Cache-Control: max-age directive.

We then gather the statistics of content TTL and plot the CDF graph as shown in Figure 28.

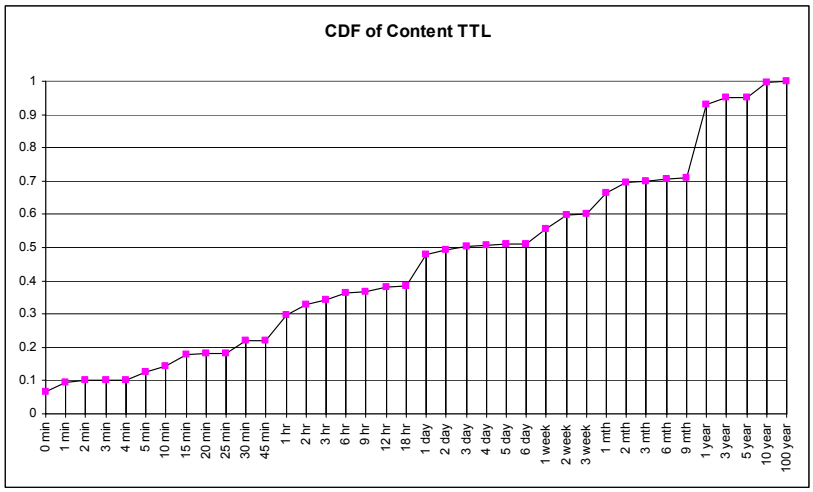

Figure 28: CDF of Web Content TTL

Since TTL can be set to very far in the future (max 68 years in our trace), we choose to only study URL with TTL less than or equal to 1 week (55.7% of all URL). The experiment was performed between Nov 9 and Dec 17 2004.

We divide the experiment into 2 phases, as shown in Figure 29. In phase 1 we monitor each URL until their first TTL while in phase 2 until the second TTL.

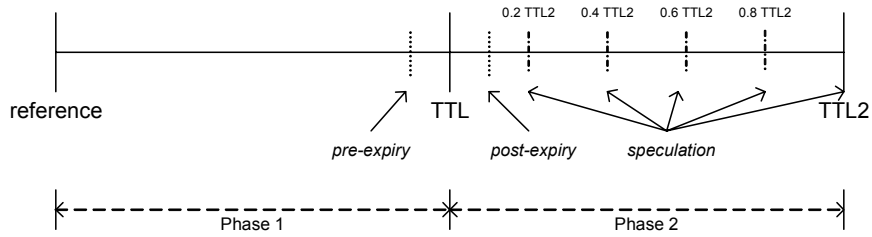

Figure 29: Phases of Experiment

7.3.1 Phase 1: Monitor until TTL

In phase 1, we obtain 3 snapshots for each URL using HTTP GET request; response headers and body are stored. The first snapshot is referred to as the reference, while the second and third snapshots are called the pre-expiry and post-expiry snapshot respectively. The pre-expiry and post-expiry snapshots are taken at $TTL - \delta$ and $TTL + \delta$ respectively, where the value of δ is defined according to the value of TTL:

- $TTL \leq 1\text{min}$: Contents with expiry equal to or less than 1 min are usually treated as uncacheable by caches. We only get the post-expiry snapshot 6 seconds after the expiry time ($TTL + 6\text{sec}$). No pre-expiry snapshot is taken.
- $TTL \leq 10\text{min}$: Content with less than 10 minutes are viewed as highly dynamic. We obtain the pre and post-expiry snapshots at $TTL \pm 10\%$ respectively.
- $TTL > 10\text{min}$: Pre and post-expiry snapshots for the remaining URL are taken at $TTL \pm 1\text{min}$.

7.3.2 Phase 2: Monitor until TTL2

In phase 1, we detected URLs that do not change at expiry (post-expiry snapshot = reference snapshot). In order to find out the modification time for these URL, we continue to monitor them until the next TTL ($TTL2$).

Snapshots obtained between TTL and $TTL2$ (inclusive) are called speculation snapshots because the intention is to capture the approximate modification time. Speculation snapshots are obtained every $20\% \times TTL2$ (with minimum 5 minutes and maximum 12 hours). The minimum value is to avoid creating denial-of-service (DOS) attack to websites with short TTL, while the maximum is to preserve a rather fine granularity of speculation. Each URL has at least 1 speculation snapshot (at $TTL2$) and at most 14 snapshots (for URL with 7 days TTL).

7.3.3 Measurements

The 2 important parameters in our experiment are content TTL and lifetime. We denote a metric called staleness or redundancy, where $staleness / redundancy = \frac{lifetime - TTL}{TTL}$. When the value is negative, we call it staleness, and if positive we call it redundancy.

- $staleness / redundancy = 0$: This means content changes exactly at the specified TTL ($lifetime = TTL$). This is the ideal case that provides good accuracy and performance.

- $staleness < 0$: This happens when content changes before TTL elapsed ($lifetime < TTL$). Users may get stale content from caches. Valid range of staleness is $-1 < staleness < 0$.
- $redundancy > 0$: This occurs when content changes after TTL ($lifetime > TTL$). Even though users do not get stale content, there is some performance loss due to unnecessary content revalidation where in fact contents do not change. Valid range of redundancy is $0 < redundancy < \infty$.

7.4 Results of Phase 1

Total URL studied	97,323	100.00%
Contents modified before TTL1	952	0.98%
- valid Last-Modified value	506	0.52%
- invalid Last-Modified value	446	0.46%
Contents modified at TTL1	1,668	1.71%
Contents modified after TTL1	94,703	97.31%

Table 26: Contents Change Before, At, and After TTL

Results from Table 26 show that only 1.71% of URLs are modified exactly at TTL1; this is the ideal case where content TTL is predicted accurately. On the other hand, 0.98% of URLs become stale as they are modified before TTL1. It is interesting to see the majority (97.31%) are modified after TTL1.

7.4.1 Contents Modified before TTL1

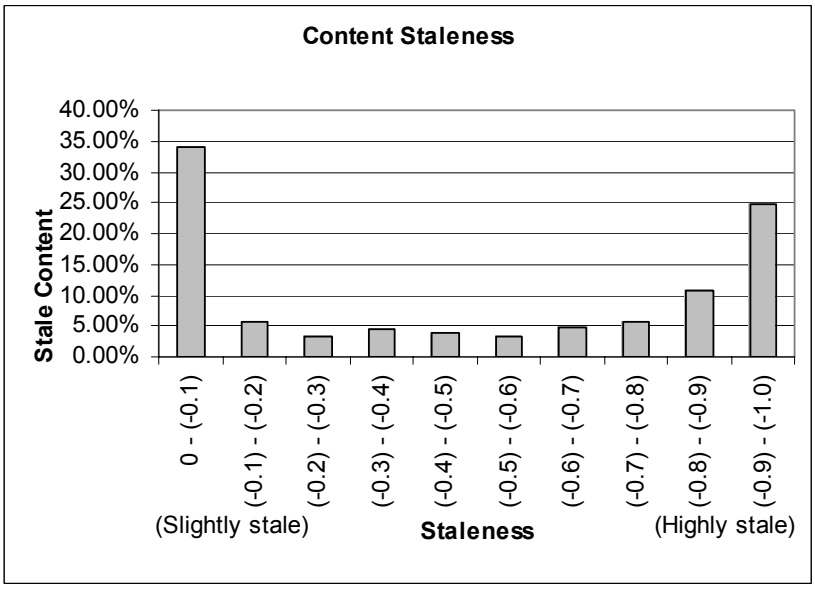

Figure 30: Content Staleness

From Table 26, we can see that 0.98% of contents are modified before TTL1. This causes cached copies of those contents to become stale.

To determine staleness of these contents, we first determine their lifetime. We do this by examining the content's Last-Modified header value, which should fall in the range of $refRT < LM \leq preRT$. The earliest possible content modification is right after the request time for reference snapshot ($refRT$) while the latest possible modification is exactly at the request time of pre-expiry snapshot ($preRT$). If the reported Last-Modified value falls outside this range, we assume the value is unreliable and the actual modification time is unknown.

After determining content lifetime, we calculate their staleness and the results are shown in Figure 30. A more detailed version of Figure 30 is shown in Figure 31 where we divide the results according to content TTL: 5min, 30min, 1hr, 12hr, 1day and 1week.

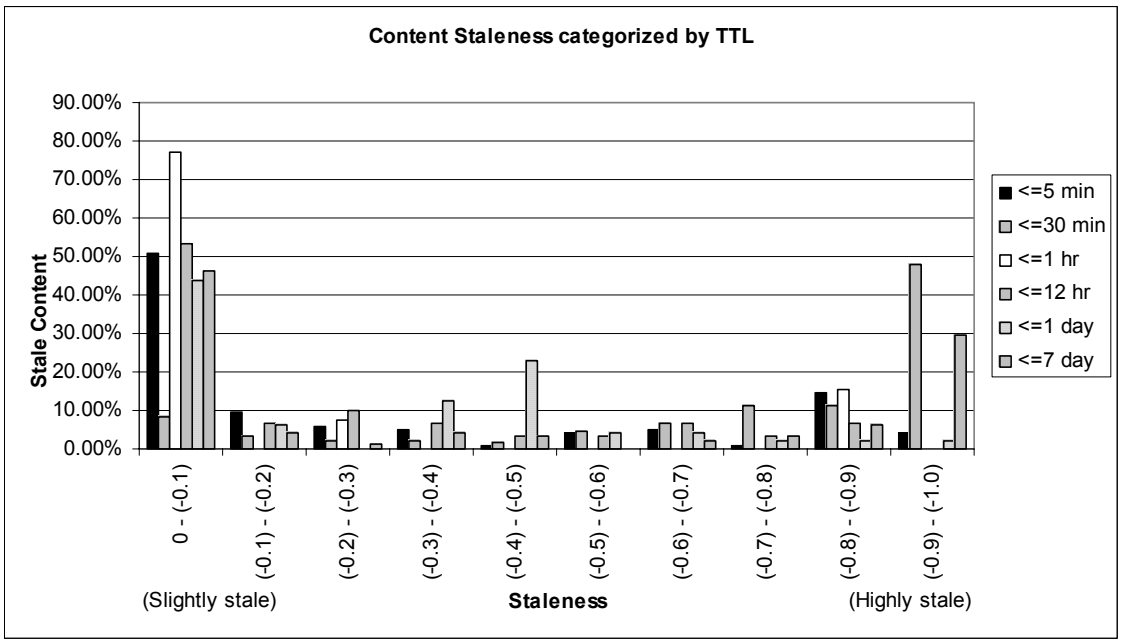

Figure 31: Content Staleness Categorized by TTL

Both Figure 30 and Figure 31 show that most staleness concentrate at both ends of the range (0 to (-0.1) and (-0.8) to (-1)). Many URLs are slightly stale as they fall in the range of redundancy 0 to (-0.1). Relatively small number of content is evenly distributed in the range from -0.2 to -0.8. Surprisingly, many stale URLs are highly stale (>=-0.9).

From Figure 31, we can see that contents with high staleness (redundancy >=-0.9) are primarily made up of contents with low TTL (<=30 min). This implies that these content are highly dynamic (probably generated on request), but the content providers assigns longer TTL than

actual lifetime in order to reduce revalidation cost, at the expense of some staleness. On the other hand, contents with high TTL (>30 min) do not seem to suffer serious staleness. Therefore, we can deduce that low TTL values (<=30 min) are usually assigned optimistically (larger than content lifetime) while high TTL values (>30 min) conservatively (less than lifetime).

7.4.2 Contents Modified after TTL1

Table 26 shows that 97.31% of content are modified after TTL1. This means the predicted content TTL are too conservative and can be set further in the future to gain more performance improvements by reducing cache revalidation. In phase 2, we attempt to find the actual content modification time by continuing to monitor these URL until the second TTL.

It is interesting to note that even when the content bodies do not change, 28.55% of URLs have either Last-Modified or ETag changed. This implies that when these contents expire, clients would not be able to revalidate successfully using conditional request, so servers return full response. It is a waste of bandwidth to retrieve full content body which did not change since previous request.

7.5 Results for Phase 2

In phase 2, we continue to monitor the URL that did not change in phase 1, until the second TTL. However, the second TTL may be different from the first. Our results show that 86.04%

of URLs have the same second TTL, but 6.91% lower and 7.05% increase their TTL. We excluded a few URLs with TTL2>7 days from our subsequent study.

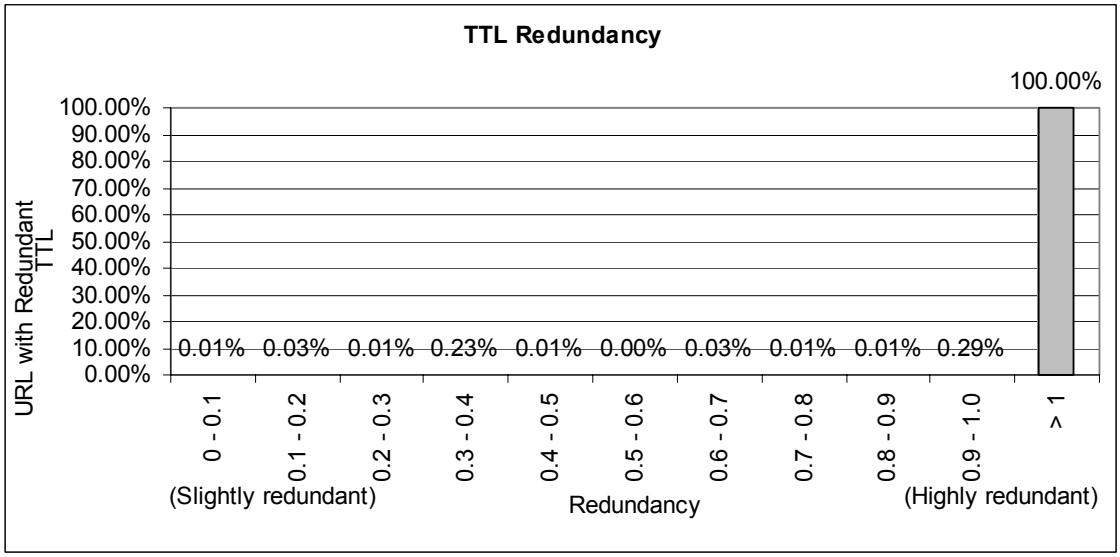

Figure 32: TTL Redundancy

The distribution of TTL redundancy is shown in Figure 32. A TTL has redundancy of 1 if the content is modified at TTL2. It is clear that even by monitoring until TTL2, very few contents changed. Most changes occur after TTL2. This implies that TTL defined by content provider are too short and this causes performance loss by increasing unnecessary cache revalidations.

7.6 Discussion

Results from this case study are less surprising than the first three. We expect TTL to differ from the actual lifetime, but how far it is from the actual?

We understand that we will waste bandwidth on revalidation by setting a conservative TTL. The question is, are we a little or too conservative? Our results should that most of the TTL values are indeed too conservative. Even though contents do not change, the TTL expire and the contents cannot be reused. It is also surprising to see that a rather high percentage of stale content are highly stale. All these observations give people the motivation to look into better TTL estimation techniques.

Related works to this case study are TTL estimation based on content modification pattern [53, 54, 27, 28, 55, 56, 57]. Most of the rules for TTL estimation are derived from statistical measures of object modification modeling. Rate of change and time sequence of modification events for individual object are most popular subjects in object dynamics characterization. Even though some of the existing works perform active monitoring of content, they only look into how good their prediction schemes work. By contrast, little has been done to study the current situation, especially on quantifying the extent of TTL redundancy and staleness.

Chapter 8

OWNERSHIP-BASED CONTENT DELIVERY

8.1 Maintaining vs Checking Consistency

In this thesis, we highlighted the problem of content consistency. The 4 case studies as well as the appropriate solutions are summarized in Table 27.

Case Study	**Appropriate Solution**
1. Replica/CDN	maintaining consistency
2. Web Mirrors	checking consistency
3. Web Proxies	maintaining consistency
4. Content TTL/lifetime	maintaining consistency

Table 27 : Case Studies and the Appropriate Solutions

There are 2 possible directions in solving content inconsistency problems. Firstly, content providers can take the active role to *maintain consistency* of cached/replicated contents. This approach usually requires close collaboration of the parties involved. For example, in CDN, origin servers tightly collaborate with replica servers and CDN surrogates so that whenever

contents are updated, origin servers push updates to all cache/replica copies. Maintaining consistency is a well studied subject; available solutions include Adaptive TTL [16], server-driven invalidation [18], Adaptive Lease [19], Volume Lease [20], ESI [21], Data Update Propagation (DUP) [22] and MONARCH [23]. Refer to chapter 2 for detailed survey of these approaches.

However, maintaining consistency cannot be easily achieved in environments such as web mirroring because web mirrors are often operated by isolated parties without any kind of cooperation. Thus, we propose the second approach – *checking consistency*. The idea is to provide users with content owner information so that users can take the active role to check the consistency of downloaded contents.

8.2 What is Ownership?

With replication and mirroring, contents are made available at multiple hosts. Even though these hosts help to distribute contents, they may not be authorized to perform other tasks such as cache revalidations on behalf of owner. A fundamental issue we have not addressed yet for the web is *ownership* of contents. Just like books have authors, contents will have owners. With ownership defined, we can answer questions such as:

- When we obtained a content from somewhere, we can go back to the owner and ask "is my copy valid"?

- When in doubt of integrity of a content, we can ask the owner "is my copy modified or corrupted"?

There are many other questions content owners can answer better than anyone else. The fundamental idea is that since contents are created and maintained by owners, they can precisely answer all questions regarding the contents. The focus our ownership approach here is to use owners for checking content consistency.

8.3 Scope

Ownership itself is a profound subject which must be thoroughly studied. In particular, we have yet to answer questions such as:

- How is ownership represented?
- Can ownership be transferred? What condition should trigger transfer of ownership?
- Can a content have more than 1 owner (multi-ownership)? What is the relationship between owners? How do owners cooperate and work together? How does validation works in the presence of multiple owners?

In this thesis, we present a simple single-ownership model. In our model, each content has only 1 owner and that ownership association is static (no transfer or change in ownership). This allows us to demonstrate the importance and value of ownership for checking content consistency, without complicating our solution.

8.4 Basic Entities

Firstly, we model a network for content delivery, making minimum assumptions about the underlying network.

> **Node:** A node (N) is a computer system on the network, identified by a globally unique identifier, *NodeID*.

> **Content:** A content (C) is a subject of interest in the network, identified by a globally unique identifier, *ContentID*.

Each content has a predefined Time-to-Live (TTL) of which the content is valid to use. ContentID should be persistent.

> **Owner:** The owner (Ow) is an entity that creates (and subsequently updates) content(s), identified by a globally unique identifier, *OwnerID*.

Typically, owner refers to an organization or individual. The OwnerID should be persistent.

> **Ownership:** The ownership (C, Ow) is a tuple that associate a content (C) with an owner (Ow).

The owner has full knowledge of the content and has complete control over all aspect of delivery of the content, such as how the content should be accessed, delivered, presented, cached, replicated etc.

When implementing ownership, we have 2 design options:

1. Tag ownership information with the content by means of its attributes, or
2. Let an ownership-manager maintain all content ownership information. Users query the ownership-manager to obtain the most updated ownership information of contents.

There are pros and cons associated with each option. The tagging option is simple and makes ownership information readily available as long as user has the content. This however requires that the ownership information persist over times (because we cannot modify ownership for contents that have been delivered). On the other hand, the ownership-manager option does not require the ownership information to be persistent, but incur overhead in additional query traffic.

8.5 Supporting Ownership in HTTP/1.1

This section describes how ownership can be supported in web content delivery – HTTP/1.1.

8.5.1 Basic Entities

First, let us map the 3 basic entities of ownership to the web, as discussed below.

ContentID of a web content is its URL.

NodeID of a host is represented by its hostname or IP address.

OwnerID is not currently defined in HTTP. We propose using content's official URL as its OwnerID. A content's official URL is the URL maintained by its author or the author's representative and we assume that each content has only 1 official URL. If the author publishes a content at more than 1 URL, he needs to choose one of the URLs as the official URL (as thus the OwnerID); other URLs are regarded as mirrors. Hereafter, the term owner and official site/URL are used interchangeably.

As described in the previous section, ownership information can either be tagged with content or externally managed by an ownership-manager. We opt for the tagging option since we do not want to introduce additional round trip when accessing or validating contents. The tagging option is deemed more efficient and suitable for the web environment.

All mirrored contents must specify OwnerID. Contents at their official URLs need not specify OwnerID.

A content's semantics is defined by its body and attributes. Therefore, mirrors should preserve the body and all HTTP headers of contents except transitional headers such as Via and Connection response-headers.

8.5.2 Certified Mirrors

To offload the owner, we propose to let owners elect trusted nodes as certified mirrors. Certified mirrors differ from other mirrors in that the mirrored contents (both attributes and

body) are certified consistent with the official site. Hence, users can download and validate with certified mirrors without any consistency issue.

We do not propose any mechanism for owner to ensure that a certified mirror is consistent with the official site. It is the responsibility of the owner to ensure that certified mirrors are consistent with the official site. Typically, the owner and the certified mirrors will have tight collaboration and the certified mirror is committed to keep all mirrored contents consistent with the owner. Among several ways to achieve consistency, the official site can push all updates to certified mirrors or certified mirrors can run in reverse-proxy configurations.

The owner elects a node as a certified mirror by granting it a mirror certificate. The certificate is a signed XML document that indicates the identity of the certified mirror, the path of mirrored contents and the validity period of the certificate. It is signed by the owner using the owner's private key, so end users can use the owner's public key to verify the certificate. Certified mirrors add a MirrorCert header into the responses for mirrored contents.

8.5.3 Validation

Validation is an important process to ensure that content is fresh and valid to use. The validation process involves 2 fundamental questions.

Firstly, who should users validate with? Users should only validate with a site that can guarantee consistency of the content. In this case, this refers to the owner and certified mirrors. Uncertified mirrors should not be used for validation (as opposed to HTTP).

Secondly, when should users validate? A content should be validated whenever it has expired or its validity is unknown (eg. when download from uncertified mirrors).

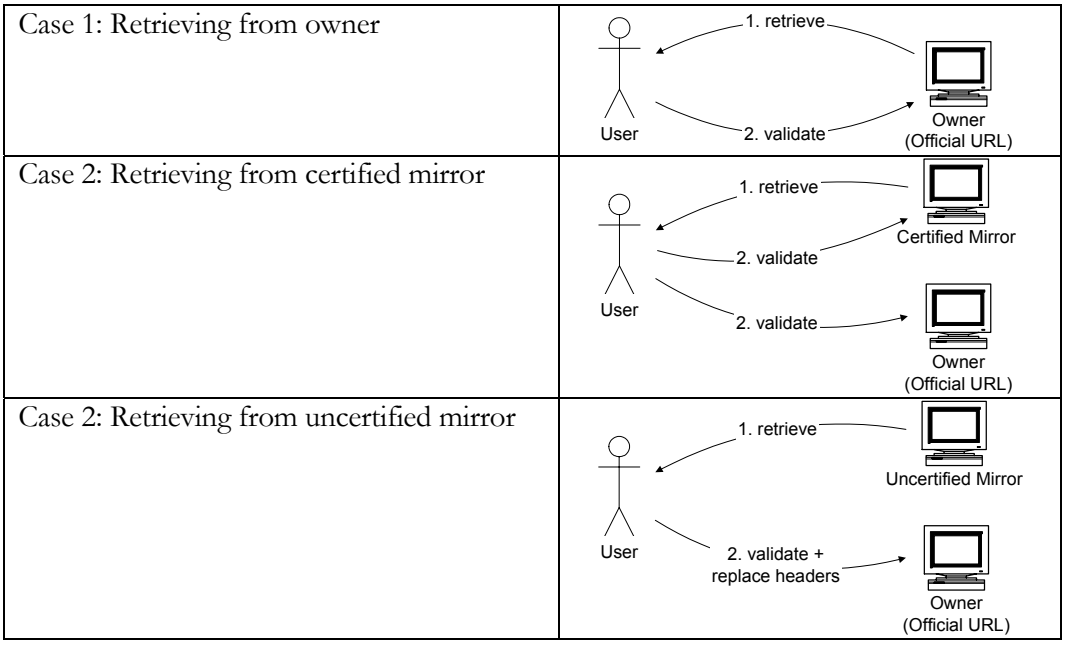

Figure 33: Validation in Ownership-based Web Content Delivery

In ownership-based web content delivery, there are 3 cases of validation depending on where the content is retrieved from, as shown in Figure 33.

In the first case, if content is retrieved from the owner, then users must validate with the owner. Users know that a site is the official site (owner) if the site does not specify the OwnerID header in the response. Note that for mirrors that does not implement the ownership (and thus does not have the OwnerID header), we would consider them as the official sites as well.

In the second case, if users retrieved a content from a certified mirror, they can safely use the content until it expires; after which users must validate with the certified mirror if mirror

certificate is still valid, or with the owner otherwise. A certified mirror is identified when the mirror specifies the OwnerID and MirrorCert header in the response and that the mirror certificate is valid. Content retrieved from a certified mirror is guaranteed consistent with the official site.

In the third case, if a content is retrieved from an uncertified mirror, validity of the content is unknown, so users should validate with the official site before using the content to ensure that the content is valid. An uncertified mirror is identified when the mirror specifies the OwnerID header in the response, but MirrorCert is absent or the mirror certificate is invalid. Validation can be done immediately after the full response-headers have been received from the uncertified mirror or after the entire content has been downloaded. Clients may postpone validation until expiry of contents, but bear the risk that the content may not be consistent with the official site. Since inconsistency of headers is a common problem, when validating with the owner, clients must replace the mirror's headers (except the OwnerID header) with those received from the owner.

An uncertified mirror should at least preserve the ETag and Last-Modified response-headers of the original content. Otherwise, using them to validate with the owner will cause validation failure and redundant transfer.

When performing revalidation, the official site will send the updated content if available. In order to offload the official site, clients should always try to download the updated content from mirror using If-Match conditional request if an ETag is given. If the mirror has the

updated content, then the client can abort the connection with official site and download from mirror instead. Otherwise, the client continues to download from official site as usual.

Protocol extensions and system implementation of ownership for HTTP/1.1 are presented in Chapter 9.

8.6 Supporting Ownership in Gnutella/0.6

The P2P architecture is growing in popularity, and has caught the attention of research communities. The most widely used P2P application today is file-sharing, which allows files to be freely shared and replicated. A popular P2P file-sharing network is Gnutella [46]. In Gnutella, all contents are assumed static (never change) and have infinite lifetime. Thus, there is no mechanism to revalidate contents. However, in reality, contents do not stay static forever or have infinite lifetime. For example, files such as source codes, books, diaries are subject to updates.

Besides content lifetime, Gnutella also does not consider ownership, which poses more problems than in the web. This is because in P2P, contents are more widely replicated - each single file may be shared by thousands of peers. When users download a content from another peer, important questions to ask are: "is it original?", "is it up-to-date?", "is it intact (not corrupted)?" etc, but Gnutella has no mechanism to answer these questions. Therefore, the consistency and quality of contents tend to degrade as contents are passed from peer to peer.

In this section, we introduce ownership and revalidation mechanisms to Gnutella/0.6.

8.6.1 Basic Entities

First, let us map the 3 basic entities in ownership to the Gnutella network.

ContentID - each content in Gnutella is identified by a simple URI which is constructed from the peer's address, fileindex and filename (eg. http://69.23.58.41/get/556/jackson.mp3). Since peers' address may change from time to time, URI is not persistent and is not a suitable candidate for ContentID. In 2002, the Hash/URN Gnutella Extensions (HUGE) [45] was proposed as identifier for contents. An URN is generated by hashing the content's body (eg. urn:sha1:PLSTHIPQGSSZTS5FJUPAKUZWUGYQYPFB). Since our goal is to allow validation for contents that may change from time to time, HUGE is unsuitable as ContentID because it changes whenever the content changes. Since existing Gnutella specification does not offer an addressing scheme that is suitable for our use, we propose to generate ContentID by concatenating OwnerID and a local ContentID (local ContentID must be unique and should be persistent within the OwnerID's namespace). The syntax for ContentID is: OwnerID "/" Local_ContentID.

NodeID of a host is represented by its hostname or IP address.

OwnerID must be globally unique and should be persistent. However, in Gnutella peers can join and leave at their wish and their addresses may change from time to time. Thus, so peer's address cannot be used as OwnerID. Since owners' availability is unpredictable, we propose that every owner elects a delegate to represent him/her in performing certain tasks. A delegate is a highly available server, with persistent NodeID (address). Each owner has only one

delegate, but each delegate can represent one or more owners each having a local owner identifier. The syntax of OwnerID is defined as: Delegate_NodeID "/" Local_OwnerID. For example, if the owner elects "delegate.com" as its delegate and the Local_OwnerID is "john", then the OwnerID will be "delegate.com/john".

8.6.2 Delegate

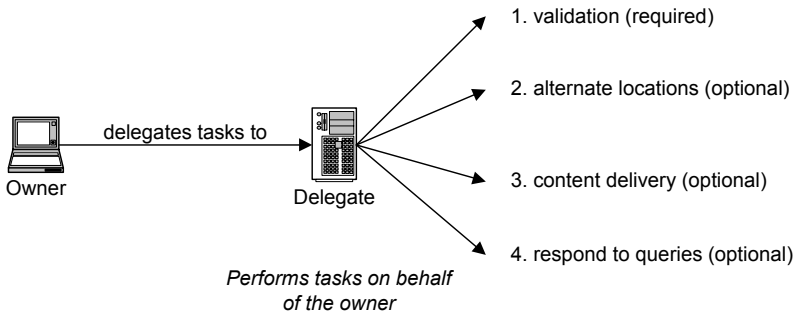

Figure 34: Tasks Performed by Delegates

Having a delegate is necessary because the owner may not always be online. The owner requires a delegate to represent him/her in performing certain task. First, let us analyze the type of tasks performed by owner when it is online:

1. Validation – we propose to introduce content TTL and a mechanism to allow peers to validate the contents they retrieve.
2. Alternate Locations – when responding to download requests, owner may send a list of alternate locations for the content using the "X-Gnutella-Alternate-Location" response header. The requesting peer then decides which peer(s) it would download the content from.

3. Content delivery – other peers can request to download contents directly from the owner. Contents are delivered to the requesting peer using out-of-band transfers.
4. Respond to queries – owner responds to queries that match its contents. To respond to queries, the owner must participate in the Gnutella network, so that query messages can be received.

The next question is which tasks should be delegated? We suggest that owner delegates "critical" tasks so that these critical tasks can still be performed even when the owner goes offline. Among the 4 tasks listed, we view validation as the most critical task. Users should always be able to validate content. Therefore, we define revalidation as a compulsory task for delegate. Whenever content is updated, the owner must submit the latest headers and validators (ETag or Last-Modified) to the delegate to allow revalidation be performed on its behalf.

The owner may also delegate the other 3 tasks depending on the requirements of the owner. We highly recommend owners to delegate the "alternate locations" task so that users can easily locate peers to download the content by asking the delegate. The "alternate locations" can be updated through the following means:

1. "Alternate locations" can be updated by peers themselves, after they have completely downloaded the content (either from the delegate or other peers), if they are willing to serve as download locations.

2. "Alternate locations" can be updated by the owner if the owner knows where the content can be downloaded from. For example, the owner may have agreements with a few peers where these peers will serve as seed download locations.
3. The delegate can periodically query the network to search for peers that have the content.
4. If the delegate also delivers the content, it can actively add peers that have downloaded the content to the list.

A delegate may internally divide alternate locations into 2 types: permanent and temporary. Permanent locations are for those explicitly marked as such by the owner, while the rest are temporary locations that will be replaced according to some replacement algorithm. The delegate may also implement some method to load-balance among the alternate locations.

The owner may delegate the "content delivery" task if the owner wanted users to be able to download the content even when it goes offline. The delegate may serve as a full-time download location, or only when the owner goes offline.

"Respond to queries" should be delegated only when "content delivery" or "alternate locations" is delegated. Delegate should not respond to queries if it cannot deliver the content or provide alternate locations for the content.

8.6.3 Validation

The Gnutella protocol only specifies how peers join the network and how to send queries. Actual transfer of contents, from sender to receiver, is done using a trimmed-down version of HTTP. Basic features of HTTP such as persistent connection, chunked encoding and range requests are supported. However, the HTTP expiration and validation model are not implemented due to the following reasons:

1. Gnutella assumes that files are static and remain unchanged once they are created. However, this assumption is inappropriate. Not all files are static, for example, software programs are subject to updates, document files are subject to editing, etc.
2. HTTP specifies that validation is performed at the host the content is retrieved from. In Gnutella, peers merely serve as download locations. The peer a content is downloaded from is not the owner of the content and has no authority to perform validation. Thus, the HTTP validation model cannot be applied to Gnutella verbatim, without appropriate modifications.

We propose to implement the HTTP expiration and validation model in Gnutella. At the same time, the validation model will be extended to include delegates. All existing HTTP caching and expiration headers are reused in Gnutella.

The proposed revalidation mechanism for Gnutella is illustrated in Figure 35. When delivering content to a requesting peer, the response carries a ContentID header, which indicates the

owner's delegate. Clients should validate with the owner's delegate, and replace the existing content headers with those delivered by the delegate.

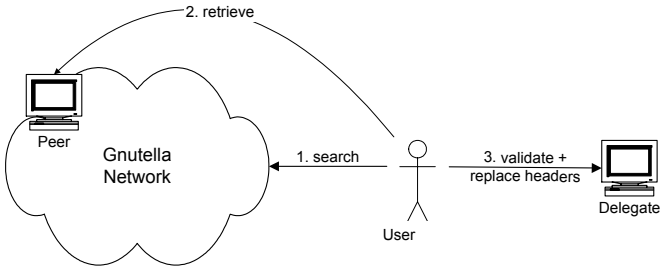

Figure 35: Proposed Content Retrieval and Validation in Gnutella

Similar to the HTTP, validation in Gnutella is performed by sending conditional requests such as If-modified-since or If-none-match. On the web, the server will deliver the updated content if the content has changed, otherwise it returns the "304 Not Modified" status. However for Gnutella, if the content has changed, the delegate may not be able to deliver the updated content if it has not been delegated the "content delivery" task by the owner. So, we proposed a new "506 Modified But Unavailable" status for this purpose.

Delegates we proposed for Gnutella is analogous to the official URL we proposed for the web. However, we do not propose to have certified mirrors for Gnutella because content delivery is already offloaded to many peers so the load on delegate is very little. Nevertheless, certified mirrors can be added easily, similar that proposed for the web.

Protocol extensions and system implementation of ownership in Gnutella/0.6 is described in Chapter 9.

Chapter 9

PROTOCOL EXTENSIONS AND SYSTEM IMPLEMENTATION

In this chapter, we propose protocol extensions and describe the system implementation to support ownership in HTTP/1.1 and Gnutella/0.6. The organization is as follows. We present the protocol extensions to HTTP/1.1 in section 9.1. Its implementation in Apache and Mozilla browser extension are then described in section 9.2; some proxy optimization techniques are also suggested. Next, section 9.3 details the protocol extensions to Gnutella/0.6 while section 9.4 describes the system implementation on the open source Limewire software. Finally, section 9.5 discusses the consistency improvements and performance overhead of the ownership approach.

9.1 Protocol Extension to Web (HTTP/1.1)

9.1.1 New response-headers for mirrored objects

We introduce 2 new response-headers to HTTP/1.1: OwnerID and MirrorCert. The OwnerID response-header is a compulsory header for any mirrored content whereas the MirrorCert response-header is required for certified mirrored contents only.

9.1.1.1 OwnerID

The OwnerID response-header field is used to specify the content's owner, which is the content's official URL. Mirrored contents must include the OwnerID response-header field in the response. It is used for validation purposes as described in section 8.5.3.

```
OwnerID syntax:
  OwnerID = "OwnerID" ":" absoluteURI
An example:
  OwnerID: http://www.officialsite.com/official/url.html
```

9.1.1.2 MirrorCert

The MirrorCert response-header field is used to specify the mirror certificate issued by the owner to the certified mirror. With a valid mirror certificate, the owner certifies that the mirror is fully consistent with the official site. User can safely use the mirrored content with guaranteed consistency.

```
MirrorCert syntax:
  MirrorCert = "MirrorCert" ":" ["type=" "embed" | "link" ","] "value="
    XMLdoc | absoluteURI
  XMLdoc = *uchar
  uchar = unreserved | escape (from RFC 1738 URL Specification)

Example:
  MirrorCert: type=link,value=http://www.mirrorsite.com/mirrorcert.xml
  MirrorCert: value=%3cXML%3e%3cowner%3e ……%3c/owner%3e……%3ec/XML%3e
```

Mirror certificate is represented in XML format. There are 2 ways to specify the mirror certificate using the MirrorCert response-header:

1. By embedding the content of mirror certificate in the response header. This is done by specifying the "type=embed" directive and the content of the certificate as the value. The "type" directive is optional, when not specified, defaults to "embed". Since a certificate (XML document) may contain reserved characters such as line feed, <, >, etc, they must be escaped to %xx format (where xx is their hex value) when specified in the response-header.
2. By providing a link to the mirror certificate. This is done by specifying the "type=link" directive and the mirror certificate's URL as the value.

9.1.2 Mirror Certificate

The purpose of a mirror certificate is to certify that the owner has delegated the task of content-delivery & validation to the certified mirror. The owner certifies that the mirrored contents are

exact replicas (both attributes & body) of the official site. The Document Type Definition (DTD) of *mirror certificate* is shown below:

```
<?xml version="1.0"?>
<DOCTYPE MC SYSTEM "MC.dtd">
<!ELEMENT Certificate (Owner, Mirror, Path+, Validity)>
<!ELEMENT Owner (#PCDATA)>
<!ELEMENT Mirror (#PCDATA)>
<!ELEMENT Path (#PCDATA)>
<!ELEMENT Validity
  From PCDATA #REQUIRED
  To PCDATA #REQUIRED>
<!ELEMENT DSS-Signature (#PCDATA)>
```

Information stored in a *mirror certificate* includes:

1. Identity of the owner, using IP address or Fully Qualified Domain Name (FQDN).
2. Identity of the mirror, using IP address or Fully Qualified Domain Name (FQDN).
3. One or more full path(s) of mirrored contents, on the mirror host. The asterisk symbol (*) can be used to match any filename, similar to how it's used in a Unix shell.
4. Validity period of the certificate states the period of which the certificate is valid. Dates are in format described in RFC1223.
5. Signature of the owner to show that the certificate is indeed generated by the owner and has not been tempered with. The signature is generated using the DSS (Digital Signature Standard) algorithm on the entire <MC:Certificate> element.

Steps to check whether a mirror URL is certified:

1. Obtain the mirror certificate and verify its integrity using the owner's public key
2. Ensure the certificate has not expired
3. Ensure the host part of the OwnerID matches the <MC:Owner> element of the certificate
4. Ensure the host part of the mirror URL matches the <MC:Mirror> element of the certificate
5. Ensure the path part of the mirror URL matches one of the paths listed in the certificate

9.1.3 Changes to Validation Model

HTTP/1.1 specifies that a content is validated with the host the content is retrieved from. We extend the validation model by introducing the ownership concept and a new validation mechanism to provide consistency guarantee. Changes to validation model are detailed in section 8.5.3 and are summarized in Table 28.

Content retrieved from	**When to validate?**	**Whom to validate with?**
Official site	1. content expires	1. official site
Certified mirror	1. content expires	1. certified mirror (if mirror certificate is valid) 2. official site (if certified mirror is unavailable)
Uncertified mirror	1. after retrieval (replace content's HTTP headers with the those sent by owner) 2. content expires	1. official site

Table 28 : Summary of Changes to the HTTP Validation Model

9.1.4 Protocol Examples

We present 2 simple examples. The first example illustrates the steps of retrieving a content from a certified mirror while the second from an uncertified mirror. The 3 URLs used in the examples are:

1. Owner - http://www.officialsite.com/official.html
2. Certified mirror - http://www.certifiedmirror.com/certified.html
3. Uncertified mirror - http://www.uncertifiedmirror.com/uncertified.html

Example 1: Suppose a client wishes to retrieve a mirrored content from the certified mirror. Request sent by the client to the certified mirror:

```
GET /certified.html HTTP/1.1
Host: www.certifiedmirror.com
```

Headers of the response sent by the certified mirror to the client:

```
HTTP/1.1 200 OK
Date: Sun, 14 Dec 2003 12:00:00 GMT
OwnerID: http://www.officialsite.com/official.html
MirrorCert: type=link, value=http://www.certifiedmirror.com/mc.xml
Last-Modified: Mon, 01 Dec 2003 12:00:00 GMT
Expires: Thu, 25 Dec 2003 12:00:00 GMT
Content-Type: text/html
```

The client detects that the content comes from a certified mirror because both OwnerID and MirrorCert response-headers are specified. A link to the mirror certificate is provided, as indicated by the type=link directive. The next step involves obtaining and verifying the mirror certificate. The client requests for the mirror certificate; content of the mirror certificate is shown below:

```
<MC:Certificate>
     <MC:Owner>www.officialsite.com</MC:Owner>
     <MC:Mirror>www.certifiedmirror.com</MC:Mirror>
     <MC:Path>/mirror/news/*</MC:Path>
     <MC:Path>/mirror/support/*</MC:Path>
     <MC:Path>/mirror/products/*</MC:Path>
     <MC:Path>/*.html</MC:Path>
     <MC:Validity From="Mon, 01 Dec 2003 00:00:00 GMT" To="Wed, 31 Dec
2003 23:59:59 GMT"/>
</MC:Certificate>
<MC:DSS-Signature>jkj4ud7cg989kshe8</MC:DSS-Signature>
```

The client first checks the signature of the certificate. If the signature is valid, the client proceeds to check that current date & time is within the validity period of the certificate. Next, the <MC:Owner> element is matched against the host part of OwnerID and the <MC:Mirror> element is matched against the host part of the mirror URL. Lastly, the path part of the mirror URL (/certified.html) is matched by the last <PC:Path> entry (/*.html). The mirrored content has been certified by the owner and thus is safe to use.

Example 2: Suppose another client wishes to retrieve a mirrored content from the uncertified mirror. Request sent by the client to the uncertified mirror:

```
GET /uncertified.html HTTP/1.1
Host: www.uncertifiedmirror.com
```

Headers of the response sent by the uncertified mirror to client:

```
HTTP/1.1 200 OK
Date: Sun, 14 Dec 2003 12:00:00 GMT
OwnerID: http://www.officialsite.com/official.html
Last-Modified: Mon, 01 Dec 2003 12:00:00 GMT
Content-Type: text/html; charset=big5
```

The client detects that the content comes from an uncertified mirror because OwnerID response-header is specified, but not the MirrorCert response-header. Validity of this content is unknown, thus the client should validate with the owner before using the content. Note that the mirror should preserve the ETag and Last-Modified value, otherwise validation with the owner will result in validation failure and redundant transfer of content body. Validation request sent by the client to the owner:

```
GET /official.html HTTP/1.1
Host: www.officialsite.com
If-Modified-Since: Mon, 01 Dec 2003 12:00:00 GMT
```

Assuming that the content has not been modified, the owner sends the response:

```
HTTP/1.1 304 Not Modified
Date: Sun, 14 Dec 2003 12:00:00 GMT
Last-Modified: Mon, 01 Dec 2003 12:00:00 GMT
Expires: Thu, 25 Dec 2003 12:00:00 GMT
Content-Type: text/html
```

The 304 status code indicates that the content has not been modified after the specified date; it also means that the (body of) mirrored content is consistent with the owner. To ensure that the HTTP headers are also consistent, the client must discard all HTTP headers it receives from the mirror (except OwnerID) and use the headers from the owner instead. For those with sharp observations, you notice that in this example, the uncertified mirror dropped the Expires header and modified the Content-Type value it received from the owner. The certified mirror in example 1 does not exhibit this consistency problem.

9.1.5 Compatibility

9.1.5.1 Compatibility with intermediaries and clients that do not support ownership

The proposed OwnerID and MirrorCert response-headers, when received by an intermediary or client that does not support ownership, will be safely ignored. A client that is not ownership-aware will validate with the host a content is retrieved from, which is the default HTTP/1.1 behavior. However, validating with an uncertified mirror may cause consistency problems and the use of an uncertified content is at clients' own risk.

9.1.5.2 Compatibility with mirrors that do not support ownership

A mirror that is not ownership-aware does not specify the OwnerID or MirrorCert response-headers. These mirrors, when accessed by clients, will be treated as the owner of contents because OwnerID response-header is absent. There is no way to differentiate a real official URL from an uncertified mirror that does not specify OwnerID. Clients validate with the

uncertified mirror (which is the correct behavior in HTTP/1.1), but bear the risk of the content inconsistency.

9.1.5.3 Compatibility Matrix

Compatibility matrix between clients and mirrors are illustrated in Table 29.

<table>
<tr><th rowspan="2">OWNERSHIP

SUPPORT</th><th colspan="2">Mirror supports</th><th rowspan="2">Mirror does NOT support</th></tr>
<tr><th>Mirror is certified</th><th>Mirror is uncertified</th></tr>
<tr><td>Client supports</td><td colspan="2">Client correctly validates with certified mirror or owner. Content consistency guaranteed</td><td rowspan="2">Client incorrectly validates with uncertified mirror. Both client and mirror are unaware of potential consistency problems.

Content consistency not guaranteed.</td></tr>
<tr><td>Client does

NOT support</td><td>Client “unknowingly” but correctly validates with certified mirror.

Content consistency guaranteed</td><td>Client incorrectly validates with uncertified mirror.

Mirror is aware of problem as it should not receive any validation requests. Mirror should redirect validation request to the owner.</td></tr>
</table>

Table 29: Mirror – Client Compatibility Matrix

9.2 Web Implementation

9.2.1 Overview

To support ownership in HTTP/1.1, modifications are needed on both server and client side. Server needs to send 2 new headers for mirrored contents: OwnerID and MirrorCert (optional). We implement this using Apache. On the client side, we have to change the way validation works in 2 aspects. Firstly, when an uncertified mirrored content is downloaded, the client can choose to revalidate with the owner. Secondly, when uncertified mirrored content expires, client will also need to revalidate with the owner instead of the mirror. We developed a Mozilla browser extension to demonstrate these features.

Web proxies are transparent to all changes on servers and clients; all contents and revalidations requested through proxies can still be cached. However, we can optimize proxies to use ownership information for better performance. This optimization is not absolutely required for ownership to work; it is only for extra performance gains. We discuss about the changes needed on proxies but do not implement them.

9.2.2 Changes to Apache

In order to add new headers to mirrored contents, we make use of Apache's built-in mod_headers module. We modified Apache's configuration file to include a section similar to the one shown below:

```
<Location /mirrors/fedora-linux/fedora-core2.ta.gz>
  Header set Owner "http:/fedora.redhat.com/download/fedora-core2.ta.gz"
  Header set MirrorCert "type=link, value=http://cert.nus.edu.sg/fedora"
</Location>
```

In this example, we use the Header directive to add Owner and MirrorCert to the response headers of mirrored content "/mirrors/fedora-linux/fedora-core2.ta.gz". The value we use for MirrorCert in this case is a link to the location of the certificate; we can also use the same directive to embed the content of the certificate in the header value.

9.2.3 Mozilla Browser Extension

A Mozilla extension is an installable enhancement to the Mozilla browser that provides additional functionality. We develop a Mozilla extension to capture outbound requests and inbound responses in order to perform appropriate revalidation for mirrored contents.

In general, Mozilla's User Interface (UI) is divided into three layers: the structure, the style, and the behavior. The structure layer identifies the widgets (menus, buttons, etc.) and their position in the UI relative to each other, the style layer defines how the widgets look (size, color, style, etc.) and their overall position (alignment), and the behavior layer specifies how the widgets behave and how users can use them to accomplish their goals.

We need no visible user interface for our system, so we created a dummy component that resides in the browser status bar. The core features are implemented in a javascript, referenced from the dummy component.

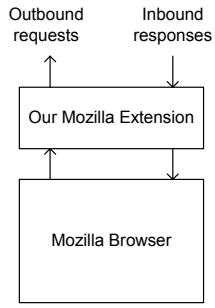

Figure 36: Events Captured by Our Mozilla Extension

In our implementation, we need to intercept HTTP responses and revalidation requests so that we can change the way validation works. Firstly, we need to receive events when Mozilla sends requests or receives responses, as shown in Figure 36. Mozilla provides this facility with the observer service. A component can register with the observer service for a given topic, identified by a string name. Later, Mozilla can signal that topic, and the observer service will call all of the components that have registered for that topic. In our case, we created an object to register with the observer service for the request and response events. The relevant code looks like:

```
var observerService = Components.classes["@mozilla.org/observer-
service;1"].getService(Components.interfaces.nsIObserverService);
observerService.addObserver(obj, "http-on-modify-request", false);
observerService.addObserver(obj, "http-on-examine-response", false);
```

The first statement is to get a reference to the Mozilla observer service. The second and third statements register the object *obj* with the observer service on topics *http-on-modify-request* and *http-on-examine-response* respectively. When the events are triggered, the observer service will call our *obj.observe()* function.

The *http-on-modify-request* event is triggered just before an HTTP request is made. The subject contains the channel which can be modified. We use this event to check if the request is a revalidation request. If it is a revalidation request for an uncertified mirrored content, we will read the OwnerID value from the content's response header (from cache) and change the destination hostname to that of the owner. The content's URI is also changed to that of the owner for revalidation to work correctly on the owner host.

On the other hand, the *http-on-examine-response* event is triggered when an HTTP response is available which can be modified by using the nsIHttpChannel passed as the subject. This is the time we check whether the content retrieved is a mirrored content by checking if the OwnerID header exists. If OwnerID exists, we further check if it is a certified mirrored content. For certified mirrored content, we verify whether the certificate is valid. For uncertified mirror or certified mirror with invalid certificate, we can optionally revalidate with the owner based on user's preference. If revalidation returns a new content (mirrored content is inconsistent), we use the *confirm()* function to notify user about the consistency problem and ask if they would like to reload the page to show the consistent content from owner.

The logic is summarized in the pseudo code in Figure 37.

```
If http-on-modify-request Then
  If revalidation request AND uncertified mirrored content Then
    Revalidate with owner
  End If
Else If http-on-examine-response Then
  If mirrored content Then
```

```
    If certified mirror Then
      Verify certificate
    End If
    If uncertified mirror OR certified mirror with invalid cert Then
      Revalidate with owner
    End If
  End If
End If
```

Figure 37: Pseudo Code for Mozilla Events

9.2.4 Proxy Optimization for Ownership

No compulsory modifications on proxy are needed in order to support ownership. However, with some optimization, proxies can achieve some performance gains by reducing cache storage requirements and saving unnecessary content downloads.

The basic idea is that when multiple mirrored copies of the same content exist, we only need to keep 1 copy in the cache. The owner's copy has the highest priority among all available copies. In this section, we discuss about the changes needed on proxies but do not implement them in our system.

9.2.4.1 Optimizing Cache Storage

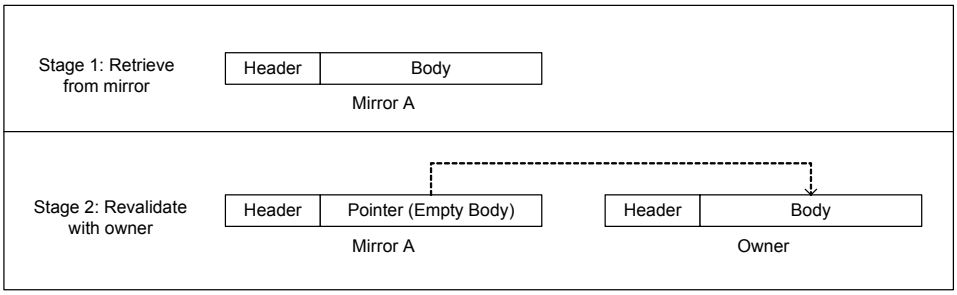

Figure 38: Optimizing Cache Storage by Storing Only One Copy of Mirrored Content

Figure 38 illustrates the cache entries in a proxy when a user downloads a mirrored content.

In stage 1, the user requests for the mirrored content. The proxy retrieves and caches the content as usual. In stage 2, the user may revalidate with owner immediately at the end of download, so a revalidation request is sent to the owner host, via the proxy. The owner responds with the status "304 Unmodified" if the mirrored content is consistent or "200 OK" together with the updated content if otherwise. In either case, the proxy now has the consistent copy, either from the mirrored content or the updated one owner sends. The proxy can now create a cache entry for the owner with the consistent content. However, the proxy no longer need to store the body of the mirrored content, it can just store a pointer to the owner's cache entry, saving considerable spaces. Even if the mirrored content is inconsistent to the owner, this does not create any problems because users are only interested in the consistent copy from the owner.

9.2.4.2 Optimizing Retrieval of Mirrored Contents

Referring to Figure 38 again, suppose a user requests for another mirrored content B. Instead of fully downloading B, the proxy can first check whether a usable copy is available in the cache using OwnerID. The proxy can send the request to mirror B, once the response headers are received, it immediate checks for the OwnerID header and uses it to locate a usable cache copy. If one is found, the proxy aborts the download of mirrored content B and uses the cache copy to respond to user. However, if no usable cache entry is found, the proxy continues to download the content as it would normally do. In essence, the OwnerID is used as a cache index in addition to URL.

9.3 Protocol Extension to Gnutella/0.6

9.3.1 New headers and status codes for Gnutella contents

We propose 2 new headers: ContentID and X-Put-Update, and a new status code "506 Modified But Unavailable" to Gnutella/0.6.

9.3.1.1 ContentID

The ContentID response-header field is used to globally identify a content. It is globally unique and independent of location of peer (unlike URL). Content owner must include the ContentID response-header when sending the content to requesting peers. The ContentID header must also be included in the response-header when the content is replicated from peer to peer.

```
ContentID syntax:
  ContentID = "ContentID" ":" OwnerID "/" Local_ContentID
  OwnerID = Delegate_NodeID "/" Local_OwnerID
  Local_ContentID = hier_path (from RFC 2396 URI Generic Syntax)
  Local_OwnerID = *uchar (from RFC 1738 URL Specification)

An example:
  ContentID: delegate.com/john/local-content7765.doc
  OwnerID: delegate.com/john
  Local_ContentID: /local-content7765.doc
```

The Local_OwnerID must be unique within the delegate's namespace. On the other hand, Local_ContentID must be unique within the local owner's namespace. The owner is free to use any algorithm to generate Local_ContentID.

9.3.1.2 X-Put-Update

The X-Put-Update request-header may be specified in a PUT request. It tells the delegate how to process the request.

```
X-Put-Update syntax:
  X-Put-Update = "X-Put-Update" ":" "headers-only" | "alt-loc-only"

Examples:
  X-Put-Update: headers-only
  X-Put-Update: alt-loc-only
```

The X-Put-Update request-header currently has 2 values defined: headers-only and alt-loc-only. If the value is headers-only, the receiving server must ignore the entity body of the request and updates the content headers to the one specified in the request headers. When using X-Put-

Update: alt-loc-only, the sender must also include the X-Gnutella-Alternate-Location header. The receiving server must ignore all other headers and entity body, and insert the URLs in the X-Gnutella-Alternate-Location to its list of alternate locations for the content.

9.3.1.3 Status Code "506 Modified But Unavailable"

When received a conditional request, a server can return the "506 Modified But Unavailable" status code if the content has been modified but the server cannot deliver the content. This status code is useful in Gnutella because a delegate is compulsory to perform validation, but may not be delegated by the owner to deliver content.

9.3.2 Validation

All HTTP/1.1 caching related headers are used according to the HTTP specification. All peers must obey, replicate and preserve these headers:

- ETag
- Last-Modified
- Pragma
- Cache-control
- Expires

Users must validate the content as long as the content is not downloaded from the delegate. Validation should be performed immediately after the content is completely downloaded. Note

that users cannot tell whether a peer is the owner because the owner participates as a normal peer on the network.

To validate a content, the user sends a conditional request (eg. If-modified-since or If-none-match) to the owner's delegate, along with the corresponding validators. The delegate compares the given validator with the latest validator provided by owner. If the content has not changed, the delegate returns the "304 Not Modified" status and includes all response-headers as if it receives a "HEAD" request. If the content has changed, there are 2 possible responses.

If the delegate is delegated to delivery content, then it replies with the usual "200 OK" status code and send the full content of the content. User must discard the response-headers sent by another peer and replace with those sent by delegate.

If the is unable to deliver the updated content, it replies with "506 Modified But Unavailable" status to inform user that the user's copy is outdated but the updated content cannot be delivered. Nevertheless, the delegate must include all response-headers as if it receives a "HEAD" request. User must discard the response-headers it receives earlier from other peer and replace with those sent by delegate. If the owner has delegated the "alternate locations" task, the delegate may insert "X-Gnutella-Alternate-Location" response-header to indicate possible download locations. After successful download from these locations, user should validate with the delegate again before using the content.

9.3.3 Owner-Delegate and Peer-Delegate Communications

Owner needs to contact delegate for the following purposes: 1) update content's response-headers or body, 2) update "alternate locations", and 3) Update the type of tasks delegated.

9.3.3.1 Updating content's response-headers (including validators) or body.

The owner updates the content headers and body by sending a PUT request to the delegate. The owner must include authentication information and the updated contents (headers and body).

If the delegate is not delegated the "content delivery" task, the owner does not need to send content body. It can update the content headers by sending the same PUT request as described above except that the entity-body is empty and the header "X-Put-Update: headers-only" must be specified to indicate that only headers are to be updated. The delegate then responds with "200 OK" if transaction is successful or "401 Unauthorized" if the authentication credentials are invalid.

9.3.3.2 Updating "alternate locations"

Owner is allowed to insert new alternate-locations, but not remove or modify existing "alternate locations" on the delegate. To insert new alternate-locations, the owner sends a PUT request containing authentication information, "X-Put-Update: alt-loc-only" header (to tell delegate to update alternate-locations only), the new locations using "X-Gnutella-Alternate-Location"

header and an empty entity-body. The delegate then responds with "200 OK" if transaction is successful or "401 Unauthorized" if the authentication credentials are invalid.

A peer can also contact the delegate if it is willing to serve as a download location. It does so by sending a PUT request similar to that of the owner except it does not provide credentials for authentication. Besides the 200 and 401 response, delegate may also return "403 Forbidden" status if peer updating feature has been disabled by the owner.

9.3.3.3 Update the type of tasks delegated

As this operation is seldom performed, we expect the owner to use an external method (such as through a web-based system) to update the delegate.

9.3.4 Protocol Examples

Suppose peer A searches for a document file named "meeting.doc" via the Gnutella network. It sends the "Query" message to the network and receives a "QueryHit" message from peer B. To download the file, peer A sends the following request to peer B:

```
GET /76543/meeting.doc HTTP/1.1
Host: 214.223.121.22
```

Headers of the response sent by the uploading peer (B) to downloading peer (A):

```
HTTP/1.1 200 OK
Date: Sun, 14 Dec 2003 12:00:00 GMT
OwnerID: http://delegate.com/bill/meeting.doc
Last-Modified: Mon, 01 Dec 2003 12:00:00 GMT
```

```
[content body follows…]
```

After peer A completely download the content, it should validate with the owner's delegate before using the content. Validation request sent by the client to the owner:

```
GET /bill/meeting.doc HTTP/1.1
Host: delegate.com
If-Modified-Since: Mon, 01 Dec 2003 12:00:00 GMT
```

If the content has changed but the owner is unable to send the updated content, the response will look like:

```
HTTP/1.1 506 Modified But Unavailable
Date: Sun, 14 Dec 2003 12:00:00 GMT
OwnerID: http://delegate.com/bill/meeting.doc
Last-Modified: Mon, 01 Dec 2003 12:00:00 GMT
Expires: Thu, 25 Dec 2003 12:00:00 GMT
Content-Type: text/html
X-Gnutella-Alternate-Location: http://65.22.45.32:6544/2343/meeting.doc,
http://202.172.32.2:6544/998/meeting.doc
```

The 506 status code indicates that the content has been modified, but the delegate cannot deliver the updated content. To ensure that HTTP headers are consistent, peer A must discard all HTTP headers it receives from peer B and use the headers from the delegate instead. Peer A may try to download the updated content from an URL specified in X-Gnutella-Alternate-Location.

9.3.5 Compatibility

Our proposed extension is fully compatible with HTTP/1.1. Features described in our proposal will be quietly ignored by peers not supporting ownership and validation. Peers who do not understand the ContentID response-header will not be able to perform validation. Thus, the validity of the downloaded content is unknown. The X-Put-Update request-header and the "506 Modified But Unavailable" status code will only be used or encountered by peers that support ownership and validation, thus there will be no compatibility problem.

9.4 P2P Implementation

9.4.1 Overview

To support ownership in Gnutella/0.6, we introduce a new entity called delegate, and propose some modifications to Gnutella peers.

The delegate we implemented performs the required "validation" and the optional "content delivery" tasks. We use the Apache web server software to perform the standard HTTP/1.1 revalidation and content delivery functions.

On the other hand, Gnutella software requires much more modifications. We develop our work on the open source Java-based Gnutella software – Limewire [47] to include these features:

1. Uploading – when Limewire uploads files to other peers, it needs to send the new OwnerID and ContentID headers as well as the HTTP/1.1 TTL related headers (Expires, Cache-Control, Last-Modified & ETag).
2. Downloading – we modify Limewire to revalidate with delegates to ensure downloaded contents are consistent.
3. TTL Monitoring – we extended the HTTP/1.1 expiration model to contents downloaded from Gnutella. We developed the TTLMonitor class to monitor all files in the "shared folder". When a file expires, Limewire revalidates with the owner and alerts the user if an updated version is available.

9.4.2 Overview of Limewire

The Limewire codes are neatly separated into 2 parts – core and GUI. The core handles Gnutella communications with other peers for query, routing, uploading and downloading. On the other hand, the GUI presents graphical interface to interact with users. The 2 parts communicate with each other via the *RouterService*, *ActivityCallback* and *VisualConnectionCallback* classes.

The classes responsible for networking in Limewire are shown in Figure 39. The modifications we make concentrate in the uploader and downloader classes.

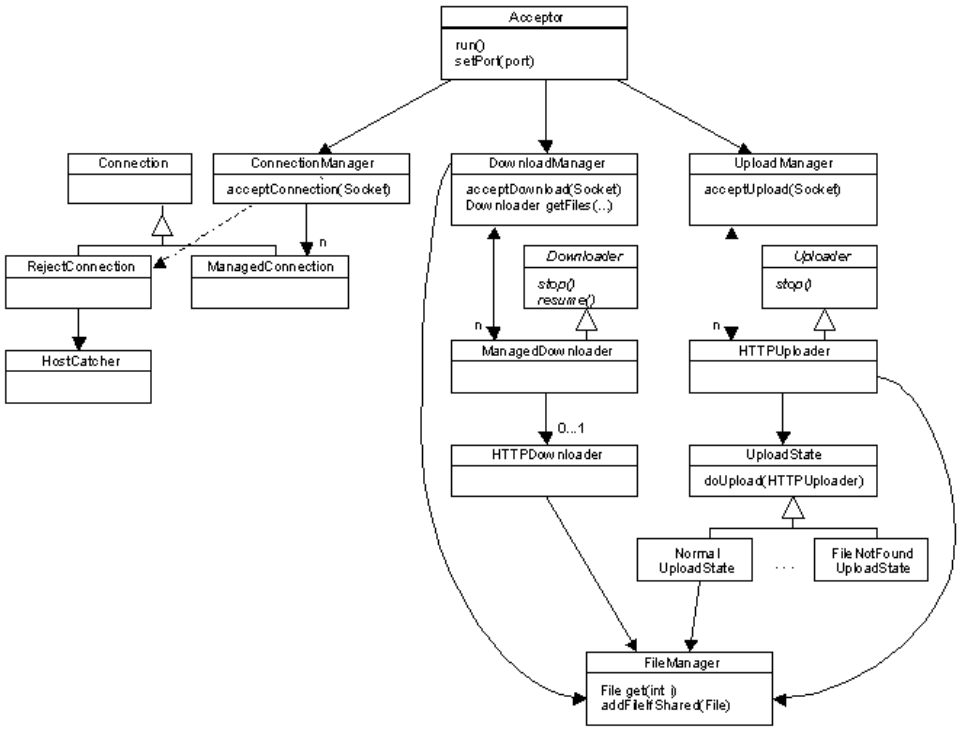

Figure 39: Networking Classes in Limewire

9.4.3 Modifications to the Upload Process

The modification we need to make to uploads is to add 6 headers (OwnerID, ContentID, Expires, Cache-Control, Last-Modified & ETag). This is done in 2 steps. Firstly, we added the 6 new headers to the *HTTPHeaderName* class, a class Limewire uses to store known headers. Secondly, we need to add these headers to contents that are being uploaded. The *writeMessageHeaders()* method of *NormalUploadState* class is used to prepare headers for uploading, so we modified this function to include the 6 headers we just added.

9.4.4 Modifications to the Download Process

Users have the option to revalidate contents with owners immediately when downloads complete. To store this preference, we add a *REVALIDATE_AFTER_DOWNLOAD* field in the *DownloadSettings* class.

In order to revalidate contents, we added a new method – *revalidate()* to the *FileManager* class. This method extracts the OwnerID, ContentID, Last-Modified and ETag values from the header and revalidate with the delegate.

9.4.5 Monitoring Contents' TTL

Existing Limewire assumes all contents never expire and therefore never checks them. Since we have added TTL related headers to each downloaded file, we can now monitor them for expiry. Web browsers normally check whether contents expire upon users' requests; that is the contents have been cached and users request them. In contrast, Limewire users do not necessarily access downloaded files via the Limewire user interface; they can directly access downloaded files via the system file browsers (eg. Windows Explorer). Hence, it is inappropriate for Limewire to perform revalidation only upon users' requests; it should actively monitor contents for expiry. We created a new *TTLMonitor* class which is responsible to check if files in the "shared folder" expire. This class is instantiated upon program execution; it reads the expiry information of each file and schedules them for revalidation.

9.5 Discussion

9.5.1 Consistency Improvements

Even though the ownership solution for web and P2P improves consistency, it is difficult to quantify the improvements it brings. Fundamentally, the ownership approach relies on mirrors or peers to provide accurate ownership information. If they give false or inaccurate ownership information, then consistency cannot be checked correctly. Therefore, instead of asking "how much consistency can the ownership approach improve", we should ask "how many mirrors or peers provide accurate ownership information".

9.5.2 Performance Overhead

Let us review the performance of our solution from two aspects: content size and latency.

Firstly, we introduced 2 new headers: OwnerID and MirrorCert (for web mirrors) or ContentID (for P2P). We calculate the overhead of the 2 headers using the statistics from the NLANR trace, as shown in Table 30.

Input traces	NLANR Sept 21, 2004
Total URL	2,131,838
Average URL Length	62 characters/bytes
Average content headers size	280 characters/bytes
Average content body size	23,562 characters/bytes

Table 30 : Statistics of NLANR Traces

For mirrored web contents, the OwnerID and MirrorCert headers will consume about 166 bytes if we assume both headers contain a URL. Likewise, P2P contents will need to include OwnerID and ContentID which occupy around 148 bytes. In these 2 cases, the content size only increases by a negligible 0.70% and 0.62% respectively.

Secondly, when mirrored contents expire, clients revalidate with owners instead of mirror hosts. Since we only change the target of revalidation, there is no extra latency overhead incurred (assuming mirror and owner has the same network latency). Nevertheless, our solution offers an option to let users revalidate with owners immediately upon retrieval of mirrored contents. If users choose to do so, there will be an additional round trip to owner to perform revalidation. On the other hand, even though certified mirrored contents do not have to be revalidated upon retrieval, users may need to retrieve the mirror certificate for verification if only the certificate link is provided. However, a certificate is usually shared among a set of mirrored contents, so the latency in retrieving the certificate will amortized among the mirrored contents.

Chapter 10

CONCLUSION

10.1 Summary

In this thesis, we study the inconsistency problems in web-based information retrieval. We then propose a novel content consistency model and a possible solution to the problem.

Firstly, we redefine content as entity that consists of object and attributes. Later, we propose a novel content consistency model and introduce 4 content consistency classes. We also show the relationship and implications of content consistency to web-based information retrieval. In contrast to data consistency, "weak" consistency in our model is not necessarily a bad sign.

To support our content consistency model, we present 4 case studies of inconsistency in the present internet.

The first case study examines the inconsistency of replicas and CDN. Replicas and CDN are usually managed by the same organization, making consistency maintenance easy to perform. In contrast to common beliefs, we found that they suffer severe inconsistency problems, which

results in consequences such as unpredictable caching behaviour, performance loss, and content presentation errors.

In the second case study, we investigate the inconsistency of web mirrors. Even though mirrored contents represent an avenue for reuse, our results show that many mirrors suffer inconsistency in terms of content attributes and/or objects.

The third case study analyzes the inconsistency problem of web proxies. We found that some web proxies cripple users' internet experience, as they do not comply to HTTP/1.1.

In the forth case study, we investigate the relationship between contents' time-to-live (TTL) and their actual lifetime. Results show that most of the time, TTL does not reflect the actual content lifetime. This leads to either content staleness or performance loss due to unnecessary revalidations.

Lastly, we propose a solution to answer "where to get the right content" based on a new ownership concept. The ownership scheme clearly defines the roles of each entity participating in content delivery. This makes it easy to identify the owner of content whom users can check consistency with. Protocol extensions have also been developed and implemented to support ownership in HTTP/1.1 and Gnutella/0.6.

10.2 Future Work

Here, we propose some directions for future work:

1. Consistency in pervasive environment – In pervasive environment, contents are transcoded to best-fit users' devices. This inherently creates multiple versions or variants for the same content. These variants are likely to differ in both attributes and object (weak consistency in our model). Consistency of any 2 variants must take into account device capabilities, user preferences, content provider policies and content quality & similarity. To efficiently support reuse and consistency, scalable or progressive data format represents an attractive solution which can be further studied.
2. Arbitrary content transformation – Besides transcoding, other types of transformations such as watermarking and translation may also be performed on content. Consistency of transformed contents is more challenging than that of transcoded content because transformed contents may not only differ in quality, but also in other aspects. To support consistency and reuse of transformed contents, some sophisticated language can be developed to annotate the operations performed on content as well as detailed instructions or conditions for reusing transformed contents.
3. Multi-ownership – In pervasive and active environments, contents may be modified by a series of intermediaries. Each of the intermediaries can be viewed as the owner of content. The issues in multi-ownership are how owners can cooperate in performing tasks efficiently, and how users should perform validation in view of multi-ownership.

Appendix A

EXTENT OF REPLICATION

It is interesting to see how many replica each replica site has, as shown in Figure 40. While majority of replica sites use only 2 replicas, we can see that there are more than 100 sites with at least 5 replicas. In our experiment, the site with the most number of replicas has 26 replicas. The more replica a site uses, the more difficult it is to maintain content consistency.

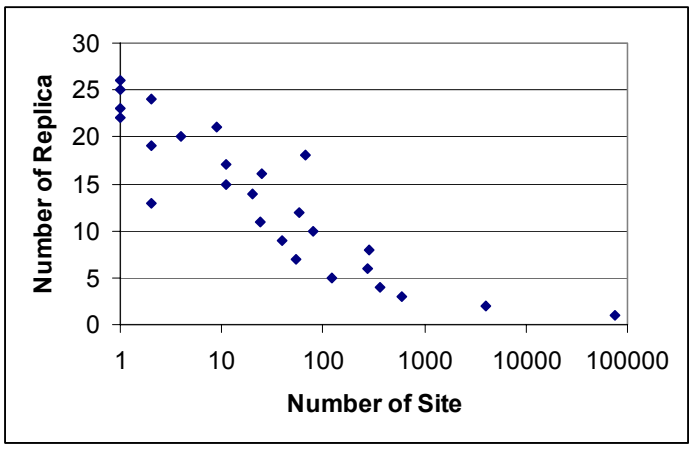

Figure 40: Number of Replica per Site

Next, we found that some replicas are used to serve many sites, as shown in Figure 41. This means some of the IP addresses are used to serve more than a site. For example, the replica

12.107.162.76 is used to serve 4 distinct sites: www.bogor.com, www.studycanada.ca, www.pekalongan.com and www.abb.com.

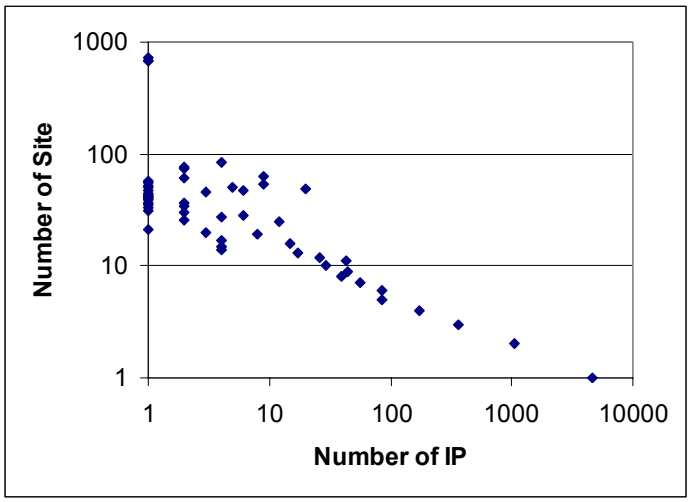

Figure 41: Number of Site each Replica Serves

We found 6597 IP addresses that serve less than or equal to 10 sites. On the other hands, there are 235 IP addresses that serve more than 10 sites, which we believe some of them are CDN "entry points".

Bibliography

1. RProxy, http://rproxy.samba.org
2. B. Knutsson, H. Lu, J. Mogul and B. Hopkins. Architecture and Performance of Server-Directed Transcoding. ACM Transactions on Internet Technology (TOIT), Volume 3, Issue 4 (November 2003), pp. 392 – 424.
3. RFC2518 HTTP Extensions for Distributed Authoring – WEBDAV, http://www.ietf.org/rfc/rfc2518.txt
4. OPES Home Page, http://www.ietf-opes.org
5. P3P Public Overview, http://www.w3.org/P3P/
6. HTTP Extensions for a Content-Addressable Web, http://www.open-content.net/specs/draft-jchapweske-caw-03.html
7. A. Fox, S. D. Gribble, E. A. Brewer, E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, 1996.
8. A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer. Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Perspectives. IEEE Personal Communications, 5(4):10-19, August 1998.

9. H. Bharadvaj, A. Joshi and S. Auephanwiriyakul. An Active Transcoding Proxy to Support Mobile Web Access. In Proc. 17th IEEE Symposium on Reliable Distributed Systems, October 24, 1998.

10. R. Han, P. Bhagwat, R. Lamaire, T. Mummert, V. Perret, J. Rubas. Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing. IEEE Personal Communications, 5(4):10-19, August 1998.

11. R. Mohan, J. R. Smith and C. Li. Adapting Multimedia Internet Content for Universal Access. IEEE Trans. Multimedia, Vol. 1, No. 1, 1999 pp. 104-114.

12. M. Hori, G. Kondoh, K. Ono, S. Hirose and S. Singhal. Annotation-based Web Content Transcoding. In Proc. of The Ninth International World Wide Web Conference (WWW9), May 2000.

13. B. Knutsson, H. Lu and J. Mogul. Architecture and Pragmatics of Server-Directed Transcoding. In Proceedings of the 7th International Web Content Caching and Distribution Workshop, pp. 229-242, August 2002.

14. C. E. Wills and M. Milkailov. Examining the Cachability of User-Requested Web Resources. In. Proceedings of the 4th International Web Caching Workshop, San Diego, CA, March/April 1999.

15. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. RFC2616 Hypertext Transfer Protocol – HTTP/1.1.

16. J. Gwertzman and M. Seltzer. World-wide Web cache consistency. In Proceedings of the 1996 Usenix Technical Conference, 1996.

17. V. Cate. Alex – a global filesystem. In Proc. of the 1992 USENIX File System Workshop, May 1992

18. C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. IEEE Transactions on Computers. 1998

19. V. Duvvuri, P. Shemoy, and R. Tewari. Adaptive leases: A strong consistency mechanism for the World Wide Web. In INFOCOM 2000.

20. J. Yin, L. Alvisi, M. Dahlin and C. Lin. Volume leases to support consistency in large-scale systems. IEEE Trans. Knowl. Data Eng, Feb 1999.

21. ESI, http://www.esi.org

22. J. Challenger, A. Iyengar, P. Dantzig. A scalable system for consistently caching dynamic web data. In Proc. of IEEE INFOCOM'99, Mar 1999.

23. M. Mikhailov and C. E. Wills. 2003. Evaluating a new approach to strong web cache consistency with snapshots of collected content. WWW 2003, May 2003.

24. A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative Leases: Mechanisms for Scalable Consistency Maintenance in Content Distribution Networks. WWW 2002, May 2002.

25. A. Myers, P. Dinda, H. Zhang. Performance Characteristics of Mirror Servers on the Internet. Technical Report CMU-CS-98-157, School of Computer Science, Carnegie Mellon University. 1998.

26. M. Makpangou, G. Pierre, C. Khoury and N. Dorta. Replicated Directory Service for Weakly Consistent Distributed Caches. In International Conference on Distributed Computing Systems, pages 92--100, 1999.

27. C. E. Wills and M. Mikhailov. Towards a Better Understanding of Web Resources and Server Responses for Improved Caching. In Proc. of the 8th International World Wide Web Conference, Toronto, Ontario Canada, May 1999.

28. C. E. Wills and M. Mikhailov. Examining the Cacheability of User-Requested Web Resources. In Proc. Of the 4th International Web Caching Workshop, San Diego, CA, March/April 1999.

29. Y. Saito and M. Shapiro. Optimistic replication. To appear in ACM Computing Surveys. http://www.hpl.hp.com/personal/Yasushi_Saito/survey.pdf

30. Z. Fei. A Novel Approach to Managing Consistency in Content Distribution Networks. In 6th International Web Caching and Content Delivery Workshop, Boston, MA, 2001.

31. Gao Song. Dynamic Data Consistency Maintenance in Peer-to-Peer Caching System. Master's Thesis, National University of Singapore, 2004.

32. J. Lan, X. Liu, P. Shenoy and K. Ramamritham. Consistency Maintenance in Peer-to-Peer File Sharing Networks. In Proc. of the 3rd IEEE Workshop on Internet Applications, 2003.

33. J. Byers, M. Luby, and M. Mitzenmacher, "Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads," in Proc. IEEE INFOCOM, vol. 1, New York, NY, Mar. 1999, pp. 275-283.

34. P. Rodriguez, E. W. Biersack. Dynamic Parallel-Access to Replicated Content in the Internet. IEEE/ACM Transactions on Networking, August 2002.

35. K. Bharat and A. Broder. Mirror, Mirror on the web: A Study of Host Pairs with Replicated Content. WWW 1999.

36. Junghoo Co, Narayanan Shivakumar, Hector Garcia-Molina. Finding replicated web collections. In Proceedings of 2000 ACM International Conference on Management of Data (SIGMOD) Conference, May 2000.

37. C. Chi and H. N. Palit. Modulation – A New Transcoding Approach (A JPEG2000 Case Study). Unpublished manuscript.

38. JPEG 2000, http://www.jpeg.org/jpeg2000/

39. MPEG Industry Forum, http://www.m4if.org/mpeg4/

40. Global Browser Stats, http://www.thecounter.com/stats/2004/September/browser.php .

41. Web Server Survey, http://news.netcraft.com/archives/web_server_survey.html .

42. Persistent Client State – HTTP Cookies, http://wp.netscape.com/newsref/std/cookie_spec.html .

43. V. Cardellini, P. S. Yu and Y. Huang. Collaborative Proxy System for Distributed Web Content Transcoding. In Proc. of the 9th Intl. Conference on Information and Knowledge Management, 2000.

44. C. Chang and M. Chen. On Exploring Aggregate Effect for Efficient Cache Replacement in Transcoding Proxies. In IEEE Transactions on Parallel and Distributed Systems, June 2003.

45. Hash/URN Gnutella Extensions (HUGE) v0.94 , http://cvs.sourceforge.net/viewcvs.py/gtk-gnutella/gtk-gnutella-current/doc/gnutella/huge?rev=HEAD .

46. Gnutella Protocol, http://rfc-gnutella.sourceforge.net/

47. Limewire Open Source Project, http://www.limewire.org

48. T. Kerry and J. Mogul. Aliasing on the World Wide Web: Prevalence and Performance Implications. In Proc. of the 11th International World Wide Web Conference, 2002

49. S. D. Gribble and E. A. Brewer, System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. USENIX Symposium on Internet Technologies and Systems 1997.

50. T. Koskela, J. Heikkonon and K. Kaski, Modeling the Cacheability of HTML Documents, Proc. Of the 9th WWW Conference, 2000.

51. X. Zhang. Cacheability of Web Objects, Master Thesis of Computer Science Department in Boston University, USA, 2000.

52. L. W. Zhang, Effective Cacheability, Internet Technical Report, National University of Singapore, 2002.

53. V. N. Padmanabhan and L. Qiu, The Content and Access Dynamics of a Busy Web Site: Findings and Implications, SIGCOMM 2000: 111-123.

54. F. Douglis, A. Feldmann, B. Krishnamurthy and J. Mogul, Rate of Change and other Metrics: A Live Study of the World Wide Web, USENIX Symposium on Internet Technologies and Systems, December 1997, pp 147-148.

55. P. Warren, C. Boldyreff and M. Munro, Characterising Evolution Web Sites, Some Case Studies, Proc. Of the 1st International Workshop on Web Site Evolution, WSE'99, Atlanta, GA, 1999.

56. B. Brewington and G. Cybenko, How Dynamic is the Web? Proc. of the 9th International WWW Conference, May, 2000.

57. J. Cho and H. G. Molina, Estimating Frequency of Change, Technical Report, Stanford University, 2000.

58. L. Qiu, V. Padmanabhan, and G. Voelker. On the placement of Web server replicas. In Proc. of the IEEE Infocom conference, April 2001.

59. Y. Chen, L. Qiu, W. Chen, L. Nguyen, R. H. Katz, Efficient and Adaptive Web Replication using Content Clustering, in IEEE Journal on Selected Areas in Communications (J-SAC), Special Issue on Internet and WWW Measurement, Mapping, and Modeling, 2003.

60. B. Krishnamurthy, C. Wills, and Y. Zhang, On the Use and Performance of Content Distribution Networks, in Proc. of SIGCOMM IMW 2001, California, pp. 169--182 November 2001.

61. P. Radoslavov, R. Govindan, and D. Estrin, Topology-Informed Internet Replica Placement (2001), in Proc. of WCW'01: Web Caching and Content Distribution Workshop, Boston, MA

62. S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt and L. Zhang, On the Placement of Internet Instrumentation, IEEE INFOCOM 2000, Tel-Aviv, Israel.