Improving Network Performance and Document Dissemination by Enhancing Cache Consistency on the Web Using Proxy and Server Negotiation

Felicia R. Doswell

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Dr. Scott F. Midkiff, Chair
Dr. Marc Abrams
Dr. Edward A. Fox
Dr. Denis Gračanin
Dr. Dennis G. Kafura

August 8, 2005

Blacksburg, Virginia

Keywords: Caching, Web, Proxy, Server, Coherence, Consistency

Improving Network Performance and Document Dissemination by Enhancing Cache Consistency on the Web Using Proxy and Server Negotiation

Felicia R. Doswell

(ABSTRACT)

Use of proxy caches in the World Wide Web is beneficial to the end user, network administrator, and server administrator since it reduces the amount of redundant traffic that circulates through the network. In addition, end users get quicker access to documents that are cached. However, the use of proxies introduces additional issues that need to be addressed. In particular, there is growing concern over how to maintain cache consistency and coherency among cached versions of documents.

The existing consistency protocols used in the Web are proving to be insufficient to meet the growing needs of the Internet population. For example, too many messages sent over the network are due to caches guessing when their copy is inconsistent. One option is to apply the cache coherence strategies already in use for many other distributed systems, such as parallel computers. However, these methods are not satisfactory for the World Wide Web due to its larger size and more diverse access patterns.

Many decisions must be made when exploring World Wide Web coherency, such as whether to provide consistency at the proxy level (client pull) or to allow the server to handle it (server push). What tradeoffs are inherent for each of these decisions? The relevant usage of any method strongly depends upon the conditions of the network (e.g., document types that are frequently requested or the state of the network load) and the resources available (e.g., disk space and type of cache available). Version 1.1 of HTTP is the first protocol version to give explicit rules for consistency on the Web. Many proposed algorithms require changes to HTTP/1.1. However, this is not necessary to provide a suitable solution.

One goal of this dissertation is to study the characteristics of document retrieval and modification to determine their effect on proposed consistency mechanisms. A set of effective consistency policies is identified from the investigation. The main objective of this dissertation is to use these findings to design and implement a consistency algorithm that provides improved performance over the current mechanisms proposed in the literature. Optimistically, we want an algorithm that provides strong consistency. However, we do not want to further degrade the network or cause undue burden on the server to gain this advantage. We propose a system based on the notion of soft-state and based on server push. In this system, the proxy would have some influence on what state information is maintained at the server (spatial consideration) as well as how long to maintain the information (temporal consideration). We perform a benchmark study of the performance of the new algorithm in comparison with existing proposed algorithms. Our results show that the Synchronous Nodes for Consistency (SINC) framework provides an average of 20% control message savings by limiting how much polling occurs with the current Web cache consistency mechanism, Adaptive Client Polling. In addition, the algorithm shows 30% savings on state space over-

head at the server by limiting the amount of per-proxy and per-document state information required at the server.

Acknowledgements

My studies at Virgina Tech have been supported by different organizations, both directly and in the form of Research Assistantships. I would like to thank the National Science Foundation Integrative Graduate Education and Research Traineeship (IGERT) (grant DGE-9987586) for providing financial support for my studies. I would like to thank the faculty at Virginia Tech who served as principal investigators on research projects with which I was involved. In particular, I thank Dr. Scott F. Midkiff, and other faculty for acquiring the grant and for including me in the research projects and workshops that afforded me many opportunities to meet new and interesting people.

The members of my committee have all contributed greatly to my dissertation, either directly with comments and ideas, or indirectly by providing me with the necessary background and advice for completing research work in the Computer Science field. I thank Dr. Midkiff, my advisor, for so generously giving me the expert guidance and motivation to finish my degree this year. The detailed comments and suggestions on this research were quite beneficial in keeping me focused on the final goal. Special thanks are due to Dr. Marc Abrams and Dr. Srinidhi Varadarajan, the former for working with me through the early years of investigating and choosing a research topic, and the latter for serving as a bountiful source of ideas and enthusiasm and for the use of the Cluster.

Many other people deserve recognition. Among them are my family and friends for their patience, support, and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The World Wide Web (WWW or, simply, the Web) has gained tremendous popularity due to the ease with which novice users can access information from all over the world. The increased population of users and the large size of files being transmitted have resulted in concerns for different types of users. Server administrators want a manageable load on their servers. Network administrators need to eliminate unnecessary traffic, thereby allowing more bandwidth for useful information. End users desire faster document retrieval. Caching is a solution, at least in part, to these Web performance issues. To improve network performance, caches can be placed in various locations in the network: where requests originate (client or browser caches), within the network (proxy or network caches), and at servers (server caches). Of the three, proxy caches are the only ones that lessen storage requirements for client machines, reduce network traffic, and promote sharing of documents between cooperating caches within a network.

Proxy caches decrease the number of messages that enter the network by satisfying requests before they reach the server. In addition, they act as intermediate agents to allow multiple clients to quickly access a group of popular Web pages. In such an approach, requests for documents are redirected to a proxy cache rather than being serviced directly by a server. When documents are fetched from a server, copies are kept in proxy caches throughout the Web. Without caching, clients must send request packets directly to a specified server and the server then responds with packets. This message exchange limits the potential growth of the Internet. As the number of client requests and server responses increase, performance is degraded. This is generally called the *scalability problem*.

Several advantages of using Web caching are described below.

1. Web caching reduces bandwidth consumption which leads to less network traffic and network congestion.

2. Web caching reduces access latency for the following two reasons. First, frequently

accessed documents are fetched from nearby proxy caches instead of remote servers, therefore, the transmission delay is minimized. Secondly, because of the reduction in network traffic, those documents not cached also can be retrieved faster than without caching due to less congestion along the path and less workload at the server.

3. Web caching reduces the workload of the remote Web server by disseminating documents among the proxy caches within a network.

4. If the remote server is not available due to a server "crash" or network partitioning, the client can obtain a cached copy at the proxy.

5. A side effect of Web caching is that it provides a chance to analyze an organization's usage patterns. This idea is applied in the analysis of this dissertation.

Although proxy caches provide a significant reduction in latency, network traffic and server load, they also present a new set of concerns that must be addressed.

1. A major disadvantage is the return of stale documents to the client due to the lack of proper proxy updating. This issue is the focus of this research.

2. The access latency may increase in the case of a cache miss due to the extra proxy processing.

3. A single proxy may become a bottleneck. A limit has to be set for the number of clients a proxy can serve.

4. A single proxy is a single point of failure.

5. Using a proxy cache will reduce the hits on the remote server which may disappoint information providers, since they cannot maintain a true log of the hits to their pages. Therefore, they may decide not to allow their documents to be cacheable.

With cache consistency being the most prominent concern when employing proxy caches, we must determine how to manage the document changes at the server that must be propagated to the proxies. These cached copies should be updated in such a manner that all versions (or at least a high percentage) are consistent with the server copy. The technique for accomplishing this is called *cache consistency* or *cache coherence*.

Two of the most important concerns in guaranteeing consistency on the Web are the message overhead (number of messages needed to provide consistent objects to the end user) and the server state space overhead (amount of space required to determine when to notify end users that an object changed in order to maintain consistency). We have two algorithms that exemplify the extremes: Client Polling (Client Pull) and Server Invalidation (Server Push). With Client Polling, the proxy contacts the server each time it receives a request.

This causes unnecessary traffic because the requested documents may not have changed since a proxy's last access. With Server Invalidation, the server sends updates to all proxies for which it maintains information every time data changes. The server is burdened with having to maintain this state information for all proxies that have contacted it. In addition, proxies that are no longer interested in data from the server will still be contacted causing unnecessary traffic. Variations have been proposed on each algorithm that improve their performance. With Client Polling, a threshold can be given to limit how often or when the proxy will contact the server rather than on every request. The problem here is determining the threshold to use that would allow consistent copies of documents while limiting how often the server is contacted frivolously. With Server Push, leases have been proposed to limit how long proxies would receive updates. In the Server Push with Leases algorithm, the server determines how long a proxy is contacted to invalidate changed documents, but does not control what information is kept. Therefore, the server would only save on the amount of time it maintains the state information for a proxy, and not on how much state information is kept. At some point in time, the state could still be extremely large.

The current document consistency mechanism widely implemented on the Web is Client Polling where the server is contacted periodically when the client requests a document that traverses through a proxy to the server. If a document changes frequently, such as online news and sports, then a stale document may be returned if the user requests the document before it is updated. Current Web technology also allows the user to avoid inconsistent documents by specifically requesting the updated version directly from the server. However, this may introduce unnecessary exchange of messages between the client and server, especially when the server documents are changed infrequently.

1.1 Research Objectives

This research explores the use of consistency methods to effectively propagate up-to-date copies of documents from servers to requesting clients. The overall intent of this research is to define a framework for restructuring Web consistency protocols to specific conditions of the Internet. This includes understanding how resources in the Web change and identifying the problems with existing mechanisms of coherency. Optimistically, we want an algorithm that provides strong consistency. However, we do not want to further degrade the network or server performance to gain this advantage. We propose a system based on the notion of soft-state based on server push where we take advantage of the benefits of client polling and server invalidation. In this system, the proxy would have some influence about what state information is maintained as well as how long to maintain the information. In addition, the proxy would send "keep-alive" to tell the server not to remove its state information, or what information to keep. Using this algorithm, we perform a benchmark study of the performance of the new algorithm in comparison with existing proposed algorithms. Based on previous work on caching and analysis of various consistency algorithms, this research

determines how to maintain consistent copies of Web documents in a cost-effective manner. The objectives of the proposed research are the following.

1. Identify which performance measures are most useful for evaluating consistency of cached documents.

2. Quantitatively assess the cost of maintaining consistency with HTTP/1.x using measures identified in Objective 1.

3. Evaluate the feasibility of server-based invalidation as compared to client-based consistency.

4. Develop a new algorithm for consistency that results in low staleness and minimal cost to the proxy, network, and server. This includes making the necessary modifications to HTTP/1.1 in order to use the new algorithm.

5. Establish the correctness of the new algorithm.

6. Demonstrate the performance of the algorithm on benchmarks using measures from Objective 1.

7. Compare the new algorithm to the existing proposed algorithms.

These objectives are discussed in detail in Chapter 3 of this dissertation. Overall, this research provides alternatives to improve network bandwidth, reduce latency, and decrease server load while maintaining a consistent view of documents on the Web.

1.2 Motivation

The advent of the World Wide Web has lead to an exponential increase in Internet usage. This augmented usage has produced several problems related to Web traffic. A large amount of Web traffic is due to refetching of documents that have recently been retrieved, possibly by persons from the same site. This redundant traffic has resulted in high bandwidth usage, slow document retrieval, and server overload. Enhancements, such as document compression, delta encoding, reduction of protocol-specific traffic, document prefetching, and proxy caching are constantly being proposed by researchers to control this problem. Compression of documents and sending document differences in response to requests, as opposed to entire documents, is one option that reduces the number of bytes that are sent over the network. However, it has been shown that only 10% of bandwidth is saved by using these methods [38, 14]. Storing HTTP headers (in compressed binary form) separately from the body is another bandwidth improvement. Unfortunately, this would require twice as many files and demand more effort to manage them. Reducing the number of protocol messages that are

transmitted over the Internet is another cost saving solution, but only applies to reducing network traffic. Document prefetching is a technique that provides quicker access to documents for the end user by predetermining and retrieving what the user may access next. However, this technique could possibly introduce unnecessary traffic onto the network by retrieving a document that the user does not want. Finally, proxy caching is a comprehensive technique that relieves the load on the server, decreases document latency, and reduces bandwidth usage. However, the benefits are not free. Consistency among cached versions of documents becomes a concern.

Several approaches have been proposed to limit the number of stale copies of a document that occur in the Web. They include Time-To-Live (TTL), Client Polling, and Invalidation [32]. These methods will be covered in detail in Section 2.2. In considering cache consistency protocols, it is necessary to determine when changes should be propagated to proxies and whether the server or the proxy should control when modifications are available to clients. The current document consistency mechanism implemented on the Web is client polling, where the server is contacted periodically when the client requests a document. If a document changes frequently, such as online news and some sports documents, then a stale document may be returned if the user requests the document before it is updated. Current Web technology also allows the user to avoid inconsistent documents by specifically requesting the updated version directly from the server. However, this may introduce unnecessary exchange of messages between the client and server, especially when the server documents are changed infrequently.

1.3 Caching Overview

The World Wide Web is a collection of client and server processes communicating to exchange data within the Internet. The server process stores and provides documents to requesting processes. Clients are the processes that request the documents from the server. In such a system, a client would send a message to a server requesting a particular document and then wait for a response message. The server, after receiving the request, sends the document in a reply message to the client. Although simple, this process produces problems for the client, network, and server due to the increase in messages [61].

1.3.1 Types of Caches and Their Effectiveness

There are, fundamentally, three types of caches: client, network, and server caches. These cache types are based on the proximity of the cache to the client and/or server.

Client Caches

Users often browse the same pages repeatedly [67]. This type of user behavior requires caching of documents that are accessed frequently. Client caches are part of the client browsers and store documents requested by the users. This type of cache aids in the repetitious retrieval of specific documents that a given client desires, such as stock quotes or news articles. However, depending on the user, this type of cache can require quite a bit of disk space and memory, especially for large audio and video files. In addition, the cached documents are not shared among multiple users and/or sessions. Therefore, every browser session may have its own cache and the same documents may be cached several times even with a small number of users.

Server Caches

Server caches are an alternative approach to client caching. These caches are located close to the server and cache documents that are frequently requested of a designated server. This prevents the server from becoming overwhelmed with requests and limits the possibility of it being a single point of failure. Server Caching also can be used to reduce access time at a server for commonly requested documents. However, requests must still travel through the network when accessing documents. Therefore, server caches do nothing to alleviate the network as a bottleneck.

Network Caches

Network caches are located somewhere between multiple clients and multiple servers and store documents from either the client or server. This allows sharing of documents and, therefore, minimizes the amount of storage space used to cache documents at the client. This is beneficial since there is usually overlap in the set of documents requested by a large group of users [67]. Network caches also decrease network delay and server overload by satisfying requests that otherwise would have required direct access to the server [48]. Finally, network caches increase reliability by allowing access to documents when the end server is unreachable. Some network caches may require that users configure their browser to use a specified network cache.

A proxy cache is a special type of network cache that caches document requests from numerous clients within a community or subnet. One function of these Web resources is to act as intermediate agents to allow multiple clients to quickly access a group of popular Web pages. It acquires dual roles as both a client and a server. The proxy acts as a client and retrieves the document from the server. It acts as a server and forwards the document to the client when the client requests it. The users of the subnet usually share a common perspective such as being in the same department within an organization. This sharing reduces network

traffic considerably when a number of clients request the same document. There are two types of proxy caches: forward and reverse. A forward proxy stores documents from the world and responds to the requests from a client group. This means that the cache will hold only documents requested by clients within an organization, but these documents originate from outside the organization. A reverse proxy caches a certain set of client documents and serves requests to the world. This means that the cache will hold documents requested from outside the organization and these documents originate from within the organization. Forward proxy caches are the subject of study in this research effort.

1.3.2 Proxy Communication

Proxy caches provide benefits to the end user, network administrator, and server administrator by reducing the amount of redundant traffic that circulates across the Internet. But what exactly is a proxy server and how do messages get from the client to the server?

Basic Proxy Cache Description

A proxy server is a computer attached to a network which accesses frequently requested data and stores that data within a disk or other cache. Therefore, any future calls for the data will access this local copy, speeding up access and reducing download times. The basic proxy cache scheme involves a single level of intermediate caches located between clients and servers. This topology includes clients that are configured to use a proxy cache as the provider of a document. The graphical representation is much like the three level tree shown in Figure 1.1. In general, when a client requests a document, a Transmission Control Protocol (TCP) [16] connection is made to a server or a cache. When a cache is involved, the request message must include the entire Uniform Resource Locator (URL) rather than just the filename part. This is because the TCP connection is to a cache and not directly to the server. If the connection is not made to the cache, the browser displays an error. Otherwise, the cache services the request.

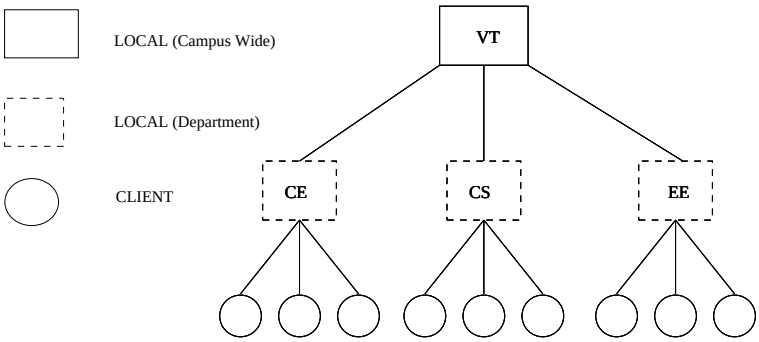

Figure 1.1: Simple Cache Hierarchy

The Message Exchange

Caches are designed to keep copies of documents presumably closer to the client (user) in order to handle repeated requests for documents. First, the client browser forwards a request to the cache it is configured to use. If the cache contains the requested document, called a *hit*, it returns the document to the requesting client. A hit occurs if the requesting client or another client, in the case of a shared proxy, has previously requested the same document using the same cache. If the cache does not have the requested document, called a *miss*, it forwards the request to the server in the same manner that a browser would if a direct connection were made to the server. In addition, the cache stores a copy of the requested document, if it was returned as a part of the response packet from the server.

The way that a proxy and server exchange the messages given above is by use of the Hyper-Text Transfer Protocol (HTTP) [28]. HTTP is a communication protocol for transferring data throughout the Internet. It operates using request and response messages between clients and servers. More specifically, the client establishes a connection to the server or proxy, and then issues a request. The server or proxy processes the request, returns a response, and then closes the connection. TCP [16] is the underlying protocol used to connect, disconnect, and reliably transfer HTTP messages. Although these messaging protocols provide the necessary capability for the generic client-to-proxy, proxy-to-server, and client-to-server communication, they also require extensions and changes for more elaborate communication.

There have been several versions of HTTP, with subsequent versions improving upon the earlier ones. HTTP/0.9 provided the basic functionality of transferring basic data. HTTP/1.0 improved upon this by allowing messages to be formatted using media types similar to the Multipurpose Internet Mail Extensions (MIME) types. MIME allows a mail message to include data other than plain text. However, HTTP/1.0 allows more extensive media types and other flexibilities. HTTP/1.1 takes into consideration the need to support caching, hierarchical proxy servers, persistent connections, and virtual hosts. This research focuses on

the extensions for caching and consistency [28]. We further discuss HTTP/1.1 in Chapter 2.

1.4 Caching Issues

Although caching provides an extensive set of benefits for Web scalability, we must deal with many issues that result from its usage.

1.4.1 Privacy and Legal Issues

Some issues that arise out of the use of proxy caches include privacy, copyright, and legal issues. These topics require concern because they affect how users and server administrators may view the use of proxy caches now and in the future.

Copyright Infringement

Caching web pages involves copying other peoples' intellectual property and may constitute copyright infringement. In order to determine whether caching constitutes a copyright infringement, we must closely examine whether caching meets the criteria for infringement and then whether cache users are eligible for the defense of fair use.

Legal Liability

Additionally, what happens if a website posts defamatory or obscene statements which it removes a few hours later, but a server has already cached the information? Now the information does not exist on the website, but it is still in the cache. If a lawsuit over the information should emerge, who should a court hold liable? Caching causes websites to lose control of their content.

1.4.2 Data Accuracy

Caches supposedly update regularly and frequently. However, no fixed schedule exists whereby caching servers guarantee to keep their cached copies current. Cautious cache servers may update every 30 minutes. Negligent cache servers may not update for days or weeks. When users request information from a website, they may receive that information from a cache. If the cache information is stale the user has received possibly outdated information and, at worst, harmful and misleading information. The degree of the threat of stale information depends on the nature of the website's content. If a user requests the Quote

of the Day, but receives yesterday's quote because the cache has not updated yet, the user suffers little harm. However, what happens if a user invests her money based on a cached page? Should she invest her money on old information?

1.4.3 Consistency

An important concern is how to keep documents in a cache consistent with the server copy. A server copy may change and the cached copy would then be *stale*. The tradeoff of using caches to enhance the Web causes further issues with how to manage the document changes that occur at the server. These modifications must be propagated to the proxies that hold copies. The HTTP/1.0 protocol provides elementary ways to deal with consistency, such as *if-modified-since* headers. However, there are even more efficient ways of extending its effectiveness. HTTP/1.1 addresses consistency by allowing the server to specify the age of a document to a client or by allowing the client to request a document only if it has been modified. Another strategy is to allow the server to initially specify the lifetime of a document when serving the client. HTTP/1.1 provides even more enhancements that are discussed later in Chapter 2. This dissertation deals with the use of the HTTP/1.1 protocol to ensure consistency among cached copies of documents.

1.4.4 Dynamic Documents

Another issue is determining which documents are cacheable. Caching is not an appropriate technique to use for dynamic Web pages. Dynamic pages are documents that change on every access, results from queries to a database, or documents that embody client-specific information (e.g., cookies). They include programs, real-time video, and audio files. Gwertzman, et al. [32], and Wooster, et al. [70] give 10% and 40%, respectively, as the percentage of documents that are dynamically created. Candan, et al. report that dynamically generated data and personalized data contributes between 30% and 40% of the total traffic [7]. Although these types of documents are increasing in occurrence on the Web, they presently cannot be cached. Therefore, the analysis described here will not include these types of documents but does require an awareness of the characteristics of dynamic documents to exclude the effect on the results.

1.4.5 Replacement Policy

A cache is limited by its finite size. Therefore, there must be ways to determine which documents are most important and, if cached, how long to remain cached. Use of shared proxy caches aids in this determination. These caches rely upon the history of requests from a group of clients and can use this information to determine the popularity of a document.

The proxy can then purge less popular documents and keep the popular ones. With the assumption that the caches have limited space, proxies are configured to use replacement algorithms to accomplish this. Replacement algorithms can improve the hit rate for the client by attempting to cache data that will most likely be referenced in the future. Research has been done to propose many cache replacement algorithms with current focus on variants of the Greedy Dual (GD) Algorithm proposed by Young [77]. Other authors have provided surveys of the different replacement algorithms and performed evaluations of their performance [13, 36, 45, 55, 60]. In general, replacement algorithms improve access time by keeping documents that take a long time to retrieve due to location, or due to an overworked server or network [68]. However, when documents are removed from a cache by a replacement algorithm, there is an impact on consistency. Replacement interferes with the cached data available, if applied before the consistency mechanism. To avoid the replacement issue, this research assumes an infinite cache size. However, several replacement algorithms are discussed further in Chapter 5 as a prelude to the design of the cache consistency algorithm developed in this research.

1.4.6 Cooperative Caching

Generally, proxy caches are independent entities that do not cooperate. A duplication of effort and unnecessary network traffic may be the end result for caches that are within close proximity. An improvement on this general scheme is to use cooperative caching [41, 57] where caches are configured to communicate with other caches within a hierarchy. This would allow caches to retrieve documents from other caches rather than the server. Ramaswamy, et. al. introduced the notion of cache clouds. A cache cloud is a group of edge caches from an edge network that cooperate among themselves to effciently deliver dynamic web content. Therefore, we have a further reduction in network traffic, and decreased delay for the user. However, the cache consistency issue is harder to tackle for so many copies. This issue is partially covered in the cache cloud implementations developed by Ramaswamy, et al. [57] because the caches in a cache cloud collaboratively share the cost of document updates. The server only needs to send a document update message to only one cache in a cache cloud, which is then distributed to other caches that are currently holding the document. However, cooperative caching will only work if an appropriate protocol is used for communication among caches and a reasonable hierarchical structure is constructed [41]. The Internet Cache Protocol (ICP) [67] provides a more efficient and flexible communication technique than HTTP, but includes no directives for consistency. In addition, although the concept is useful, hierarchical caching and cooperative caching are not in wide use today. Therefore, we do not investigate its effect. We limit our research to the impact of HTTP on caching and consistency.

1.5 Caching in Alternative Architectures

Some alternative proxy cache architectures have been the focus of investigation that improve upon the basic one discussed above in Section 1.3.1. They include the Web Proxy Auto-Discovery Protocol (WPAD) [17], Transparent Caching [40, 72], Content Distribution Networks [12, 78] and Mobile Computing [42, 37, 66]. The deployment of these alternative architectures remains somwhat limited. Therefore, we mainly focus on the basic architecture for proxy caching in our discussion of cache consistency. However, many of the solutions for cache consistency management can be applied to these other architectures as well.

1.5.1 Web Proxy Auto Discovery Protocol (WPAD)

The basic proxy caching architecture is the most widely employed. It however requires that all the client web browsers be manually configured to direct their requests to a dedicated proxy cache. Subsequently, if the proxy is no longer available, all clients have to reconfigure their browsers, which is cumbersome. One solution to this problem is browser auto-configuration using script files. In addition, the Internet Engineering Task Force (IETF) has developed the Web Proxy Auto-Discovery Protocol (WPAD). WPAD relies on resource discovery mechanisms, including domain name service (DNS) records and Dynamic Host Configuration Protocol (DHCP), to automatically locate a proxy configuration file [17]. With WPAD, the attacker has increased possibilities for posing a threat to Internet security. HTTP proxies have access to security-related information, personal information about individual users and organizations, and proprietary information belonging to users and content providers. Therefore, an attacker who wishes to intercept an organisation's web traffic could achieve this goal by compromising its proxy server(s). This is a concern for proxies in general. This problem is increased for users of the WPAD protocol. The only thing needed is for a hacker to set up a phony proxy, inject a malicious proxy configuration into the target browsers, and then try to make the target browsers use the phony proxy [53].

1.5.2 Transparent Caching

A web cache is said to be transparent if the clients can access the cache without the need to configure their browsers, for example, without any need of either a proxy auto-configuration URL or a manual proxy setting. Transparent caches filter all HTTP requests from all outbound Internet traffic, and redirects them to appropriate proxies. It not only solves the proxy configuration problem, but also enables additional administrative controls, e.g., adaptively distributing requests among multiple caches. When a client sends a request for a web site to a router, the router forwards this request to the transparent cache rather than to the server. A transparent cache accepts all request routed to it. The transparent cache sends reply messages to the client for requests, and thus disguises itself as the original server.

The clients do not have to be aware of the existence of the proxies. Most Internet Service Providers (ISPs) prefer transparent caches, since these caches do not require action from the clients. Even with these benefits, HTTP filtering is still computation-intensive, resulting in higher latency for Web browsing. More importantly, it violates the end-to-end principle as the clients' requests to the origin server are now unconsciously intercepted by the transparent proxies. This may be a problem when a client requires states to be maintained throughout successive requests or during a request involving multiple objects.

1.5.3 Push Caching

Another interesting proxy caching architecture includes push caching [31] where the web server is an active participant. It involves sending information to a client without the client requesting it. Push technology describes an content delivery system where information is delivered from a origin server to a client computer based upon request parameters outlined by the client computer. A client computer subscribes to various topics provided by a content provider and as that content is created by the content provider, the information is "pushed" or delivered across the Internet to the client computer. The technology first gained popularity in the 1990s. However, most people did not find push technology useful, and it later faded into more obscure corners of software packages. It has recently gained popularity in the move to improve availability of data to clients through use of proxies.

A push caching scheme involving proxies keeps objects of interest close to its clients by dynamically mirroring objects to the proxy near the clients. Since the origin server has knowledge of the global request pattern, it can easily identify where requests originate and decide which proxy to push the objects. Servers record their access histories, and use this information to decide where to place replicas of their data. Proxy-servers are ideal candidates for accepting replicated objects because they already run cache software and are generally attached to large disks. Push-caching can distribute the load from overloaded primary servers onto proxy servers without imposing an unacceptible load because all servers caching replicated objects may refuse additional objects at any time. Distributing the load created by popular items helps the Web scale as its population grows, but it creates a potential bottleneck at the primary server for two reasons. Clients must use the primary server to locate replicas, and replicas must use the primary server to maintain consistency [44, 31, 59]. This will increase the availability of resources, but compromises network bandwidth and lacks network interoperability.

1.5.4 Content Distribution Networks

Due to enormous growth of the World Wide Web, proxy caching alone will not provide a scalable solution. This led to development of Content Distribution Networks (CDNs) [12, 78], that provide scalable and efficient dissemination of data to several clients spread

geographically within a network. CDNs are mechanisms to deliver content to end users on behalf of origin web servers which decreases latency and increase resource availability. It offloads work from origin servers by serving some or all of the contents of web pages through several replica servers that are part of the CDN. A CDN is comprised of Origin Servers at root level, Replica Servers at the next level and intermediate proxy caches at the lower level. Normally, content is delivered to clients from replica servers via intermediate proxy servers.

A common content delivery approach involves the placement of cache servers at major Internet access points around the world and the use of a special routing code that redirects a Web page request to the closest server. When the Web user clicks on a URL that is content-delivery enabled, the CDN re-routes that user's request away from the site's originating server to a cache server closer to the user. The cache server determines what content in the request exists in the cache, serves that content, and retrieves any non-cached content from the originating server. Any new content is also cached locally. Other than faster loading times, the process is generally transparent to the user, except that the URL served may be different than the one requested. CDNs provide increased availability and reduces the load on the origin server. However, current research [27, 29, 34, 52, 73] suggest enhancements to minimize the network bandwidth usage and cache consistency issues that are evident.

1.5.5 Caching in Mobile Environments

Mobile environments involve accessing data from a wireless network connection which is highly dynamic, unreliable and insecure. One of the major issues that wireless networks confront is frequent disconnections of clients which may result in loss of data packets. Caching is used in mobile networks to deal with disconnections, but it is essential to ascertain that the client cache have the consistent data. Mobile environments are prone to data inconsistency between the server and the clients. This is because the clients access data from the data server which is not physically connected to clients as they are in wired networks. The updates carried at the server may be lost by the clients if they are moving into an area where the network is not available or if they are 'sleeping' at the time updates are sent by the server. Clients may also have a local cache that can store some data that has been sent by the server for future use. This cached data should be consistent with the data at the server in order to correctly serve the user. But, limited network bandwidth, low battery power and low processing power of mobile devices make them more vulnerable to inconsistencies. Time-based mechanisms, client polling, and server invalidation consistency algorithms have been proposed to handle consistency in wireless environments [37, 42, 66] . The advantages and disadvantages given by these algorithms for the basic proxy scheme also apply in mobile environments, but we must also be concerned with how to handle lost data.

1.6 Environmental Effects on Consistency

Consistency mechanisms are affected by the users accessing the documents, the characteristics of the documents, and administrators modifying the documents. The next few subsections describe the characteristics and how they affect consistency.

1.6.1 The Effects of Document Size

From the client perspective, small document retrieval is preferred to retrieval of large ones. This is due to the delay involved in retrieving large documents. Lorenzetti, et al. [45] reveal that most requests refer to documents with size 5 KB or less. However, this does not mean large documents are not popular. They point out that software packages distributed over the Web are examples of popular large documents. This is why the content length of documents is studied in relation to consistency. Studying document size helps to determine how much network traffic is due to maintaining cache consistency.

1.6.2 The Effects of Document Type

The most popular documents distributed over the Web are reported to be graphics and HTML documents. According to Artlitt and Williamson [5], 90% to 100% of the total requests to a server can be attributed to HTML and image documents. In addition to these types of documents, sound and video are the other types of documents characterized in the work of Artlitt and Williamson [5]. In studying document types, it is important to consider the issue of whether documents are cacheable or not. At present, dynamically created documents, documents created by programs or password protected documents, are not cacheable. Therefore, these documents are not covered in this research. Document type does not impact the development of a consistency algorithm, but may have significant effect on determining what documents are most popular and therefore need to be cached.

1.6.3 The Effects of Access Rate

Users usually access data in an unpredictable manner. Some users browse through pages and pages of information with no particular intent. Then, there are users that have a specific page to retrieve and usually require fast access. However, most requests on the Web are made on behalf of human users, and like other human-computer interactions, the actions of the user can be characterized as having identifiable regularities [21]. Therefore, it is possible to estimate the pattern of accesses based on knowledge about the document being retrieved and prior access characteristics. This supports the theory given by Pitkow and Recker [55] that as access patterns change, the number of documents, the size of the cache, and the

actual documents in the cache should also change. Using the knowledge about the access rate and patterns, adaptive algorithms can be designed to handle the consistency problem. This research covers the rate of access of popular documents. The analysis of how often documents are accessed in a set period of time can lead to some prediction of its popularity and when it will most likely be accessed in the future.

1.6.4 The Effects of Modification Rate

The frequency with which documents are modified directly impacts the process of getting consistent copies of documents to proxies. Determining the pattern of modifications and the rate at which modifications are made can provide valuable information about which consistency mechanism should be used. With this information, administrators can select which algorithms to use for consistency. The challenge here is determining how best to get this information. Is it better to classify documents by content (e.g., news or business) or group them by media type (e.g., text or graphics)? We address modification rate in this work. Knowledge of how often documents are modified in the past gives some indication of when the next modification to the document may occur.

1.7 Thesis Organization

The subsequent chapters of this dissertation are organized as follows. Chapter 2 introduces previous literature on Web consistency, which includes a discussion of the message protocol, a survey of the theoretical cache consistency mechanisms included in the studies performed, and a review of analysis performed on consistency mechanisms. Chapter 3 describes the scope of the research including the problem statement, research objectives, and methodology. Chapter 4 presents the preliminary analysis necessary to determine the feasibility of using invalidation as a consistency method. Chapter 5 presents the new framework that was developed to improve upon the currently proposed consistency algorithms. Chapter 6 describes the performance analysis of the new algorithm in comparison to currently proposed algorithms. Chapter 7 discusses the conclusions, contributions, and future work related to this research.

Chapter 2

Literature Review

All web caches attempt to achieve some degree of consistency, but the approach taken to achieve it depends on the current protocol limitations, the degree of consistency desired, and the effect on the current Web infrastructure. In this chapter, we review the communication protocol currently used in the Web, the existing cache consistency approaches proposed in the literature, and the performance analysis techniques applied to evaluate the usefulness of the approaches.

2.1 The HyperText Transfer Protocol

The way that most proxy caches and servers send messages is by using the HyperText Transfer Protocol (HTTP) [47]. It provides ways to control cache consistency. The following sections describe the HTTP message format and how HTTP addresses consistency in the Web today.

2.1.1 HTTP Message Format

When a hypertext link is chosen in a Web document or a new URL is entered in the browser, an HTTP transaction occurs. In an HTTP session, the client establishes a connection to the server. It then sends a request message to the server that includes a request method, Uniform Resource Identifier (URI), and protocol, followed by header fields, and the message body, if any. The server responds with a response message to the client that includes the protocol version, status code, and the textual representation of the status code, followed by header fields and the body, if required.

The first line of a request message, the request line, defines the request method, the URI of the requested document, the protocol, and the protocol version. There are three request

methods.

GET - Retrieves the document identified by the URI.

HEAD - Returns the header information about the document without transmitting the body. This method is mostly used for testing hypertext links for validity, accessibility, and recent modification.

POST - Posts information from the client to the server. It is used for posting items such as electronic mail, news, and form fill-in input. There is no expected return data.

The URI identifies the document to be accessed. The header information of the request message includes ways of passing additional request information including modifiers and client information. The body contains the document content and is only included for the **POST** method.

2.1.2 HTTP and Consistency

Most Web caches today use the rules of HTTP/1.1 to accomplish consistency as well as message passing capability. However, it is necessary to discuss HTTP/1.0 since it paved the way for the next generation communication protocol. The next three sections describe versions 1.0 and 1.1 of HTTP and how they address consistency. In addition, the HTTP browser's role is discussed.

HTTP/1.0 and Consistency

An important feature for caching using HTTP/1.0 is a variation of the **GET** method, called the **conditional GET**. This feature allows a document to be retrieved based on whether it has been modified since the last access. The **GET** message includes an *if-modified-since* header that results in the return of a page only if its last modification date is greater than the date in the *if-modified-since* header. The reply to this **GET** will be different depending on whether the requested document has been modified since the date provided by the *if-modified-since* header. If it has been modified, the new document is returned. If it has not been modified, the server will return a *not-modified* message which instructs the proxy to return its own cached version. It is apparent that simply using the **conditional GET** every time a request is sent will provide the client with consistent documents. If we assume that most documents do change often, then employing such a technique could give the desired consistency. However, this method could also waste more bandwidth by introducing unnecessary messages onto the network and Web servers. This is a problem when most of the documents will result in the return of a *not-modified* message and the cache copy could have been sufficient in providing a consistent document. Our preliminary research analysis

on log file content indicates that 15% to 25% of server responses to **conditional GETs** are *not-modified* [24]. From our background study four years later, there is indication that the *not-modified* responses have increased to between 30% and 40%.

An alternative is to use the dates or timestamps to identify when an entity is inconsistent with the document on the origin server. A client can be notified of a document's potential staleness by using the *Expires* header. Similarly, the age of a document reveals whether a cached document is stale. These directives are used by proxies to determine if a cached document is inconsistent at some particular moment in time. If so, then the server is polled to determine if the document has changed. One problem with the *last-modified* and the *expires* mechanism is that Web server administrators often do not use these header fields, which makes it impossible for proxies to control cache consistency in this manner. Finally, the *no cache* option in the HTTP1.0 headers provide ways to tell the proxy that the document should never be cached and therefore should be directly retrieved from the server [46].

HTTP/1.1 and Consistency

In contrast, HTTP/1.1 uses entity tags (ETags) to compare two or more objects from the same requested resource. ETags are unique identifiers that are generated by the server. They change each time the document changes. In addition to these new identifiers, HTTP/1.1 still provides the functions specified in the HTTP/1.0 version of the protocol. This includes specifying what should be cacheable (public), what may be stored by caches (no-cache), an expire mechanism (max-age), and the reload operation. HTTP/1.1 also offers revalidation tags where the server tells the clients exactly how to validate the data. This includes the ability to force each cache along the path to the origin server to revalidate its own entry with the next cache in the path. In addition to the etag header fields, HTTP/1.1 adds the new header fields: If-Match, If-None-Match, and If-Range. These fields provide the same capability as the HTTP/1.0 *if-modified-since* and *not-modified* header fields.

HTTP Browsers and Consistency

In addition to the proxy, the browser cache plays a role in consistency control. The *preferences* dialog of Internet Explorer, Netscape, or Mozilla's Firefox browsers contain a "cache" setting. There are two mechanisms here that allow the user to specify when a fresh document must be retrieved from the server or whether it would be sufficient to have a copy from the browser or network cache. The client user must manually configure which proxy to use, thereby giving the permission to retrieve possibly cached documents. In addition, browsers have a cache setting where the user can specify whether it wants the browser to verify a document once per session, every time the document is retrieved, or never. If the document has an expiration time or other age-controlling directive set, the browser can determine if the document is still within the fresh period and will not contact the server. However, requiring

the document to be retrieved directly from the server will cause increased delay in getting the document which may or may not have been modified since the last retrieval. Also, the user always can use the *reload* button that is available with common Web browsers. This involves use of the *no-cache* header mentioned earlier. If the user feels that a retrieved document is not up-to-date, they can simply press the reload button and retrieve the latest copy from the server.

2.2 A Survey of Cache Consistency Mechanisms

Cached copies should be updated when the originals change or at least at some point in the time before the next access by a client. *Weak consistency* is defined as a model where a stale document might be returned to the user and *strong consistency* as the model where no stale copies are returned after modification to a document completes. Most existing Web caches provide weak consistency where a stale copy may be returned. Weak consistency mechanisms include TTL, where the client considers a cached copy to be fresh if its TTL has not expired, and client polling, where the client periodically contacts the server to verify the freshness of a cached copy. Use of such mechanisms requires the user to ensure freshness, when desired, by pressing the reload button on a browser. This causes a burden on the server as well as the user [32]. The HTTP/1.1 protocol supports a combination of TTL and client polling. Here, we present a survey of weak and strong consistency protocols that have been proposed in the literature.

2.2.1 Time-Based

Time-to-live (TTL) [32] is a weak consistency technique where *a priori* estimates of a document's lifetime are used to determine how long cached data remains valid. This method is most useful for server administrators who know when a document changes. For instance, if a news page is updated once a day at 7 a.m., the object can be set to expire at that time, and the cache will know when to get a fresh copy. With the TTL approach, each document is given a fixed "time-to-live" *expire* field associated with it. The time-to-live field is used by the cache to determine if a cached document is fresh. When the TTL elapses, the data is considered invalid. Subsequent requests of invalid data result in the client requesting the newest version from the original server. This weak consistency mechanism is simple to implement using the HTTP/1.1 *expires* field. However, it is not easy to select approximate TTL values. Too short of an interval could cause data to be unnecessarily reloaded. Too long of an interval results in increased staleness. The adaptive TTL approach is a proposed idea where the TTL value is adjusted periodically based on observations of its lifetime. This reduces the possibility of stale documents. Gwertzman and Seltzer [32] have shown that adaptive TTL keeps the probability of stale documents to below 5%.

2.2.2 Client Polling

In general, client polling is considered a weak consistency mechanism if polling is done selectively. However, if it is applied every time a proxy cache receives a request, it is a strong consistency mechanism. In the version of client polling used in the Web today [32], clients periodically check back with the server to determine if the cached documents are still valid. Each client cache sends an *if-modified-since* request to the server. The server then checks to see if the document has changed since the timestamp. If so, a status code of "200" is sent along with the fresh document. Otherwise, the server returns a code of *304 Not Modified*. The Alex filesystem [11] uses a form of client polling where an update threshold is specified to determine how frequently to poll the server. The threshold is a percentage of the document's age. The age is the time since the last access to the document. A document is invalidated when the time since the last validation exceeds $update\ threshold * document\ age$. This weak consistency mechanism is simple to implement using the HTTP/1.1 *if-modified-since* request header field. It is also fault resilient for unreachable caches. However, use of client polling can result in a cache returning stale data or invalidating data that is still valid.

Polling-every-read [75], also called client invalidation [39] and polling-every-time [10], is a version of client polling where the server is polled every time a request is sent rather than periodically. The advantage of this approach is that it is easy to implement using the HTTP/1.1 protocol. In addition, Internet Explorer, Netscape Navigator, and Mozilla's Firefox, the popular browsers mentioned above, already allow users to select this approach. However, this technique negates the need for caching because it always bypasses the cache even though the document may be cached. In this instance, client polling becomes a strong consistency mechanism at the cost of bombarding the server and network with excessive requests and responses. In addition, the user will experience delay in receiving the document every time a request is made.

2.2.3 Invalidation

Server invalidation [32], also called callbacks [75], is a mechanism where the server sends notifications to clients when a document is modified. The advantage of this approach is that it provides strong consistency while introducing fewer unnecessary messages than the other currently proposed consistency methods. However, this scheme requires the server to keep track of which clients or proxies store copies of a document. This is expensive in terms of storage overhead and processing. In addition, the list of clients maintained by the server may become out of date. Another problem is how to handle recovery when caches or servers become unavailable. If a client becomes unreachable due to failure or lost messages, the servers may have to wait indefinitely to send a new document version to a client. Another issue to consider is that invalidation requires modifications to the server and the HTTP/1.1 protocol, whereas TTL and client polling are implemented at the proxy level. There are variations of the invalidation technique proposed in the literature that minimize these costs.

Four variations of the invalidation method are described below.

Purge Invalidation

Upon receiving an invalidation request, each proxy purges its copy from its cache. On the next request, the server will be contacted. The advantage to this approach is that less bandwidth is used than attempting to send the entire new document version to each proxy. This is especially true if new changes result in long messages sent to a large number of proxies. In addition, what if 50 writes were done in a one-hour period before a read is done? We would rather delay updating the cache until it is absolutely necessary, not updating it 50 times. The disadvantage is that extra messages (invalidation, subsequent request, and subsequent response) are sent between the proxy and server in the cases where the document is subsequently requested from the proxy. If a document is not subsequently accessed, then only the invalidation is sent [10].

Update Invalidation

Along with each invalidation, the new document version is returned in the invalidation message. There is no need for the proxy to contact the server on subsequent requests. The disadvantage is the excessive overhead of sending large documents to caches that may never be contacted again for the given document [25].

Delta Invalidation

Along with each invalidation, the server sends only the revisions to each proxy, rather than the entire updated document. This would require use of delta encoding which is only beneficial if the resource being referenced has already been cached and if it has been modified. Although delta encoding minimizes the bandwidth usage, only 10% of all requests and 25% of requests for HTML files are eligible for delta encoding because many URLs are never referenced more than once and do not change often [50]. If a document is only referenced once, it would be useless to employ delta encoding because we would only send changes to a document that will never be referenced in the future. Therefore, the sending of revisions rather than full documents would rarely occur and would provide little benefit for Web documents. However, with the changing dynamics of the Web, delta encoding would decrease the number of bytes being transferred for documents that change often, if those document changes are significantly smaller than the document being transferred.

PiggyBack Invalidation

Along with each invalidation message, the server sends additional invalidations for pages that may be accessed in the future. The basic idea is for servers to piggyback on a reply to a proxy the list of resources that have changed since the last access by this proxy. The proxy invalidates cached entries on the list and can extend the lifetime of entries not on the list. This will decrease the number of subsequent request messages [39].

2.2.4 Lease-based Algorithms

Distributed file systems currently use leases [30] to address the problems with basic invalidation. With leases, every document that is sent to a client contains a lease to specify a length of time that servers will notify clients of modifications to cached data. If a document changes before the lease expires, then the server will use invalidation to notify the client of the modification. After the lease expires, the client will send a renewal request in the form of an *if-modified-since* message. The protocol is a combination of client polling and server invalidation. This decreases the amount of time that a server will have to wait to complete a write due to unreachable clients. The server only waits until the lease expires, rather than indefinitely. In addition, leases decrease the possibility of contacting obsolete clients that retrieved the document in the past but no longer access them. To be effective in the Web, a lease should be long enough to span several reads. Unfortunately, repeated accesses to an object are often spread over several minutes or more [75]. When lease lengths are shorter than the time between reads, leases reduce to client polling. If lease lengths are severely long, then they reduce the advantages noted above. In either case, the problem of maintaining state space at the server is still a concern.

Krishnamurthy and Wills [39] examine ways to improve polling-based consistency by piggybacking optional invalidation messages on other traffic between a client and server. Their volume-based approach allows delayed invalidations where servers delay object invalidation messages to clients whose volume leases have expired. Volumes are groups of files that have a lease value associated with them. In the volume lease approach, a long lease is assigned to every page and a short lease is assigned to sets of pages called volumes. Caches must renew a lease whenever either the page lease or the volume lease expires.

Yin, *et al.* [74] propose two volume lease algorithms aimed at reducing validation traffic of short leases. They use these algorithms in a comparison study in a later paper [75]. The approach assumes volume leases, where each lease represents multiple objects cached by a stand-alone proxy. They examine issues such as delaying invalidations until lease renewals. They also discuss prefetching and pushing of lease renewals. Yin, *et al.* [76] also discuss

Mechanisms for scaling leases are studied in Ninan, *et al.* [52]. They employ cooperative leases where a lease can represent multiple proxies. Rather than delaying invalidation until lease renewal, they employ a formal model, Delta consistency [52], for propagating invali-

dations. Delta consistency allows a separation of the notification frequency from the lease duration, providing additional flexibility to the server. In their renewal policies, leaders need to interact with member proxies to decide on renewals. However, if a large number of objects are serviced by only one proxy in a region and if several such objects originate from the same server, they could further optimize state and message overheads by employing a single volume lease to manage these objects.

2.3 Consistency in Other Systems

Cache consistency was an issue long before the World Wide Web made its entrance. Hardware caches and caches within distributed systems have provided some extensive coverage of cache consistency and have resulted in proven methods to solve the consistency problem. Here, we discuss why the solutions for hardware and distributed systems caches are not applicable to the Web.

2.3.1 Cache Consistency in Hardware Systems

Hardware systems can provide strict cache consistency due to smaller size [33]. In addition, hardware caches do not require attention to network and host failures. Providing strict consistency as presented in hardware systems is hard to apply to the Internet due to the need for scalability as more caches are added. Also, hardware systems require simultaneous handling of multiple reads and multiple writes by users, whereas the Internet is currently a one writer-multiple reader problem [10].

2.3.2 Cache Consistency in Distributed Systems

Distributed systems include distributed databases, file systems, and main memory. The systems that are most closely related to the Internet, in terms of cache consistency, are distributed file systems since both are implemented in software and involve access to volumes of information. However, the Web is different from distributed file systems in terms of access patterns, its larger scale, and its single point of access. The first two issues make caching more difficult in the Internet than in file systems, while the latter one makes it easier [32].

The size of Web documents vary greatly as compared to the local documents in hardware systems and distributed systems. For example, a Web text document may be 10 KB, while a video document may be over 100 MB. The number of users is far greater on the Web than for hardware systems and typical distributed systems. The distance between the requesting client and the site of the original document may be longer for the Web.

2.4 Analysis of Cache Consistency Mechanisms

Several approaches have been applied to maintaining cache consistency in the Web. There have been many algorithms developed to address strong cache consistency and to attempt to decrease the amount of space that the server has to maintain. Research efforts have included comparisons and proposals of several variants of the three major protocols: time-to-live, client polling, and invalidation. Evaluations of these protocols, covered in the prior literature, are discussed below. In addition, the characteristics of document access and modification have received some attention.

2.4.1 Evaluation of Cache Consistency Protocols

Worrell [71] did a simulation study to compare the time-to-live and invalidation algorithms. He concludes that invalidation is a better approach for cache consistency. His study includes the investigation of consistency in a hierarchical network, which already carries a higher overhead with respect to communication than a simple, flat network [71]. Contradicting the results of Worrell, Gwertzman and Seltzer [32] discuss various cache consistency approaches and conclude that a weak-consistency approach, such as adaptive TTL, would perform better for Web caching in terms of network bandwidth [32]. Cao and Liu [10] extend this work and study the implications of using other metrics like server load, response time, and message delay. These authors conclude that invalidation performs better than client polling and similar to adaptive TTL in terms of network traffic, average client response time, and server load [10]. Their study however, assumes that all documents are equally important in terms of consistency and trades this guarantee for strong consistency against the cost of increased bandwidth usage. They also do not address the overhead necessary for servers to invalidate copies. We are concerned with how current consistency mechanisms will affect network performance, particularly bandwidth, server storage, and processing.

Adaptive lease is a strong consistency mechanism based on server invalidation. It establishes different ways to determine what the lease duration should be to control how long state information is maintained for objects. The duration is computed adaptively based on the state space overhead and the control message overhead [26]. However, this approach does not correct the problem of determining how much state information is maintained at any point in time.

The Web Cache Invalidation Protocol (WCIP), proposed by Li, *et al.* [43], is a protocol that defines the propogation of server invalidations using multicast. The approach uses leaders to send invalidations and manage lease renewals for a multicast group. We propose a similiar architecture, but do not use multicast to propogate invalidations. However, some of the protocol specifications are similiar to the changes made in the HTTP protocol to implement the cooperation between proxy and server to ensure strong consistency. We utilize the existing HTTP protocol and incorporate a new caching scheme to improve the message and

server state overhead. It is our belief that small modifications to the current HTTP protocol are more easily deployed than the promotion of a new protocol. Also, multicast is not supported in many backbone networks, making the approach infeasible in today's Internet.

Cooperative lease [52], proposed by Ninan, *et al.* is another invalidation with leases technique that is used in a content distribution network environment. In this approach, a single lease is used to represent multiple proxies, and delta consistency is employed to provide a level of staleness. This follows the idea that there may be a tolerance for some minimal amount of staleness. The authors limit the rate at which proxies are notified of updates and allow the server to grant a lease to a group of proxies instead of issuing a separate lease to each individual proxy [52]. In this work, there is a limit on how often to contact the proxies for updates, but there is still no limit on how much state information is maintained. In addition, the strong consistency algorithm, invalidation with leases, is now reduced to a weak consistency algorithm since they are allowing some degree of staleness. We propose an algorithm that keeps the strong consistency, by employing the help of the proxy to determine what state information is important to maintain at the server and still allowing the server to control when to send those updates for a subset of documents.

Yu, *et al.* [78] developed a consistency scheme based on a hierarchy of caches using multicast invalidation and volume leases. The maximum and average staleness of documents were measured. In addition, the percentage of pages that are delivered stale (stale hit rate) is evaluated. Median stale rate is a metric that is proposed to further verify the analysis. In addition, their analysis is based on use of synthetically generated workloads and arbitrary read/write rates, without any foundation as to why the workload parameters were chosen. Server and proxy logs can aid in determining the rate at which documents are read and written. Then these values can be used as more realistic values in the analysis. Yu, *et al.* propose a consistency algorithm that employs "push" technology, where servers guess which pages are popular and forward them to clients that are possibly not currently accessing a volume. This improves read latency, but at the cost of increasing network traffic and proxy cache overhead. The main objective is to reduce the number of redundant messages on the Web without allowing the consistency mechanism to increase the message count. According to Yin, *et al.* [75], a client-driven technique for these clients is better, where the server waits for the clients to access volumes of data combined with the lease capability.

Duvvuri, *et al.* [26] suggest an adaptive lease scheme where the load of the server is observed and considered when lease renewal time arrives. They analytically compare several algorithms to determine the level of user satisfaction, client response time, and stale rate (percentage of pages delivered that are stale) of documents. None of the research involving algorithm development uses live network traffic, or even captured proxy and server logs. Most of the research uses synthetic workloads to simulate data. Of the three cited works, [78], [75], and [26], Duvvuri, *et al.* [26] justify the parameters and workloads used for measuring consistency algorithms. Unfortunately, they based one of their analytical results on previous work of Gwertzman, *et al.* [32], which was contradicted by Wills, *et al.* [69] and Douglis, *et al.* [25] who reveal that popular documents are not modified often. We propose

an algorithm that makes use of the access patterns to the proxy cache and the write pattern of the server to dynamically ensure consistency.

Deolasee, *et al.* [23] examine methods that combine pull and push based models. They present approaches that adaptively choose between pull and push based models. In this case, the proxy mainly is responsible for pulling updated data from the server, while the server may push additional updates [23].

2.4.2 Effect of Web Access Patterns on Cache Consistency

Douglis, *et al.* [25] performed a study to determine the rate at which documents change. Their main goal was to uncover how the modification rate affects systems that use delta-encoding. This information is also important to cache consistency. If we can detect that modifications occur at regular intervals, then we can predict which algorithm will perform better given the rate of modifications. In their study, Douglis, *et al.* collected traces at the Internet connection of two corporate networks and analyzed this data for a rate of change. This data set restricts the study to measuring requests at a specific point in time (when it was collected) [25]. Similarly, statistics were collected only when a reference to a resource occurred. This does not account for documents that are accessed once and never accessed again. It is important to know how often such documents change. Measuring the data periodically using various predetermined intervals is an alternative approach to determine how often documents change. This is a better approach for determining how to set the TTL value of a document. Douglis, *et al.* [25] also discuss how often modifications occur when affected by certain characteristics (e.g., content-type and number of references). Since people generally determine what they will access based on categories (e.g. news or business) rather than content type (e.g., text, video, or image), it is important to investigate chracteristics based on the category of links accessed. Another realistic perspective for which to measure access is to consider the time of day that modifications occur. Both of these options take the users' habits into consideration.

Wills and Mikhailov [69] did a similar study to determine the nature and rate of change of documents using URLs rather than testing log samples. They detect differences in documents by calculating the Message Digest 5 (MD5) Checksum [58] on the contents of each document to determine whether the document change between successive retrievals [69]. The MD5 algorithm takes text as input and produces as output a 128-bit value representing the input. This output is called the message digest. If you change a single bit in the input, the message digest is usually different. It is conjectured that it is computationally infeasible to produce a collision, two different messages having the same message digest. However, the MD5 checksum algorithm has been found to produce collisions when computing hash functions [22]. Also, the MD5 checksum is normally used to verify data integrity not to compute differences between files. The UNIX DIFF function works well for that application. Therefore, it would be better to use the UNIX DIFF or CMP commands to determine the difference among

documents. These commands provide less possibility of error than the computation of the MD5 checksum. In addition, CMP allows the user to compare binary content of images.

Wills and Mikhailov [69] used ".com," ".net," and search (which are special types of ".com") categories in their analysis. More realistic categories of business, family, news, and search should also be investigated. The research data of [69] was collected over a one-day period. To have a random sample, a longer collection period is required. We consider these options in our preliminary study discussed in Chapter 4.

2.5 Chapter Summary

Based on the current literature, it has been shown that cache coherency can be provided in the World Wide Web using various methods. However, there has not been much effort in examining when and how to best apply each of the methods. Such a solution would lead to discovery of a dynamic protocol, employing a combination of existing coherence methods to be triggered based on necessity.

Most research involving consistency on the Web has focused on evaluating the three basic algorithms: TTL, client polling, and invalidation. These studies include the assumption that all documents are equivalent in size, access rate, or modification rate. These document access characteristics are important in determining when a consistency mechanism is best applied and, therefore, should be considered in analyzing the methods available. Also, the researchers use simulation to evaluate the algorithms and generally base their conclusions on a single cost measure. One researcher may use bandwidth, while another uses server load. First, it is important that actual network components and realistic models of traffic are used in measuring these techniques to provide a practical solution. This ensures that the software solution to cache consistency will work within real networks. Second, the use of a single cost measure gives a false sense of the benefits of mechanisms because all possible costs were not considered. It is imperative that all potential costs are measured using the same workload and method to permit comparisons. Most importantly, all current research ignores the potential for existing and proposed consistency mechanisms to further degrade the network. This can occur when too many messages are sent in an attempt to control consistency. Researchers, such as Worrell [10, 71, 8, 49, 52], compare the mechanisms using a cost measure and then recommend the one that works best. But, maybe that better mechanism could threaten network scalability. This dissertation describes work that identifies the potential of several previously proposed algorithms using a consistent set of cost measures with a real network. The mechanisms used depend upon the nature of document retrieval i.e., how often documents change or how often they are requested. Therefore, given a scenario based on these document access characteristics, recommendations of an efficient consistency algorithm can be made.

Chapter 3

Scope of Research

This chapter discusses the problems addressed in this research and the methodology employed. The first section, Section 3.1, explains the problem statement for this work. Section 3.2 reiterates the goals that are accomplished by this research effort. Section 3.3 describes the methodology for development and verification of an adaptive framework for consistency, including the research plan and approach taken to accomplish this research. Finally, the chapter is summarized in Section 3.4.

3.1 Problem Statement

The current cache consistency scheme used on the Web, client polling, is bandwidth inefficient. Too many messages are due to caches guessing when their copy is inconsistent. If many requests that are sent to the server are for documents that are actually not modified, then bandwith could have been saved by simply serving the request from a cache in closer proximity to the requesting client. To address this problem, server invalidation with leases techniques have been proposed in the literature. The algorithms assume that it is possible to estimate the time that documents will change, but it is known that Web traffic and access patterns change frequently. Therefore, the leases should be dynamically adjusted based upon the frequency of writes as well as the request rate to capitalize on the benefits of a lease invalidation scheme. Objects that change frequently, but are seldom requested, do not need to be invalidated often. Also, objects that are requested often, but are seldom changed, may benefit from a lease technique. These conditions lead to a need to determine the threshhold of when client polling, invalidation, or both should be employed. Due to the server overhead and, in some cases, unnecessary traffic generated by object invalidation, there is minimal acceptance of strict server invalidation with lease techniques. However, it is shown that server invalidation with leases provides some benefit over client polling in conditions where the server makes many writes to popular objects and the tolerance for stale data is low.

The overall intent of our research is to define criteria for adapting Web consistency protocols to request and modification frequency and to specific conditions of the Internet. This includes understanding how popular documents in the Web change, determining their request and write history, and identifying the network problems that can occur when employing existing mechanisms of coherency. A set of effective consistency policies is then identified from the investigation. We make recommendations based on how frequently documents change and are accessed. The main objective of our work is to use our findings to design and implement a consistency algorithm that provides improved performance over the current mechanisms presented in the literature. We develop an algorithm that adapts based on requests and updates to documents. This is a server-dominated approach that changes to a proxy-dominated approach, introducing the notion of Synchronous Internet Nodes for Consistency (SINC). The approach to this research is twofold: to evaluate Web cache consistency in its current state and, then, to develop an improved approach to Web cache consistency based on the findings about the existing access pattern and network effect of the existing infrastructure.

3.2 Research Objectives

This research provides possible alternatives to the cache consistency problem introduced in the Web when caching is employed and users are concerned about the response rate and bandwidth issues that arise due to the current approaches. By reducing the stress on the existing network, decreasing document latency, and creating a more efficient and reliable Internet environment for receiving Web documents, Web caches with an efficient cache consistency framework can significantly improve while minimizing network and server costs. The objectives of this research are as follows.

1. Identify which performance measures are most useful for evaluating consistency of cached documents and the efficiency of maintaining consistency.

2. Quantitatively assess the cost of maintaining consistency with HTTP/1.x using measures identified in Objective 1.

3. Evaluate the feasibility of server-based invalidation as compared to client-based consistency.

4. Develop a new algorithm for consistency that results in low staleness and minimal cost to the proxy, network, and server. This includes making the necessary modifications to HTTP/1.1, incorporating a staleness calculation for all documents cached at the proxy, and using an estimation technique for lease calculations at the server.

5. Establish the correctness of the new algorithm.

6. Demonstrate the performance of the algorithm on benchmarks using measures from Objective 1.

7. Quantitatively compare the new algorithm to existing algorithms using the metrics defined in Objective 1.

Overall, this research provides alternatives to improve network bandwidth, reduce latency, and decrease server load while maintaining a consistent view of documents on the Web.

We examine each of the objectives in further detail below.

3.2.1 Objective 1: Performance Metrics

The measures mostly used in previous studies are average number of polls [23] [63] [49] [26], server overhead [23] [26], and response time [26]. The experimental measures used in our preliminary studies and the performance evaluation of the new algorithm include message count, message byte count, control message count, file transfer count, unnecessary update count, stale rate, period of staleness, and server state overhead. These measures were determined based on the necessary characteristics needed to identify the advantages and disadvantages of consistency methods. Table 3.1 summarizes the metrics considered in this study.

3.2.2 Objective 2: Quantitative Assessment of HTTP1.x

We assessed the quantitative limits by which HTTP/1.1 provides a solution to the consistency problem. We use the first four metrics defined in Objective 1 to determine how the Web in its current state would work using general methods of Server Invalidation, Time-to-live, or Client Polling. To optimize the performance of caches, we must determine as much information about the Web with the HTTP/1.1 mechanism. Various scenarios were studied to provide an acceptable view of the Web as it is, as well as to prepare for scaling in the future. The following questions are answered based on the experiments done to evaluate the current conditions of the Web.

- Is it best to provide consistency at the proxy level or to allow the server to handle consistency depending upon the request and response patterns?
- How many messages are generated as a result of using Server Invalidation, Time-to-live, or Client Polling?
- How many *get_if_modified* requests were sent and received? Of these requests, how many resulted in the return of a modified document?
- How many requests result in a stale document being delivered to the client?

Table 3.1: Experimental Metrics

Measure	*Definition*
Messages	Total number of messages exchanged to gain consistency in a given period of time. This is a count of all messages involved in a complete transaction between the proxy and server including control messages and file transfers.
Bytes	Total number of bytes exchanged to gain consistency in a given period of time. This reveals whether file size affects how much traffic is introduced due to consistency.
Control Messages	Number of control messages sent due to consistency. Control messages are the status messages necessary to notify the proxy of any changes. They include the *Get-If-Modified-Since* request sent by the proxy and the *Not-Modified* response sent by the server.
File Transfers	Number of file transfers sent to maintain consistency. File transfers produce the largest amount of traffic, especially for large files. This metric only accounts for the number of messages that contain the actual data.
Unnecessary Updates	Number of unnecessary messages sent to maintain consistency. This is a count of the requests that result in a *Not-Modified* response.
Staleness	Percentage of document requests made by the end user that result in a stale copy being returned to the user.
Period of Staleness	Average time when the proxy objects are not consistent with the server. This metric occurs when we retrieve date information in proxy logs and determine average duration of stale documents.
Server State Overhead	Amount of table space (number of proxies and documents) required to determine when to notify end users that an object has changed.

In the end, we show that the benefits of using server-based invalidation consistency schemes in some form do outweigh the cost of consistency traffic that results from the traditional client-based protocols used today. Chapter 4 gives further detail on this analysis.

3.2.3 Objective 3: Feasibility of Server-based Invalidation

We strengthen the choices for our algorithm by analyzing the results from traditional approaches using the existing infrastructure of the Web. The HTTP/1.1 protocol does not include invalidation messages, so there was a need to determine what protocol changes would be necessary to implement such a scheme. If these changes required so many modification to the HTTP/1.1 protocol that the alternative would be unacceptable to the general user, then another tradeoff is implementation considerations. In addition, the overhead of maintaining client lists at the server may be a greater burden than desired. It was necessary to justify the

need for invalidation before incurring such costs. The purpose of this study is to determine whether the Server-based Lease Invalidation protocol is an effective choice to solve the cache consistency problem. We determine this by performing two experiments. First, we measure the fraction of documents that are modified often and determine whether the changes occur in a predictable manner. In other words, if a large number of documents change in a short period of time, do they change every x hours. If x is large, but the rate of read accesses is small, then HTTP/1.1 today generates lots of unneeded reads. We also determine how many sites need to be notified of a change in a server document. In addition, we determine how many times proxies contact the server. This reveals how much overhead is involved in using the invalidation method to disseminate changes at the server. We want to provide consistent documents with high probability while minimizing the number of consistency checks necessary to accomplish this task.

The following questions are answered based on the execution of the stated objective.

- What percentage of documents change? Does document type affect how much and/or how often changes occur?
- Of the documents that change, how many clients and proxies need to be contacted to invalidate and update copies?
- What are the tradeoffs of using *strong consistency* versus *weak consistency*?
- When should we send updates? Should we invalidate or update immediately when a change occurs at the server?
- Will invalidation work when a large number of proxies hold copies of a server document?

3.2.4 Objective 4: A New Approach to Consistency in the Web

Along with observing current algorithms, we design a new method and study its capability. This algorithm takes into consideration the measurements from the feasibility study stated here. The overall intent of this research is to achieve increased consistency with reduced overhead for the server and the network. To accomplish this, we seek to achieve the following goals when developing the algorithm.

1. Unify weak and strong consistency into a single, generalized algorithm.

   Server invalidation is used as the starting point for the algorithm. We must limit the amount of time that state information is maintained at the server and the amount of storage required. Client access history is used to control how much state information is maintained at the server. Server write history is used to control how many messages are sent from the proxy due to polling of the server. More specifically, we focused on the following techniques.

(a) Proxy Side Configuration where the proxy cache is split into two parts: proxy-controlled entries using the Client Polling with Adaptive Time to Live (ATTL) and server controlled entries using Client Polling with Server Invalidation. We call this a Split Cache Configuration. There are two versions of client polling at the proxy.

i. **Client Polling with Adaptive Time to Live (ATTL).** This is the algorithm used on a portion of the cache. The main idea is to poll the origin server whenever the TTLs expire. TTL is an estimation of the duration in which the file is less likely to change. We use an adaptive TTL algorithm where the client proxies initially calculate TTL values based on the polling history within proxy log files. However, it takes into account a proxy's received requests to reduce the polling overhead. Most requests that could be written frequently, but are not accessed frequently, should utilize this algorithm.

ii. **Client Polling with Server Invalidation.** This is the algorithm used by the other portion of the cache. This portion of the cache holds documents that are required to be invalidated based on the write history of the documents within the log file. It takes into account a proxy's received response from the server to reduce the polling overhead. Most requests that are accessed frquently, but not written frequently, should utilize this algorithm.

(b) Server Side Configuration where two lease invalidation policies are employed: Document Level Server Invalidation with Leases where the server maintains state on a document basis and Proxy Level Server Invalidation with Leases where the server maintains state on a proxy basis.

i. **Document Level Server Invalidation with Leases.** In this scheme, a file is modified at its origin server. Then we construct an invalidation message and broadcast it to the proxies within a list of proxies that hold a requested document. Proxies that hold the file check their local copies upon receiving the invalidation message and change the state of the local copies to stale if they are older than the master copy. Upon receiving a subsequent request for a stale document, a request is propagated directly to the server.

ii. **Proxy Level Server Invalidation with Leases.** In this scheme, the server simulaneously employs both Proxy Level Server Lease Invalidation and Adaptive Polling algorithms to provide consistency. The Polling part of the algorithm is exactly the same as the Invalidation algorithm. However, the Polling part of this algorithm is slightly modified from the normal Adaptive Poll algorithm such that it takes into account the received invalidation messages for a subset of the documents at the proxy to reduce polling overhead at the proxy.

2. Allow server to control both the local state and the network bandwidth.

Data is maintained at the proxy and the server. At the server, we must control how much space is used by each proxy and how much time the server will maintain that information. Communication between the proxy and server is needed to accomplish this. The server must tell the proxy how much space it is allowed and for how much time, and the proxy must tell the server what documents to keep based on the limits given. The necessary variables, data structures, and messages needed to accomplish this are described in Chapter 5.

3. Ensure that existing weak consistency algorithms should not incur additional performance penalties.

   We have only introduced new header fields that are ignored by the new server if the proxy uses the existing weak consistency algorithm. In addition, an old server ignores any new fields used by a new proxy. Therefore, we have not introduced any additional performance penalty for the existing weak consistency algorithms used in the Web today.

4. Ensure that the old proxies work with the new servers and the new proxies work with the old servers. It should be possible to incrementally deploy the new framework, i.e., it should be backward compatible.

   To ensure that the existing proxies work with the new scheme, we have messages (implied or explicit) that tell the server not to maintain information for that proxy. The new server maintains information or refrains from maintaining state information based on messages that the proxy sends. Our implementation is based on HTTP/1.1, which allows for user-defined extensions as part of the protocol's request and response header. Header extensions are used to allow proxies to initially request and renew leases from a server. Lease request and response settings are included in the normal HTTP request and response messages. The lease renewal and invalidation requests are sent as HTTP request header extensions. The message exchange given in Chapter 5 depicts what happens for the various combinations of new and old proxies and servers.

3.2.5 Objective 5: Measuring the Performance of the New Technique

We discuss the performance of the invalidation with leases algorithm reported in the literature. We use the existing Web infrastructure with our framework, combining the proxy and server algorithms. We define a set of criteria, the sub-goals given above for Objective 4, for evaluating the performance of the new algorithm and use a technique to determine how the algorithm performs under defined conditions. All analysis is performed using emulation with a workload of server log files.

To validate our new technique, we have to start with measurements using log files to verify our data. This is done through trace file instrumentation. This is necessary to explain what

savings our algorithm provides. We want to be able to say something like "thirty-three percent of *Not-Modified* messages are saved (not sent) by using our algorithm." We need know how many messages are sent prior to our algorithm usage to estimate how many are supressed by our algorithm. This is done by implementation of code to parse log files and extract relevant data to be used in the experiments. We answered the following questions about the log files.

1. How many requests were *Head/Post*, *Get*, or *Get-If-Modified-Since*? From log files, we deduce how many were *Get-If-Modified-Since* by observing the *Get*s and counting the second *Get*s for each IP address. We must be careful when analyzing the data from the server logs because some of the "200" return codes are *Get-If-Modified-Since* and some are simply *Get* requests. We do know that the *Not-Modified* responses are definitely in response to *Get-If-Modified-Since*. One way to determine this information from the log files is by rewriting the files as follows.

   - First access (same IP, same document, with "200" code) is a regular GET.
   - Second access (same IP, same document, with "200" code): Convert to "1001" (or some other special) code. These are the "Get-If-Modified-Since" messages.

2. What are the number of accesses per unique IP address? For example, we determine how many IP addresses resulted in 1 to 10 accesses, 10 to 100 accesses, 100 to 200 accesses, and so on.

3. What file types occur more often in the workload? Knowing the characteristics of the log files based on file type also gives some indication of whether the document will change or not. For example, a substantial amount of traffic can be saved if most of the objects are images, because images do not change often.

3.2.6 Objective 6: Establish Correctness of the New Algorithm

To prove that the algorithm is correct in general, we must have the following property hold true.

Theorem: If the document on the server changes, then within y time units, all cached copies equal the new document version.

In proving the above theorem, which is described in Chapter 5, we must verify that the following have occurred.

1. The system accurately sends and receives messages.

2. Calculations are computed correctly.

3. Representation of data is used accurately.

3.2.7 Objective 7: Comparison Study of the New Algorithms to Existing Algorithms

The objective of this performance evaluation is to support our framework and to reveal the inadequacies in directly applying some of the traditional algorithms to accomplish cache consistency in the Web. A set of existing consistency mechanisms and the new algorithm were studied to determine when they are most effective. This will aid in the identification of which algorithm performs best for specific Web environments.

3.3 Methodology

The methodology used in this study includes characterization of Internet traffic using the current workload, system requirement analysis, algorithm design, trace-driven emulation, system implementation, and assessment of the new system.

3.3.1 Research Plan

We now present a list of alternatives for accomplishing the seven objectives stated in Section 3.2. Information on choosing an implementation language, an approximation equation for determining read and write frequency, workload, and an evaluation technique are given.

Algorithm Implementation

1. **Existing Proxy and Server or New Implementation**

   The first choice that was required in the research was whether to use an existing proxy and server, and, if so, if it should be commercial or research-oriented. Although there are many servers and proxies implemented, it was determined that most of the ones investigated either had too much functionality or not enough. If the system was rich in features, it was difficult to "decipher" the source code to know where consistency modifications would occur. It was our desire to implement the cache consistency framework using the simplest approach to allow the basic proxies and servers to employ the technique. In addition, having many features means decreased performance when running experiments due to additional code executing that is unrelated to the cache consistency algorithms. Therefore, the final decision was to implement a simple proxy and server with only the features required to implement and evaluate cache consistency.

   Existing systems that were considered were SQUID [62] and Apache [4], both of which are large systems with many features. Squid is a full-featured Web proxy cache that runs on Unix systems and is free, open-source software. Squid supports proxying

and caching of HTTP, FTP, and other URLs, proxying for SSL, cache hierarchies, and many other features. The Apache HTTP Server is the result of a collaborative software development effort aimed at creating a featureful and freely-available source code implementation of a Web server. Apache is a full-featured web server with full support for the HTTP 1.1 standard, proxy caching, password authenticated web pages, and many other features.

2. **Language Selection Due to Performance**

   Initial implementation of the proxy and server with cache consistency was done in the Java programming language. The choice was based on the ease of implementation using threads for handling multiple connections to the server and proxy. However, after acquiring large log files and realizing the amount of time and resources that would be required to parse the logs and then to emulate proxies for each IP address located in the log files, it was necessary to change the programming language to the C programming language. It is well known that C still outperforms Java in most cases [6].

3. **Lease Determination Estimation Techniques.**

   Two estimation techniques were considered in approximating the leases required for documents: Lagrange interpolation and Exponential Weighted Moving Average (EWMA).

   Lagrange Interpolation. Lagrange interpolation is an estimation technique that fits an n-degree polynomial using $n + 1$ points. The polynomial is then used to extrapolate values for different future time periods. The time period that extrapolates to a value that is a significant change compared to the last available value is taken as the time interval to the next lease value. The time series here consists of time intervals between consecutive updates at a server.

   Exponential Weighted Moving Average. We consider the EWMA algorithm as is described by Sundaresan, *et al.* [63]. These authors use the EWMA in an adaptive polling technique to determine when a client should poll the server. The polling adjusts adaptively based on estimation techniques that try to predict the time of the next update to memory using historical information in grid resource monitors. We apply this same strategy to determine when a document will change at the server to assign a lease value to documents held by a server.

   EWMA operates on the principle of assigning exponentially lower weights to older values of the time series. There are different types of EWMA estimators. According to Sundaresan, *et al.* [63], the single EWMA estimator is effective in estimating traces which do not exhibit well defined trends or patterns. At any time instance t, the estimation of the time to next poll, $E(t)$, is given by

$$E(T) = \alpha(ts_t - ts_{t-1}) + (1 - \alpha)E(t - 1)$$

   where $(ts_t - ts_{t-1})$ represents the last observed time between two consecutive significant updates. The value of α determines how much weight is given to the

last observed value and how much to the previous estimates. The smaller the value of α, the more weight is given to the older values in the series. Larger values of α result in more weight being given to more recent observations.

Choosing the right value for α requires some amount of knowledge of the usage patterns on the series. This implies that the value of α should be chosen dynamically using the observed time series characteristics. Every time the server estimates the next time to update, it uses the current time series of update timestamps and simulates the estimation on that time series for different values of α. It then chooses that value of α that results in the least error. This mechanism ensures that the value of α adjusts to the usage pattern of the current iteration of lease assignments.

Single EWMA is a powerful mechanism for estimation on time series that do not exhibit any type of regular trend. Since it is reported that Web data does not exhibit a particular trend, the single EWMA is appropriate for our data analysis [63].

Workload

When determining the method to use to perform certain experiments, we must consider the workload data used. Davidson [20] identified the following three ways of collecting data.

- Synthetically generated traces: Generate set of logs with appropriate characteristics of real data usually following some predefined distribution.
- Captured client request logs: Capture logs containing trace data that represent likely request patterns and include exact information about object sizes and retrieval. This approach is replicable as the logs can be replayed over and over again.
- Actual live request: Collect traces at a "live" Internet connection. Live request streams are not replicable, since samples taken change over time, to the availability and access characteristics of proxies and servers.

According to Jain [35], a test workload can be real or synthetic. A real workload is a set of data gathered by observation on a system in current operation. It cannot be replicated and is usually not suitable for use as a test workload. A synthetic workload, which has similar characteristics to the real world, can be replicated and is generally used for studies on large-scale distributed systems [35]. Since we are attempting to compare a set of algorithms on the same set of data, we must use a synthetic workload to do this. However, on a minimal scale, we can use captured server logs. Servers normally log the requests that are received. Analysis of these logs is useful for understanding properties of the Web such as establishing object popularity (e.g., how often a document is accessed and the time of day), the origin of the requests (e.g., which clients requested the objects), and how information is disseminated

(e.g., pull caching, where the client determines what is cached, or push caching, where the server realizes which items are important and decides to duplicate it somewhere in the network) [19, 18]. However, to test for scalability, we must test extreme conditions that do not occur in the Web or logs, but that we expect to occur in the future. Generated synthetic workloads can accomplish this. We use captured log files in our evaluations of the algorithm.

Evaluation Technique

Following the taxonomy described by Davidson [20], we consider the following three evaluation techniques and apply each one to different aspects of the research objectives.

1. Simulated System/Network: This is the simplest mechanism for evaluation as it does not require full implementation. However, simulations require meticulous knowledge about the system being implemented. For simulation of the Web and Internet, we would be required to know how all the code works including the Web browser, server, proxy caches, HTTP/1.1 implementation, and the message traffic. This level of detail is not always possible when time constraints are involved. Furthermore, simulation cannot accurately assess the delay that occurs due to document requests sent between network connections because the delay is influenced by network conditions external to the simulation [70]. It is also impossible to accurately simulate caching mechanisms that fetch documents based on the contents of the page, since there are access links within Web pages (something that is not available from server logs). The complexity of the Web makes it next to impossible for a simulation to be used for accurate evaluation [54].

2. Real System/Isolated Network: Researchers often eliminate the variability of data from a live network, such as the Internet, by using local servers and isolated networks. This may generate unrealistic performance results by not considering current Web server and network conditions.

3. Real System/Real Network: The state of the Internet changes constantly. For example, Web servers and proxies may be functioning one moment and be inoperative the next. Therefore, live network conditions are usually not replicable. More importantly, connection to a real network requires compatibility with, and no abuse of existing systems (one cannot run experiments that require changes to standard httpd servers or experimental extensions to TCP/IP). For the work described here, it is necessary to modify the browser and server, which cannot be done on a real network.

Based on the three possibilities given above, the better environment for performing evaluations on a network is a real system within an isolated network. Once a working system has been developed, tested, and some performance analysis has been performed, then a stable solution can be safely incorporated into a real system within a real network situation. We

chose to use a real system with an isolated network. In addition to the above alternatives, Jain mentions that the best way to compare two or more systems for performance is with benchmarking [35]. Therefore, we chose a benchmark study to analyze the performance of our algorithm in comparison with the existing algorithms.

3.3.2 Research Approach

There were many decisions to be made for this research. Choices were based on the better approach to take based on resources available, trial-and-error, and suggestions in the literature. The consistency algorithm is based on the Server Invalidation with Leases technique and requires cache consistency to be handled by both the proxy and server. We chose to develop a proxy and server in the C programming language. The server software required research into how to best assign leases. It was decided that the EWMA would be an appropriate choice for predicting when updates would occur based on prior modification history of documents at the server. The end network environment, used for development and analysis of performance, involves a real system (actual proxy and server) with an isolated network (use of cluster nodes to emulate network traffic) using a synthetic workload of server log files. A benchmark study is done to evaluate and compare the algorithms.

In the remainder of this section, we describe how we accomplish each objective using the chosen workload and evaluation techniques described above in Section 3.3.1. We give details on why we chose to use a synthetic workload and a benchmark study to carry out the objectives.

Objective 1: Performance Metrics

The criteria listed in Section 3.2.1 for Objective 1 will be used in evaluating the performance of the new algorithm in comparison to existing algorithms.

Objective 2: Quantitative Assessment of HTTP/1.x

Using the six performance metrics given for Objective 1, an evaluation study is done to measure the cost of the TTL, Polling-Every-Read, Purge Invalidation, and Update Invalidation consistency mechanisms using HTTP/1.1. These parameters, along with others extracted from the literature, are used to produce a table of suggested consistency algorithms for the Web today. The type of workload that works best here would be server logs along with a list of popular URLs. The real network and a live connection is used to analyze the data. Analytical modeling is the initial tool used to determine the preliminary effect of various parameters and their interactions.

Objective 3: Feasibility of Server-based Invalidation

The feasibility study is an analytical approach to approximate which cache consistency algorithms work best given certain network parameters. We determine how frequently documents on the Web change, and reveal the number of messages required to communicate these changes to proxy caches. To determine the feasibility of server invalidation, we acquire an upper bound on the number of proxies that would be notified of changes in a server document if server invalidation techniques were used. This requires use of server access logs to determine how many proxies access a document. This involves extracting the addresses of proxies that access the server and measuring the characteristics of those clients from the time and frequency information provided in the log files.

Objective 4: A New Approach to Consistency in the Web

Developing the new algorithm requires a number of choices to be made. This includes determining what browser and server software to modify, which affects the programming language to be used. The implementation of a simple proxy and server is the basis of our development with the addition of the cache consistency source code necessary for the new server and proxy to accurately exchange cache consistency directives. Our implementation is based on HTTP/1.1, which allow for user-defined extensions as part of the protocol's request and response header. We make use of these extensions to specify the lease timestamps. Finally, the programming of the consistency algorithm in C occurred followed by testing of the source code using commercial browsers.

In addition, we are concerned with measuring the tradeoff between staleness and cost. How do we measure that a document that is returned is actually stale and whether there were significant changes? We make use of the modification rate and access rate to assign a score to documents that are cached at the proxy. This is similiar to what Google has done to assign ranking to documents to determine the results to return for a query from its search engine. In March 2005, Google was awarded a patent for a system of identifying and scoring documents in relation to historical data. The patent illustrates that Google will assign scores to websites and individual web pages by analyzing various historical data associated with the page and site, and pages and sites that link to it, since its creation. We will use a subset of the factors specified in Google's patent, as listed below.

1. Frequency of document change, e.g., how often a web page or site is updated.
2. The amount of change to a document over time. This applies to changes within individual pages (e.g., updating of content on a homepage).
3. Age of links and associated documents, i.e., the dates on which the links were first created, and the age of the documents on which they were created. Google specifies

that the system aims to penalize a document's ranking if the links and their associated documents are short-lived, and vice-versa.

Objective 5: Measuring the Performance of the New Technique

A benchmark study is performed to test the new algorithm under conditions that model today's Internet. This required that a testbed be defined on which the new algorithm could be applied. Then, the algorithm was observed to measure its performance. We used an artificial workload of requests that are replayed onto an isolated network.

The performance of Web browsers and servers depend on several parameters, such as the network protocol, the network, and the hardware. The hardware for the benchmark includes a cluster of 200 nodes. These nodes run Linux 6.1. The nodes were used to install Web server software and proxy software to model the Internet. The machines are connected over two network backbones. The high speed backbone is based on 2.56 GBs Myrinet connections and is used for Message Passing Interface (MPI) [51] communication. The base backbone is a 100-Mbps switched Ethernet network.

Objective 6: Establish Correctness of the New Algorithm

The benchmark and all relevant code needed to be measured for correctness. This was done in a way similar to what Yin, *et al.* [75] did with their simulation. They acquired the simulator of Gwertzman and Seltzer [32] and one of their traces. They compared the results of their simulator to those of Gwertzman and Seltzer. Secondly, they used simple synthetic workloads for which they could analytically compute the expected results. The workloads in this study are different from those of Yin, *et al.* and Gwertzman, *et al.* Corporation and university traces are used. Yin, *et al.* used 1995 traces from Boston University and Gwertzman, *et al.* use three university workloads to simulate the proxy reads. The writes are artificially generated based on data from the literature and the findings that result from Objectives 1 through 3. Mikhailov, *et al.* [49] also performed a study to measure the strong web cache consistency mechanism using snapshops of collected content rather than log files. The results of our benchmark study are compared to the results of Yin, *et al.* and Mikhailov, *et al.*

Objective 7: Comparison Study of the New Algorithm to Existing Algorithms

Based on the information given by Jain [35] and Davidson [20], the best way to compare the algorithms is to use a testbed consisting of an isolated server with multiple proxies connected to it. The server and proxy logs collected at various organizations are used to generate traffic. A comprehensive benchmark study is done using various consistency algorithms and measures. This allows a detailed analysis of these mechanisms in comparison

to the new algorithm. The benchmark study supports four algorithms: client polling, basic server invalidation, adaptive leases invalidation, and our hybrid algorithm.

3.4 Chapter Summary

This research investigates several cache consistency techniques, broadly divided into client-polling or server invalidation methods. In addition, we use a combination of client polling and server invalidation with leases, which our evaluations show improved consistency with minimal impact on the proxies, network, and server. To evaluate the effectiveness and overhead of each technique, we executed a detailed performance study. We performed experiments for one type of network environment, specifically a stable network of proxy nodes and a server node that remains in the network throughout the study.

3.4.1 Pre-analysis

We apply the various existing algorithms and evaluate their usefulness in solving the consistency problems evident from the growth of the Internet. We answer questions such as how useful is the caching scheme in the real world? Are current consistency algorithms sufficient for all caching needs? Answering such questions requires the collection and analysis of data and the interpretation of that data with respect to one or more criteria. Our measured criteria for this study are a subset of the metrics defined in Objective 1. They are messages, bytes, control messages, file transfers, and unnecessary updates. The evaluative study, based on a synthetic workload and performance analysis, allows us to choose the best components of caching to propose alternatives for specific real world applications. We employ the trace-driven technique to model performance of the various cache configurations and schemes. This analysis results in a summary of existing algorithms and scenarios for which they apply.

3.4.2 Algorithm Development and Analysis

Our approach supports both initial configuration, based on requests and responses recorded in an existing log file, and dynamic adaptation of that configuration in response to observed changes in real traffic as they happen. Synchronous Internet Nodes for Consistency (SINC) is structured as a set of basic policies responsible for handling HTTP/1.1 request and response messages and caching of documents. The communication aspect is encapsulated in the HTTP/1.1 protocol with changes to the messaging to handle the invalidation portion of the communication. Our measured criteria are messages, bytes, control messages, file transfers, unnecessary updates, staleness count, period of staleness, and server state overhead. We use a workload of traces from various organizations. Since we need to compare a wide variety of characteristics, the benchmark approach provides the best way to evaluate alternatives.

Chapter 4

Feasibility of Server Based Invalidation

4.1 Introduction

The popularity of the Web and its inherent problems have produced numerous research efforts for finding solutions to reduce bandwith requirements and to document the latency concerns of users and network administrators. The focus of most research to address the above stated problems is to observe Web performance by studying traffic under certain predefined conditions. This chapter presents preliminary work done to show the feasibility of employing server-based invalidation. The study uses an analytical approach to determine which cache consistency algorithms work best given certain network parameters.

The purpose of this study is to address the cache consistency issue. It provides an analytical approach to approximate which cache consistency algorithms work best given certain network parameters. A study was done to ascertain the rate at which documents on the Web change and to reveal how many messages are required to communicate these changes to proxies. This is accomplished by measuring when changes occur in documents at regular intervals and then estimating the number of messages that would be needed to propagate those changes. This allows us to enumerate the circumstances in which certain cache consistency mechanisms are most appropriate.

Two experiments are performed. First, we measure the fraction of documents that are modified often and try to determine whether the changes occur in a predictable manner. In other words, if a large number of documents change in a short period of time, do they change every x hours? If x is large, but the rate of read accesses is small, then today's version of HTTP generates many unnecessary requests to the server. A major part of this study is to determine how many proxies would be notified of changes in a server document if server invalidation were used. We also analyze the data to reveal how many proxies actually access

more than one document at a server or access a document multiple times. This reveals how much overhead is involved in using the invalidation method to disseminate changes at the server. We want to provide consistent documents while minimizing the number of consistency checks necessary to accomplish this task.

The workload characteristics used in the feasibility study are based on log file analysis. The logs for this study are dated, mainly due to the lack of access to current log files. Many companies, government organizations, and Internet service providers do not provide access to their log files due to security and privacy concerns. This results in a reliance on log files from universities who engage in Web and Internet traffic analysis. We are aware that the Web has changed since 1999 with the increase in e-commerce and the rise in broadband users. According to the Nielsen NetRatings 2005 report, the number of Internet users has grown to 200,933,147, 67.8% of the U.S. population. This is double the usage that was reported by UTI in 2000 of 95,354,000 Internet users or 33.1% of the U.S. population. However, we are only using these log files to evaluate existing algorithms for feasibility of implementation. While the workload for this study is not current, we use more recent files for the performance analysis in Chapter 6.

The overall intent of our research is to define criteria for restructuring Web consistency protocols to adapt to specific conditions of the Internet. This includes understanding how popular documents in the Web change, determining their request history, and identifying the problems with existing mechanisms of coherency. This is accomplished by studying, and understanding the pattern of access to documents in terms of frequency of requests and the number of proxy accesses for a document. A primary goal of this study is to determine the effectiveness of server invalidation and its role in providing consistency by defining when invalidation is most useful. Finally, a set of effective consistency policies is then identified from the investigation. We make recommendations based on how frequently documents change and are accessed.

We answer the following questions.

- Are the documents modified in a predictable manner (never, sometimes, often, always)?
- How does the access and modification rate affect the number of messages sent?
- How many web sites need to be notified of a change in a document if server invalidation is used?
- How often do proxies contact the server for the same document and for different documents?

4.2 Analytical Evaluation of Consistency Mechanisms

In this section, we describe the analytical evaluation of the performance of four basic consistency algorithms. We compare Polling-Every-Read, Time-To-Live, Purge Invalidation, and Update Invalidation. These algorithms were chosen to give an equal sampling of algorithms that provide strong consistency and algorithms that provide a measure of weak consistency.

The metrics used to measure the performance of these algorithms are the number of control messages, C, the number of file transfers, F, and the accumulated size of all file transfers in bytes, B. The file transfers are the messages that carry data.

Control messages are the messages that trigger forwarding of requests to the server and responses that do not carry data. These are generally **Get** or **Get-If-Modified-Since** requests, invalidations, or *304 Not Modified* response messages. Since the control messages are generally the same size, we use a single term, R, for all control messages in the derivations below.

The byte size is the measurement of the number of bytes transferred due to maintaining consistency. We represent the size of each control message with the term, L_C, and the size of each file transfer with the term, L_F. Although, the reduction in the number of messages being transferred is a good indication of bandwith savings, the size of those messages give a better view of the savings, especially for file transfers.

Table 4.1: Variables Representing the Number of Requests or Writes That Are Sent

Variable	*Definition*
C	Total control messages
F	Total file transfers
R	# of requests
R_t	# of requests after timeout (TTL, first occurrence only)
R_{tw}	# of requests after a timeout and write (TTL, first occurence only)
R_w	# of requests after a write (Invalidation)
W	# of writes
W_t	# of writes after timeout (TTL, first occurrence only)
L_C	Length of control message in bytes
L_F	Length of file transfers in bytes
T	Length of time of the traffic sampling
t	Timeout value

4.2.1 Formula Derivations

Next, we describe derivations of formulas used to estimate the number of control and file transfer messages sent for Polling-Every-Read, Time-To-Live, and the two variants of invalidation, Purge Invalidate and Update Invalidate. We use the variables defined in Table 4.1.

Polling-Every-Read

Polling-Every-Read [75] is activated whenever a user requests a document using an *If-Modified-Since* header in the **Get** request. The number of consistency control messages that result is $C = 2R$, where R is the number of requests. If no modifications occurred, $C = 2R$ would be the upper bound of the number of frivolous control messages that occur. Therefore, the number of extra requests that occur in a given time period is reduced by the number of file transfer messages resulting after a write has occurred. The number of file transfers is $F = R_w$. F is bounded above by $F = W$, if there are more writes than reads generated. $F = R$ if more reads than writes occur. The size of those transmissions are $B = 2R(L_C) + R_w(L_F - L_C)$. If no modifications occur, then $B = 2R(L_C)$ and $R_w = 0$.

Time-To-Live

The Time-To-Live [32] consistency mechanism is activated whenever a cache times out its copy of a document. It involves the cache recognizing that a cached copy is stale using the TTL value for the document. Once a copy is determined to be stale it is fetched from the server on a subsequent request for that document. The upper bound on the number of control messages, C, is the number of timeout intervals in a sampling of traffic. That result generally depends on whether there is a read in every interval. Therefore, $C = T/t$ where T is the length of time of the sampling of traffic and t is the timeout value. If the interval length, T, is less than the average time between reads, then the number of requests that generate a message to the server is R_t. This value represents the first requests that occur after a timeout. Therefore, $C = 2R_t$. The number of file transfers is $F = W_t$ where W_t is the number of writes after a timeout, but we only count the first occurrence of that write. Once expired documents have been refreshed after a timeout, then it is considered fresh, although multiple writes and reads could have occurred after the refresh. The size is $B = 2R_t(L_C + W_t(L_F - L_C))$.

Invalidation

We consider the purge and update invalidation techniques in this analysis. With the invalidation approach, the mechanism is activated when data changes on the server (due to a

write). It involves the server notifying the proxies that a cached item has become stale. If we let N denote the number of proxies to contact in the event of an invalidation and W denote the number of write modifications, then the number of invalidation messages is NW. For the purge invalidation method, the total number of control messages is $C = NW + R_w$ where R_w is the number of subsequent requests after an invalidation. The number of file transfers resulting from consistency is $F = R_w$, where R_w is the number of subsequent requests after an invalidation. The byte size is $B = NWL_F + R_wL_F$. For the update invalidation method, $F = R_w = 0$, which reduces the number of control messages to $C = N * W$. Although F is larger for the purge method, there is more bandwidth usage for the update method because the invalidations include data. The extra overhead of sending the file with the invalidation occurs because we are sending the file to all proxies that require invalidation whether a subsequent request comes from the proxy or not. In addition, we are sending updates to these proxies for every write whether they are requested or not. For update invalidation, byte size is $B = NWL_F + WL_F$. Note that purge invalidation approaches the performance of update invalidation as there are more reads that occur after each write, because the R_w term approaches W.

Table 4.2 gives the formulas for determining the number of control messages (C) and file transfer messages (F) for each pattern fragment of message traffic.

4.2.2 Summary of Analysis

Using the formula derivations above, we summarize how each of these variables affect consistency. To simplify comparisons, we let the number of proxies, $N = 1$. Therefore, N is not shown in the formulas for the invalidation methods in Table 4.2. Note that in Table 4.2, $R_t \leq R_w \leq R$ because R_t refers to the first occurrence only of a read after a timeout, R_w refers to all request after a write has occured, and R refers to all reads. In addition, $W_t \leq W$, because W_t refers to the first occurance of a write after a timeout and W refers to all writes. To get a tighter bound on the general equations for the size in bytes discussed above, the pattern of access and frequency of requests and modifications need to be known. Therefore, some of the terms in the general equations for size can be reduced or eliminated depending on how access and modifications occur.

In Table 4.2, the numbers of consistency control messages, file transfers, and byte transfers depend on whether the writes dominate or the reads dominate. When the reads dominate, the number of control messages is controlled by both the number of writes and the number of reads for all protocols except update invalidation. Update invalidation is controlled totally by the number of writes. When the writes dominate, then the number of control messages is based on the number of reads for all protocols except purge and update invalidation. Finally, for the TTL approach, additional overhead is based on whether the frequency of reads and the frequency of writes are less than the rate of timeouts. Figures 4.1 and 4.2 demonstrate how the rate of reads, writes, and timeouts control the number of messages, specifically how

Table 4.2: Number of Control Messages, File Transfers and Byte Sizes for Four Consistency Mechanisms Proposed in the Literature

Method	*Message Count By Read/Write Frequency*		
	$R_t \leq T/t$, $W_t \leq T/t$	$W_t \leq R_t$	$W_t > R_t$
TTL	$C = 2 * T/t * R_t$ $F = T/t * W_t$ $B = T/t(R_t)(2L_C + W_tL_F)$	$C = 2 * R_t$ $F = W_t$ $B = 2R_tL_C + W_t(L_F - L_C)$	$C = 2 * R_t$ $F = W_t$ $B = 2R_tL_C + W_t(L_f - L_C)$
Poll Every Read	NA	$C = 2 * R$ $F = R_w$ $B = 2RL_C + R_wL_f - L_C$	$C = 2 * R$ $F = R_w$ $B = RL_C + R_w(L_f - L_C)$
Purge Invalidate	NA	$C = W + R_w$ $F = R_w$ $B = WL_C + R_wL_F$	$C = W + R_w$ $F = R_w$ $B \approx WL_C + WL_F$ $R_w \rightarrow W$
Update Invalidate	NA	$C = 2 * W$ $F = 0$ $B = W(L_C + L_F)$	$C = 2 * W$ $F = 0$ $B = W(L_C + L_F)$

they affect the C and F values. Other graphs were generated using different parameters and yielded similiar results. Those graphs can be found in Doswell, *et al.* [24].

The most obvious trends in Figure 4.1 are that the the number of control messages for Polling-Every-Read starts to increase significantly as the reads increase, while the other methods start to level off. This is because the number of control messages for the Polling-Every-Read technique directly depends on the frequency of reads occur, while the number of control messages for the other scheme depends on a timeout value (for TTL) or the frequency of writes for Invalidation. The number of control messages for the TTL algorithm depends on the threshold value, T, and the timeout value, t, which are both constant. The threshold is the duration of statistics collection and the TTL value is chosen by the proxy administrators. We choose an arbitrary constant value of $T/t = 8$ in Figure 4.1. When the average time between reads or writes exceeds the timed interval, t, the counts start to depend on whether there is a $Write \ldots Read$ pattern that occurs immediately after that timeout. Figures 4.1 and 4.2 show that Polling-Every-Read is better until the reads become more frequent than the writes when $Reads/Writes \leq 1$. When reads dominate, Update Invalidation becomes better with respect to the number of control messages. Update Invalidation is shown to perform slightly better than Purge Invalidation because of the count of subsequent requests that result in a miss. The Update Invalidation algorithm returns data when it invalidates, so there are no subsequent misses. However, we must consider the fact that objects that are modified may never be accessed again. For that reason, sending bytes before they are

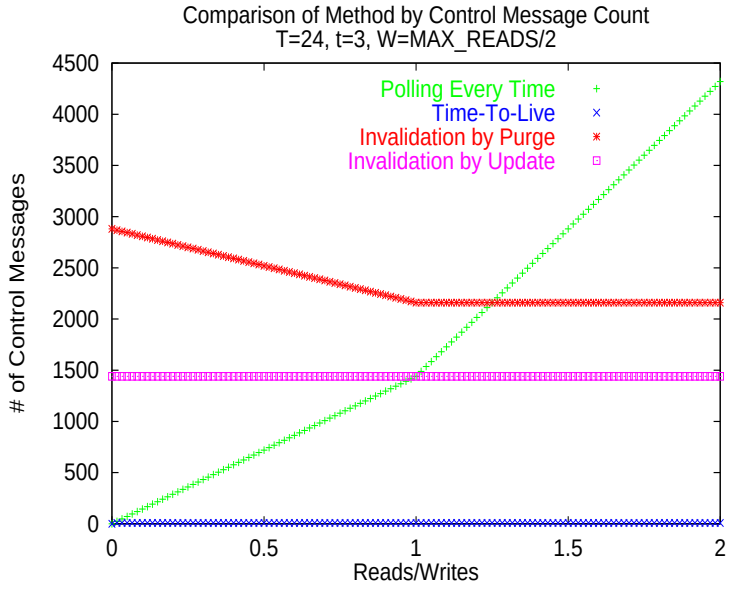

Figure 4.1: Number of Control Messages Based on Ratio of Reads to Writes

requested becomes a risk.

4.3 Experimental Design

The experiments performed to address the objectives are described next. We present the data collected for input, details of the experiment implementation, and the results.

4.3.1 Experiment 1: Percentage of URL Changes

About the Input Data

In this experiment, we analyzed lists of popular URLs from two Web sites. Each day for a period of 14 days, we collected popular URLs from two *Hot Spot* sites, *100HOT.COM* [1] and *PCmagazine Top* Listing [2]. We use both first level URLs (inital list of URLs) and second level URLs (traversed links within the first page). At each predetermined interval, the full content of each retrieved page is saved and compared for differences.

We selected five classes of documents with which to perform this experiment: popular business, family, news, search, and a combination of the top 100 sites from the PC magazine listing. The first four categories are extracted from the 100HOT.COM list. The 100hot page ranks the top Web sites based on the analysis of log files from cache servers at strategic

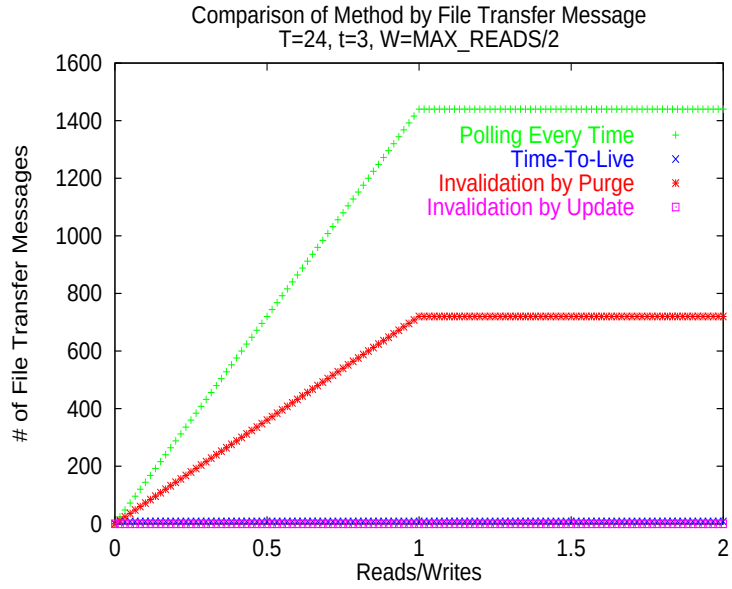

Figure 4.2: Number of File Transfers Based on ratio of Reads to Writes

points on the Internet backbone [1]. The 100 top site selections were made by a group of PC Magazine editors. A summary of the numbers of URLs per category is given in Table 4.3. In addition, the average percentage of URLs that report Last-Modified-Time (LMT) and Content Length (BYTE) are given in Table 4.3. In the table, Level 1 and Level 2 URLs are indicated by $L1$ and $L2$, respectively.

Table 4.3: URL List Characteristics

Category	*# URLs*		*% LMT Reported*		*% BYTE Reported*	
	L1	*L2*	*L1*	*L2*	*L1*	*L2*
Business	52	624	45	80	45	84
Family	68	400	58	58	58	62
News	68	610	46	72	37	57
Search	30	856	56	60	42	43
PCMag	100	774	30	54	34	50

Data Collection Methodology

We wrote URL collection sofware to download the popular URLs and separate the information retrieved at those URLs for processing by the statistical analysis software. The top 100 web sites are retrieved using the software. Some processing was necessary after downloading the file because it was initially in HTML format. The resulting file is the top URLs without

the HTML directives. Then, the software retrieves the content at each URL and saves the URL content in a file specified by the date and time it was retrieved. This file is used to calculate the differences between content across collection intervals. This is done for every URL in the popular files. In a separate file, the content pointer file, we record the URL, the filename of the URL content (pointer to the URL content), the *Last-Modified-Time*, and the *byte size*. The last two items are retrieved from the HTTP header during retrieval of the content. Finally, the software processes the content files to get embedded links within each Web page and repeats the entire process for those URLs.

The URL collection software is written in Java and supports the following configurations that determine what is retrieved.

- GetURLsFromWebPage. Parse the HTML pages downloaded from HOT.com and PC-Magazine. This generates a page of URLs for each site excluding any HTML code that was embedded in the downloaded pages.
- GetHTTPHeaderInfo. Get the *Last-Modified-Time* and *byte size* from the HTTP content of the URLs retrieved.
- GetURLContent. Get the Web page content from each URL and save in separate files. This generates a content file for each URL.
- GetEmbeddedURLs. Get the embedded URLs from the retrieved content of the first level URL HTML pages.
- PrintServerType. Take the list of URLs returned by GetURLs and determine the server type for each of the URLs. There is a count given for each type of server. This reveals the types of servers most frequently contacted.

Implementation Description

We measure the percentage of popular documents that change and how popularity affects the number of messages sent by three consistency mechanisms. We also compare the change in results calculated from using the UNIX DIFF utility with the difference using *Last-Modified-Time* and *Content-Length*. With these measures, we report the availability of these directives. This will give us a measure of whether users would specify TTL values and if content length can be an indicator of document change. We measure the first level URLs (initial page) as well as the second level URLs (traversed links within the first page). We acquire statistics that give us the percentage of documents changed over a specified time period. We account for this by receiving the content of a group of URLs over several intervals and comparing them using the UNIX DIFF command. We determine the percentage that changed over consecutive 3, 6, 12, and 24 hour periods. A Java program was written to obtain these percentages. We save the contents of two consecutive intervals and compare them using

DIFF. We also compare the last-modified-time and content-length returned in the header. Using a script, this process is repeated for several intervals over 14 days. Finally, the produced interval files were compared to determine what percentage of documents change within each interval.

Results

We repeat the analysis done by Wills and Mikhailov [69]. We collect additional information during data retrieval, specifically last-modified-time (LMT) and content-length (CL). These values are used to determine any correlation between them and the content differences. Table 4.3 reports the percentage of documents that specify LMT and CL. In general, the percent of pages reporting LMT and CL is 30 to 58 percent for the first-level URLs and 50 to 84 percent for the second-level URLs. This shows a large gap in how documents report the directives and suggest that there is inconsistency in their usage. Due to the large number of accesses that exclude these parameters and the large gap, we conclude that LMT and CL are insufficient indicators of document change. Our results are significantly different than those of Wills and Mikhailov, who reported that about 85 to 90 percent of LMT and CL values were specified [69].

The results of Experiment 1 are given in Figures 4.3 and 4.4. The figures depict trends that occur due to interval length (hours), URL type (business, family, search, news, or combined PCMAG), and page level (first level or second level). We found that documents change in a similiar pattern for the 2, 3, 6, 12, and 24 hour period. We concluded that the documents that changed in 2 hours were the same ones that changed in the subsequent intervals. With further experimentation, we determined that documents change half of the time when observed on an hourly basis than they did in the 2, 3, 6, 12, and 24 hour periods.

We also found that a large number of popular documents are modified over a short period of time. For first-level pages, PC Magazine's lists of the top 100 sites reported the largest percentage of changing documents (67% to 72%) with Search (58% to 75%) and News (54% to 62%) trailing closely in percentages. Business and Family oriented pages have a smaller percentage of documents that change in a small time interval. This means that these URLs, although popular, change less often than the other category of URLs. Special-purpose servers could benefit from knowing what consistency method to use based on the type of documents it serves.

The percentage of URL changes were less for the second-level pages for Business, News, and PCMag and remains the same for Family and Search sites. The decrease occurs due to the sites that change a small item, such as date or hit counters. Although these first-level pages change more often than second-level pages, there is little difference (10%). Overall, our results imply that when a front page of a popular site changes, other pages pointed to by embedded links within that site also change. According to the browsing nature of users, if a page is accessed, its links will most likely be accessed. Since we know that these links

also change as many times as the reference page, we should consider consistency schemes that allow additional invalidations to "piggyback" onto related invalidations [15, 39].

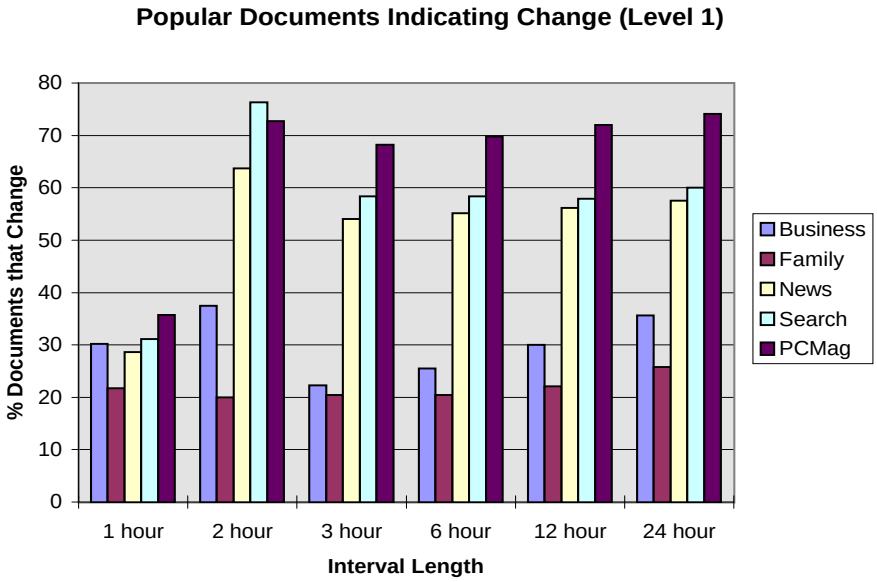

Figure 4.3: Document Differences (Including Deleted Documents) for the First Level Pages

4.3.2 Experiment 2: Proxy Cache Holdings Per URL

About the Input Data

We use three days of server log files from Virginia Tech's Computer Science server, *www.cs.vt.edu*. These logs, containing daily collections, include accesses from August 1, 1999 to August 3, 1999. We refer to these logs as VT-CS1, VT-CS2, and VT-CS3 to represent the day that the log covers. In addition to the VT-CS logs, we use two weeks of log files from several servers at James Cook University in Australia. These files, containing weekly collections, represent servers in the Engineering department (JCU-ENG) and the university library (JCU-LIB). These JCU-LIB logs cover the first two weeks in August 1999, therefore, JCU-LIB1 and JCU-LIB2 represent weeks 1 and 2, respectively.

Implementation Description

We study a group of server logs to reveal the percentage of proxies that access documents at a server. This tells us how many Web sites should be notified of changes in a document maintained by the server (for invalidation). We also measure the number of references made to each server document by each proxy. We calculate the number of proxies that access a

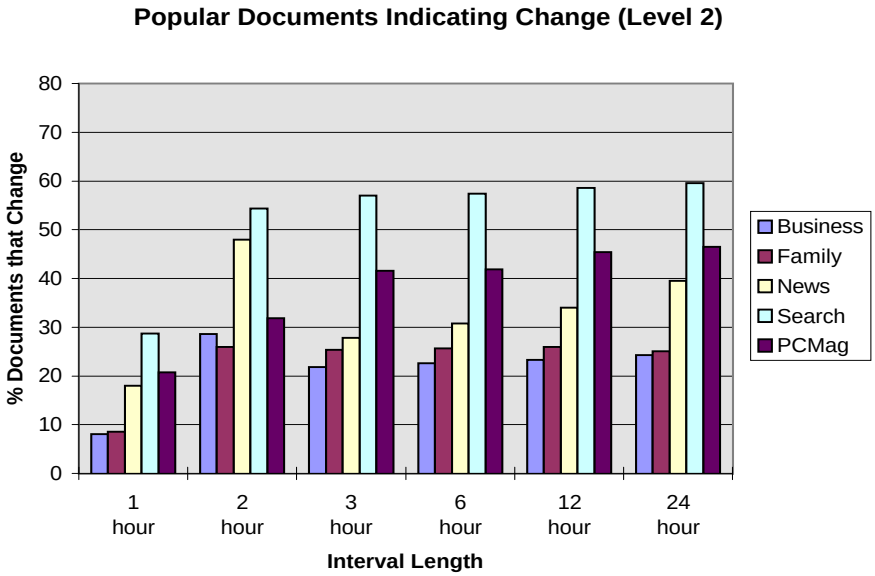

Figure 4.4: Document Differences (Including Deleted Documents) for the Second Level

certain set of server documents. This is done by attempting to access the IP addresses that occur in a server log entry, and determining if the host at the IP address is serving as a proxy. We examine the server log and get the IP addresses. Then we attempt to open a socket on port 80, the well-known HTTP port. If we cannot connect, we assume that we have not accessed an HTTP server, therefore the IP address represents a client. Otherwise, the IP address is assumed to be a proxy if it is running a Web server and sending requests.

Results

Table 4.4 gives the percentage of references to a server. It summarizes the percentage of unique references and accesses to the URLs by three client types: proxy, non-proxy, and other clients (clients that return "no-connection" or "unknown-host" file). The unique count is the fraction of distinct first time references to the server. The accesses, which include total duplicate references, give the fraction of total accesses to the server. The values in the table for acesses include multiple accesses to the same document by different clients and multiple accesses by a client to several different documents. In addition, the average percentage and standard deviation is given in the table. The data reveals that an average of only 11% of the unique accesses and 14% of the total accesses come from proxy clients. Therefore, a large percentage of the accesses to servers are done through non-proxy clients.

The experiment also involved calculating the number of proxies that access each URL in the server log file. Detailed statistics are given in [24]. About 92 percent of URLs were accessed by only one proxy. This shows that not many proxies have to be contacted. We

Table 4.4: Percent Unique and Total References by Proxy, Non-Proxy (Non-P), and Unknown Clients (X)

	% Unique 1st References By			*% Accesses By*		
	Proxy	*Non-P*	*X*	*Proxy*	*Non-P*	*X*
VT CS1	8.86	87.61	3.53	8.98	87.51	3.51
VT CS2	13.36	81.07	5.56	12.94	81.56	5.50
VT CS3	12.07	84.01	3.92	12.08	83.87	4.05
JCU Lib1	8.04	91.96	0.00	14.75	85.25	0.00
JCU Lib2	9.33	90.67	0.00	14.03	85.97	0.00
JCU Eng	15.55	84.45	0.00	23.04	76.96	0.00
Avg	11.20	86.63	2.17	14.30	83.52	2.18
Std Dev	0.03	0.04	0.03	0.05	0.04	0.03

also measured the number of proxies that accessed one URL, two URLs, three URLs, and so on. These values give an idea of the number of URLs, owned by this server, that a particular proxy is holding. The results show that between 20 and 30 percent of proxies access a single URL on the server. Therefore, 70 to 80 percent of the proxies that access the Virginia Tech and James Cook University servers, considered here, request multiple URLs from the server. There would clearly be an advantage to using schemes where several invalidations are sent in batches [15, 39] to proxies holding multiple documents owned by servers. Finally, we measured the number of duplicate URLs that proxies accessed to determine, for instance, how many proxies access a document only once or twice. This gives an upper bound on the number of URLs that were accessed multiple times by the same proxy. We found that a small percentage of proxies accessed a document more than once at the server (between 3% and 18%). This is influenced by the existing consistency mechanism used with HTTP/1.1.

4.4 Summary of Results

To summarize, the analytical analysis and experiments revealed the following results.

- Based on the ratio of reads to writes, invalidation performs better than the other strong consistency algorithm, Polling-Every-Read, when there are more reads than writes performed on a document.
- Popular documents that are more than one hour in age will most likely change giving an argument for using TTL or client polling.
- Although first-level pages change more often than second-level pages, there is little difference between the two. This implies that other pages at the site will most likely be

modified. This is an argument for "piggyback" server invalidation where the user sends updates of a group of related documents when invalidating a document.

- Servers administrators can benefit from the knowledge that if given a certain type of document, a given mechanism works best. News and Search Engine documents change more frequently than other categories, but they also require the most consistency. Invalidation would work well for these documents that are known to change often. Business and Family oriented pages would benefit from a weaker consistency mechanism like client polling or adaptive time-to-live.

- The majority of the documents at servers in our workload are only accessed by one proxy. In addition, these proxies accessed documents once in majority of the cases. Therefore, our results reveal that invalidation will not consume any more bandwidth than the other mechanisms when used for popular documents, especially since the number of proxies to contact is minimal.

4.5 Chapter Summary

Web caching reduces access time for the user. However, maintaining consistency of cached documents is a concern and may explain the reason for the low usage of proxies in the Internet. Existing algorithms allow stale documents, which may be unacceptable to users. There are many algorithms that have been proposed to decrease the possibility of stale documents without determining the impact on the Internet. We have investigated some characteristics of Web documents that affect how consistency should be maintained in the Web. Table 4.5 summarizes the conclusions. The results show that a high percentage of popular documents change over a small interval of time. Specifically, 50 to 72 percent of the most popular documents, News and Search sites, change within a two hour period. This rate of modification is less than the rate of access reported in the literature. Using this knowledge, we propose that strong consistency mechanisms can be used for popular documents. Client polling will bombard the server with unnecessary requests if no changes occur between one and two hours. TTL will require that the document creator have *a priori* knowledge of when a document changes, which is not always possible. Invalidation is shown to be the most useful of the three general consistency methods proposed in the literature, but how effective is it?

We observed that of the 11% to 20% of proxies that did access the servers, many did not access the servers more than once for the same documents (only about 15% make more than one access). Also, 92% of URLs were accessed by only one proxy. In addition, proxies do not generally access documents more than twice, but they do access many different documents at one server. Based on these statistics and the low number of proxies that need to be notified, server-based invalidation will minimally impact the bandwidth used due to consistency. Therefore, server invalidation is recommended for popular documents that

change frequently and infrequently. For documents that are accessed infrequently, client polling or TTL would work best.

Table 4.5: Selection of a Consistency Approach

Criterion	*Suggested Consistency Method*		*Explanation*
	Strong	*Weak*	
A priori knowledge of lifetime	Invalidation	TTL	This assumes that proxy has advance notice of exactly when a document changes. This is usually impossible even with documents that have a set time to change. However, it is best to use TTL unless strong consistency is required.
Popular documents that are modified frequently ($R \approx W$, R is large, W is large)	Invalidation	Client Polling	According to Douglis, et al. and our results, the rate of requests are higher than the rate of modifications. This implies that more reads are performed in comparison to writes. In addition, an average of one proxy needs to be contacted. Therefore, invalidations produce less messages than Polling Every Time. Client Polling decreases the need to poll the server at the expense of acquiring possibly stale documents.
Popular documents that are modified infrequently ($R > W$)	Invalidation	Client Polling	According to our analysis, Business and Family sites fall in this category. Invalidation would be better, since less modifications occur to documents than access to them.
Unpopular documents that are modified frequently ($R < W$)	Polling Every Time	Client Polling	If documents are not frequently requested, then there are fewer If-modified-since (IMS) requests and 304 response messages generated. Polling the server gives fewer messages while invalidations would produce unnecessary messages. Since it is highly probable that most writes will occur in succession with no intercepting reads, we do not need to poll the server as often as Polling Every Time.
Unpopular documents that are modified infrequently ($R \approx W$, R is small, W is small)	Polling Every Time	Client Polling	This scenario implies that documents are not requested very often and hardly ever change. These are usually old documents continuing to exist on the Web. The current Web mechanism, Client Polling, would work best.

Chapter 5

SINC: An Adaptable Web Cache Consistency Algorithm

5.1 SINC Overview

Synchronous Internet Nodes for Consistency (SINC) is a framework that involves both the proxy and server in providing a degree of strong and weak cache consistency driven by access and modification patterns of documents. We have designed and implemented a prototype system to deliver consistent documents to users. The system consists of three components: the Log Parser Request Generator (LPRG), the SINC Web Server (SWS), and the SINC Proxy Server (SPS). The components are illustrated in Figure 5.1.

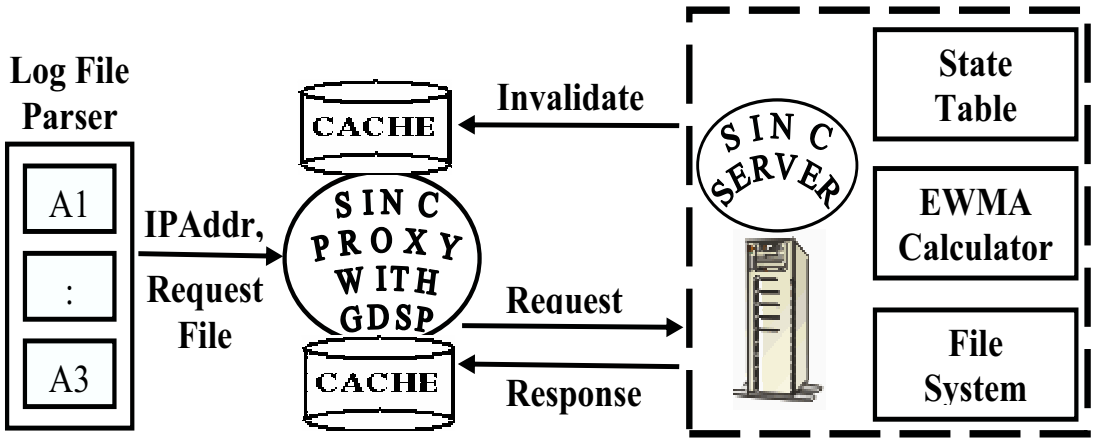

Figure 5.1: SINC System Components

All components are implemented in the C programming language or as shell scripts. We chose not to use a popular proxy server, such as Squid or Apache, because we did not want the progress of the analysis to be tempered by the extra features, not related to consistency, available with those large systems. It would require an enormous amount of computing resources to run the software, more time to learn the systems, and decrease the performance of the experiments to be run after implementation. Our choice to use the C programming language over the popular Java programming language also deals with performance of the experiments when executed with large log files.

One of the main features of the SINC framework is the splitting of the proxy cache into two sets of documents: invalidatable objects (with strong consistency) and non-invalidatable objects (with weak consistency). The other main feature is the algorithm used to mark whether an object changes frequently at the server or is accessed frequently at the proxy. In our architecture, we have a network of proxies that communicate with a central server. The proxies are not assumed to cooperate with each other because our interest is in limiting the traffic that occurs between a proxy and origin server. The origin server is used to store the write access frequency using prior history of writes and the Exponential Weighted Moving Average (EWMA) algorithm for computation. The proxy provides an environment where invalidation of a portion of its cache is allowed. Objects within the cache are monitored to determine if consistency should be maintained by invalidation or by client polling. The Greedy Dual Size Popularity algorithm (GDSP) developed by Jin and Bestavros [36] is used to determine the popularity of the Web pages at the proxy. We use the algorithm for a different purpose than the general use of document replacement. The algorithm is used as a placement algorithm rather than replacement algorithm where the documents are inserted in the cache in decreasing order of popularity. Therefore, the cache is maintained as an ordered list. We chose this approach over having an unordered cache list and having to swap between cache portions when the cache split thresholds change. We have no overhead of swapping between the two portions of the cache, just overhead for placing them in the cache. The future surge in requests is predicted on the basis of previous statistics. The GDSP algorithm helps provide a robust and efficient caching architecture, which provides a high hit ratio and lower staleness count than other placement or replacement algorithms mentioned in the literature [36]. The advantages of our framework are the following.

- The system tracks user accesses to determine the popularity of documents so that we can ascertain the ones that require invalidation from the server to the proxy to prevent the server from becoming a hot spot.
- The design decreases the server state overhead of maintaining state information for all documents and all proxies for some period of time.
- The system observes staleness time length and maintains object freshness above a certain threshold of time.

- The design limits bandwidth usage due to client polling by reducing the number of cached documents that require server access.

The GDSP part of the caching system addresses the following problems.

- How to capture the temporal locality in web requests.
- How to avoid cache pollution (the inclination of previously popular objects to linger in the cache).
- How to maintain the popularity profile of web objects with efficiency.
- How to accurately use the above profile to predict access frequency.

5.2 SINC Requirements

We discuss SINC requirements that we must consider during development in this section. We divide the section based on components in the SINC system.

5.2.1 Log Parser Request Generator

There was a need to automate the process of generating several proxies and having those proxies generate requests to the server. The Log Parser Request Generator (LPRG) generates multiple simultaneous HTTP requests to the server over a period of time and measures the metrics for request count, staleness, and server state. Log files are created by the proxies and server to obtain finer granularity of results.

5.2.2 HTTP Proxy Cache

The main characteristics required of the SINC Web cache are large disk space, caching software, and the characteristics of simplicity, efficiency, and stability.

- A large amount of possible disk space is a necessary consideration of any Web cache. Caches store web documents on disks for future access to speed up the response time to the user. Larger disk space means more documents can be stored, which results in higher probability of hits and lower response time to the client (user). Due to the low cost of disk drives, current configurations often have hundreds of gigabytes of disk space.

- HTTP caching software is required for the proxy to be able to send and receive documents. Caching software implements the insertion and deletion of items into a data structure and/or file system that represents the cache. It includes the replacement algorithm, consistency features, filters for including or excluding entities and other software intelligence required to send and receive HTTP messages and to store those messages for future access. For the SINC system, we require software that accepts and handles leases using the existing HTTP/1.1 request and response messages, manages a split cache, collects statistical data from the request and response data, processes the requests and responses, and handles cache boundary maintenance.
- Efficiency should be maintained by keeping network, proxy, and server overhead to a minimum.
- Stability should be maintained to ensure that any other network component is not affected by the instantiation of the caching system.
- The cache should be simple to implement or incorporate into the Internet in order to promote higher acceptance in the network community.

5.2.3 HTTP Web Server

The SINC server requires the following functions: handling the HTTP requests and response, maintenance of a state table, handling of cache consistency using a prediction algorithm, and handling of multiple clients and logging of access.

- The server must be HTTP complient. The basic role of the SINC Server is the processing of HTTP *GET* requests. After accepting the connection, the server must serve the basic HTTP methods (GET, HEAD). The processing of leases has to be handled by the server. This involves computation of leases, and injecting lease data into the request and response messages sent and received by the proxy. After serving the request, the HTTP server is ready to receive another connection. We are only concerned with static data in this instance.
- Server logging is important to provide consistent feedback about the access and modification of documents on the server. The Web logs of SINC are stored in the W3C's Extended Log File format for maximum compatibility with existing log analysis tools.
- The server needs to maintain state for each proxy that caches a document held by that server. The state information must refer to a proxy's identification, usually the IP address. The invalidation with leases technique requires that the documents that are invalidated also be maintained in the cache. Our framework requires that references to the modification frequency be maintained with each document.

- The server must be backward compatible with HTTP/1.0 and support our goal of ensuring that the existing Web remains operable after our framework is used in the network.

5.3 SINC Design

In this section, we describe the design issues for implementing the framework for our lease negotiation scheme. This includes proxy cache storage structure, policies for choosing a lease, renewing a lease, and sending invalidations and/or invalidation requests. We have implemented the lease negotiation scheme in a general proxy cache and web server.

5.3.1 Client-initiated Techniques

Implementing client-initiated techniques in SINC requires the following information to be stored with each document for this scheme.

- Version Number: the file version number is incremented upon each update
- Origin Server IP Address: where to poll for the status of the origin server copy
- Polling Frequency: number of times a server has been polled for the document
- Consistency Status: this is defined in five different variables
  - valid - when a file is consistent with the origin server copy
  - stale - when a file is older than the origin server copy
  - possible-stale - when the origin server of a file is unavailable
  - last update period - time of last update
  - staleness period - length of time the document is stale

Polling-every-time

A poll request is generated whenever there is an attempt to access a local copy of a document. The poll request includes the file name, the version number of the file, and it is sent to the origin server of the document. The stated document information specified with each file is sufficient for a correct implementation of this algorithm. The advantage of this technique is its simplicity and strong consistency guarantee. Its problem is that every request to the cache involves an extra round-trip delay and every response is also delayed because of the polling. Because of the control message overhead and the performance degradation, we do not find this technique appealing.

Client Poll with TTL

To implement polling with TTL, we maintain an additional piece of information for each file: the TTL value. The success of the client polling technique depends on the accurate estimation of the TTL value. There are various techniques proposed in the literature that compute the TTL value. They can be broadly divided into static and adaptive TTL algorithms.

- Static TTL is the simplest TTL algorithm. It chooses a low TTL value (based on some heuristics) and uses it throughout some specific period of time. It is easy to implement. The cache state overhead is low since computing the TTL does not require any previous history. However, it is difficult to choose a sufficient TTL value that gives a good consistency guarantee and has low control message overhead.
- Adaptive TTL improves on the problems of static TTL. Urgaonkar, *et al.* [65] have developed adaptive TTL algorithms that compute TTL values based on the history of file updates and other heuristics. Urgaonkar, *et al.* [65] proposed a linear increase multiplicative decrease (LIMD) algorithm to adapt the TTL value to the rate of change of the document. The LIMD algorithm gradually increases the TTL value as long as there are no updates at the server. On detection of updates, the TTL value is reduced by a multiplicative factor. It probes the server for the rate at which the object is changing and sets the TTL values accordingly. We chose this algorithm as it is the most recent implementation that yields good estimates of TTL values.

5.3.2 Server-initiated Techniques

We have described previously that the message protocol in the SINC architecture is HTTP/1.1. Our server lease cache consistency technique utilizes this protocol to propogate invalidations to the cache for a bounded number of documents, proxies, and time. Whenever a document at the server is updated, an invalidation message is constructed and sent to every proxy that has a lease in its state table. The message contains at least the following information.

- Proxy server ID (e.g., IP address)
- File name
- New version number of the file

A proxy receiving the message will check if it is cached. If yes, it compares the local version number of the file with that in the message, and invalidates its local copy if the message contains a newer version number. The advantage of this invalidation algorithm is its simplicity. It guarantees strong consistency given that all the proxies that cache the file are reachable

from the origin server. However, it can add unnecessary control message overhead into the network for files that will never be accessed again. Also, if a file is cached at many proxies, the overhead at the server for maintaining state information is considerable.

To implement this algorithm, the following items would need to be maintained at the server.

- Proxy server ID (e.g., IP address)
- File ID - must uniquely identify a file in the entire system (e.g., IP, Author Name, File Name tuple)
- New version number of the file - necessary to identify if a cached copy is out of date

Whenever a file gets updated at its origin server, the server sends an invalidation message in the same manner as it will send out a HTTP response message, but to all proxies holding copies of the file. Upon receiving this message, every contacted proxy will compare the File ID with those in its cache. If a match is found, the cached copy version number is compared with that in the message. The cached copy is marked stale if its local copy is older.

SINC: Server Invalidation with Adaptive Polling

The Server Invalidation with Adaptive Polling technique combines the Server Invalidation and Adaptive poll algorithms described earlier in this section. Using Server Invalidation alone gives strong consistency guarantees to proxy caches. However, it does so at the expense of high server overhead for maintaining state information on all proxies holding cached copies of the document and on all documents held at the server. Therefore, we add Adaptive Polling for a subset of the documents, the ones that have low access frequency. We now only have to provide strong cache consistency guarantees for a subset of the documents held at the server, which minimizes the state overhead at the server especially for less popular documents. In addition, it limits the bandwidth usage that is inherent in using strict Adaptive Polling.

The invalidation part of this algorithm is exactly the same as the invalidation-based algorithm described in Section 5.3.2. However, rather than maintaining server state information on all proxies that hold every document at the server, we limit the number of documents for which this state information is maintained, and we limit the length of time that the information is maintained. The novel part of this is that we allow the proxy to provide hints to the server about what documents are considered popular and require invalidation.

The adaptive poll portion of our algorithm is as described above in Section 5.3.1. However, we only poll the server on a subset of the cached items at the proxy. Since we are now combining invalidation and polling, ideally only those documents that have a low popularity factor are supposed to poll the servers.

- Now we want to take into account the number of requests a document has. The heuristic is that the more requests there are, then the better chance it will receive a future invalidation message. Let N_{req} denote the number of requests for a document and N_{avgreq} denote the average number of requests per document (the general cache statistics). We then need to change Eq. 5.1 to the following:

$$TTL_{estimate} = TTL_{latest} + (1 + N_{req} - N_{avgreq})/N_{avgreq} \tag{5.1}$$

  Based on Eq. 5.1, a file will be polled more actively at the server if the file has fewer accesses than the average number of document accesses at that proxy, and less actively if that document has more accesses than the average number of document accesses.

- Additionally, whenever a proxy receives an invalidation message for a file, it will mark the local copy of the document as *stale* if the message contains a newer version number for that file. At the same time, the proxy updates the file's next TTL value by adding to it a constant value, for which we can use the same constant as in the adaptive TTL algorithm. Although a proxy will not poll for a *stale* copy, when the copy is updated, the proxy will restart polling for the object using the TTL stored with the *stale* copy. By combining server invalidation and client poll, we provide strong consistency for the most popular documents and bounded consistency for the remainder of them.

We also must decide how to determine what is popular. We use the GDSP replacement algorithm for this purpose.

5.3.3 Replacement Algorithms

One decision that needed to be made was how to arrange the split caches and keep the invalidation portion filled with popular documents only. We decided to employ a caching technique that is already required in caching systems, a replacement algorithm. There are many cache replacement policies proposed in literature. A survey of these techniques is provided by Jin and Bestavros [36]. Two of the most basic policies that apply to Web caching include Least Recently Used (LRU) and Least Frequently Used (LFU). Least Recently Used (LRU) evicts the object which was requested the least recently. Pitkow and Recker [55] developed a proxy cache replacement algorithm that use access patterns to determine when to remove documents from the cache. Their algorithm is based on the Least Recently Used (LRU) algorithm [64], and reveal that there is a strong relationship between frequency and probability of access. LRU builds on the concept of temporal locality of reference, that recently accessed objects are likely to be accessed again. Recent studies have shown that this relationship is weakening for Web documents that are cached at proxies. One reason for this trend is the use of client caching. Documents get placed into the proxy cache because there was a miss on the cache in the past. Once the client has requested and received the

document, it will presumably cache the document at the client. It has no reason to contact the proxy for its most recent requests. Therefore, the recently accessed documents will most likely not be accessed in the near future because it is now cached at the client. Lease Frequently Used (LFU) evicts the documents which are accessed least frequently. LFU builds on the popularity of objects in a reference stream, that objects that were frequently accessed in the past are likely to be accessed again in the future. This implies that LFU depends upon the reference history. The popularity of Web objects is shown to be highly bursty, meaning objects that are popular over a short period of time are not necessarily popular for long periods of time. Therefore, we end up with objects that linger in a cache but are no longer accessed. The unique characteristics of Web access patterns observed at caching proxies (e.g., variable-size objects, variable-cost requests, burstiness of accesses, weakening temporal locality, etc.) limits the effectiveness of the basic cache replacement properties of LRU and LFU. Hybrid algorithms also are proposed in the literature where a utility function associates a cost with each object and evicts object with the lowest cost and size. The Greedy Dual (GD) algorithm, proposed by Young [77], is a popular hybrid algorithm that was proposed to deal with variable-cost (but uniform-size) page caching problem. The Greedy Dual Size algorithm, proposed by Cao and Irani [9], incorporates the size of documents as a part of the utility function. This enables a cache replacement strategy to be sensitive to the variability in Web object sizes and retrieval costs (miss penalty). We employ a variation of the GDS algorithm proposed by Jin and Bestvros [36], Greedy Dual Size Popularity (GDSP), that also considers document access as a part of its utility function. The algorithm capitalizes on various factors affecting web caching. It makes use of all the important factors such as temporal locality, size, and retrieval latency to generate a popularity profile. The popularity profile is then used to predict which objects should be placed in the invalidation portion of the cache and which object should be subject to polling.

5.3.4 Staleness Calculation

The cache consistency protocol is characterized using worst-case staleness and average staleness. We bound worst-case staleness by the difference in T and t, where T is the last update time that occurred and t is the amount of time that can elapse past the last update. We have defined a threshold on the amount of allowed staleness based on time. This insures that data returned by a read is never stale by more than t units of time. Average staleness is expressed in terms of the fraction of reads that return stale data and the average amount of time for which the returned data has been obsolete. For example, a Web site may want to guarantee that, in the worst case, it will not return input more than five minutes old, but make its best effort to actually return updates within a few seconds. It is best to provide this upper and lower bound on staleness. Invalidation with leases is an example of a consistency algorithm that guarantees worst-case staleness because it places an upper bound on how long a proxy can return cached documents without communicating with the server. An example of average-case staleness is the use of strict invalidation where the server sends invalidation

messages to clients when the data is modified.

5.4 Implementation

5.4.1 Log Parser Request Generator

To perform its measurements the LPRG uses a master process, written as a CSH script, that spawns numerous proxy processes. The proxy processes generate HTTP requests, while the server concentrates solely on creating these proxies and then tallying their results.

5.4.2 Proxy Cache Implementation

We have designed and implemented a caching proxy server that handles HTTP traffic. The proxy cache represents the actual proxy caches on the Internet. Our cache acts as the client, generates requests from a log file, populates the cache based on the log file statistics and settings of the various thresholds of the system, and communicates with the central server over sockets. The proxy includes the general features required by most proxy caches including parsing of HTTP requests, generating an appropriate response, logging of access data, and dealing with multiple requests.

The proxy maintains a ranking table that is partitioned into two parts: a server restricted part (SR-Cache), and a general cache (G-Cache). The SR-Cache maintains documents that are invalidated by the server for consistency. We call these server push entries. The server specifies how much space should be allotted to this part of the cache. If the server tells the cache that it can maintain 100 invalidation entries, then the Proxy reserves cache space for 100 documents to be invalidated by the cache. The G-Cache maintains all documents not controlled by the SR-Cache. We call these client pull entries. The G-Cache maintains consistency on this portion of the cache using the time-based mechnism currently used on the Web. There are d push entries (the same d entries maintained by the server) and an undetermined number of pull entries.

General Cache Features

Newly received documents are always inserted into the cache (they replace older versions if they exist). Since disk space is bounded, when there is no room to insert a new document, cache cleanup must be performed. In our application, the upper bound to the size of the cache directory is determined by the *max_size* parameter. The current size of the cache directory is indicated by the *current_size* parameter. Upon cleanup, documents are removed until the size of the cache directory is below the threshold parameter (a fraction of the *max_size*

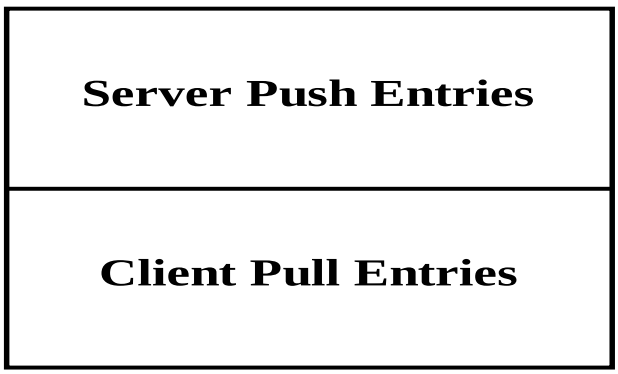

Figure 5.2: Proxy Cache Split Into Two Storage Spaces

parameter). The *max_size* and threshold parameters are customizable. The *Cache Manager* interface to these general features is presented below.

```
int parse_http_request(parsed request **request, const char *buffer,
                       int buflen);
```

Parses the HTTP request. *buffer* is a string containing an HTTP request. *buflen* should be specified as the total length of the buffer. *parse_http_request* will allocate memory for a parsed request structure and place a pointer to it in request. The calling procedure should check the error code returned by the function to help handle errors. If there is no error, it returns 0.

```
int cache_nelements();
```

Returns the number of elements in the cache.

```
int cache_insert(const char *url, const char *response, int url_len,
                 int response_len);
```

Inserts an item into the cache. *url* is the key, while *response* is the cached value of the key. *url_len* and *response_len* are the number of bytes to store from each buffer. Returns 0 upon success, CACHE ERROR upon failure.

```
int cache_update(const char *url, const char *response, int url_len,
                 int response_len);
```

Removes any element keyed on *url*, and inserts a new element with value *response*.

```
void cache_remove(const char *url);
```

Removes an item from the cache keyed on *url*.

```
const char *cache_find(const char *url, int *response_len)
```

Attempts to locate the value associated with *url*. This function returns a pointer to the buffer stored in the cache. If *response_len* is not NULL, this function will place the length of the returned buffer in *response_len*. Returns NULL if there is no element in the cache keyed on *url*.

```
void cache_evict(const char *url, int obj_size, int space_needed)
```

Implements the replacement policy for the cache. When the cache runs out of space, it calls *cache_evict* to free up space for a new object that is about to be inserted. *url* is the key of the object about to be inserted, *obj_size* is the size of the object, and *space_needed* is the number of bytes that needs to be reclaimed to store the object. The GDSP algorithm is used to handle this as well as the placement of items in the cache.

Log Generation

The proxy also keeps track of all requests in a log file. Each line should record the client IP address, URL, size of the returned object, access count, stale count, and time of last update.

Handling of Multiple Request

The proxies handle multiple request. Real proxies do not process one request at a time, sequentially. They process multiple requests in parallel. This is particularly important when handling a request can involve waiting.

Robust Cache Features

The proxy features a robust caching mechanism, that is optimized to minimize the time that end users wait for documents to load by giving preference to documents that have been referenced more frequently, and modified infrequently. In addition, the caching mechanism incorporates cache consistency control to decrease the possibility of returning stale data. This is achieved by incorporating estimations of the bandwidth between the local server and remote proxy servers, frequency of references to cached documents, and modification rate of documents.

The cache handles objects that correspond to files in the local cache directory and contain information about their creation date, size, and number of times they have been referred to. An object is created and inserted into the cache after the successful receipt and saving of the desired document. The *Cache Manager* implements an interface beyond the general features, which has the following functions.

```
int setURLData (String[] addURL);
```

This is a function called by the *Cache Manager* to add the URL to the ordered cache list. The cache also runs the GDSP algorithm whenever a URL request arrives.

The *Cache Manager* also maintains information about the request along with the cache status. The interface offered to handle this is as follows.

```
int getPage (String url);
```

The *Cache Manager* invokes this function. If the SR-Cache does not have a particular page requested by the client, it checks whether the page is available in the G-Caches by calling this function. The function either returns the cache ID which has that particular page, or NULL saying that the page was not found. In this case, the cache retrieves the page from the server by sending a *Get-If-Modified* message.

```
public int setMetaData (String[] addURL, String[] removeURL);
```

The SR-Cache invokes this function when some change in the data, which SR-Cache holds, occurs. This may happen in two cases.

1. The origin server decides to evict a few objects because of lack of space.
2. The *Cache Manager* decides to allow more or fewer documents to be a part of the Invalidation portion of the cache. This is done by changing the threshold pointer to the ordered cache to allow the SR-Cache to be larger.

Thus, the first argument tells the SR-Cache the URLs that are to be added to the SR-Cache and the second argument gives the array of URLs that are to be removed from the SR-Cache and moved to the G-Cache. The function returns NULL if the operation fails else the method returns a positive integer.

```
public int registerCache (String ipAddress);
```

The *Cache Manager* calls this function when it starts and registers with the server. The argument to this function is the IP Address of the cache. This is the call that requests a lease from the server.

```
public int setCacheStatus(int cacheID);
```

The *Cache Manager* calls this method after a predefined interval of time. It informs the *Cache Manager* about its status, i.e., the current capacity it has. When it is detected that a rise in popular documents has occurred in the G-Cache, the *Cache Manager* needs to change the setting for those popular documents so that they can be invalidated. The server is notified of the changes in the cache as well.

GDSP

The algorithm that chooses documents for removal is called the Greedy Dual Size Popularity Placement algorithm. Documents are placed in the caches to ensure that the most popular documents are using invalidation. We use the GDSP replacement algorithm to achieve this goal of inserting popular documents into the SR-Cache and less popular items in the G-Cache. Greedy Dual Size Popularity (GDSP) comes from the Greedy Dual Size (GDS) algorithm, which had the inability to capture long-term access frequencies of web documents. This occurs due to the prevalence of Zipf-like distribution in Web access characteristics. For example, the probability of future references to an object is dependent on past access frequencies. GDSP, a generalization of GDS, depends on knowledge of the popularity profile of Web objects. Access frequency is incorporated into the GDS algorithm through the use of a utility value for a given object. The calculation for the utility function of the GDSP Algorithm is

$$u(p) = f(p) * c(p)/s(p)$$

where s(p) is the size of p, c(p) is the retrieval cost (miss penalty), and f(p) is the access frequency. The $u(p)$ calculation represents the cost saved per byte of p as a result of all accesses to it in a given period of time. The pseudo-code for the GDSP Algorithm is

GDSP Algorithm

```
L = 0.0
H(p) = 0
For each request for object p do
       if p is in cache then
            H(p) = L + f(p) *  c(p) / s(p)
       else
             while there is not enough free space for (p)
```

```
                L = min{H(q) | q is in cache}
               Evict q which satisfies H(q)=L
        fetch p
H(p) = L + f(p)  *  c(p) / s(p)
```

Access frquencies are time varying. To account for this, we must adopt a mechanism to give preference to more recent references in the prediction. The weight of the first reference is set initially to $f_1 = 3$. This value was chosen because our traces reveal that the fraction of documents that were referenced two or more times was around 1/3. We discount the first reference because of the known Zipf-like nature of popularity distributions in Web requests. These accesses display Zipf-like characteristics because many objects in the Web are accessed only once, as was shown with our logs and the logs of Jin and Bestavros [36].

The overhead in GDSP is as follows

- The processing overhead (hit/replacement) is $O(\log n)$.
- Maintaining the popularity profile is low.
- Estimating the access frequency is low.

The maintainance of popularity information is as follows.

- Maintainence of document information for a subset of requests. The information includes the object size, the retrieval cost, the last access time, and the estimated access frequency. Very little of cache space is used to store this information. All except the last attribute can be found from the request message.
- The necessity of keeping information for cached objects. This is done to provide all objects fair treatment. It also allows us to avoid cache pollution, a phenomena that occurs when aged, unused, documents stay in the cache.
- Estimating long-term access frequency can increase the effectiveness of popularity profile maintenance.
- The popularity information can be efficiently managed by using either hash tables or priority queues.

Popularity Computation

The easiest way of computing the frequency of documents is the keep track of the entire reference history of each document requested. But this method is unrealistic due to the large scale of the web. Our solution maintains the access frequency and last access time of a fraction of the documents requested through the proxy. The space is bound in GDSP by satisfying two conditions.

- Less than 1% of the cache is used to keep the access frequency, and
- The total number of documents for which the access frequencies are kept is no more than 20% of the total number of objects expected in a given access stream.

Using the above conditions the total space requirements is only a few megabytes.

Management of Popularity Information

For efficient search of popularity information associated with a given URL, a hash table, using the URL as the key, is used. As explained, the information of only the popular objects is kept in the hash table. Then the replacement policy is employed to maintain ordering of the cached entries. For such a replacement policy, we maintain a priority queue with the document popularity calculation as the key.

Proxy Cache Statistics

The cache statistics is a monitor program, which generates all the cache results, which are useful for plotting the graphs. This program runs on all the proxy caches, which measure the hit ratio, average stale count, average stale period, and the average access frequency.

5.4.3 Server Implementation

The server keeps track of each document access in a table that contains the documents, a list of the proxies that access that document, and the Per-document Lease T_d. This allows the server to easily notify each proxy that accesses a document when the Per Document lease expires. We maintain an additional table that contains a list of proxy clients that access a document on the server, and a Per-Proxy Lease for each proxy T_p. When a proxy's timer expires, we go through the state table and discard the document lease information for that proxy.

Maintaining Spatial Control

We propose that the server limit the number of proxies it will keep information on. Lets call that number p. The server also tells each proxy how many documents it will maintain for it in its state space. Lets call that number d. Therefore, the server will keep at most $O(p * d)$ objects in the state space. We must bound p and d to minimize the storage requirement. Either of the values can be fixed or variable (adaptive). We propose that the server sets p to a fixed number and computes d based on an adaptive algorithm that dynamically determines

how much space to give to each proxy. So, our state space will look as is shown in Tables 5.1 and 5.2, where we assume some specific values for the purpose of illustration.

Table 5.1: Per Proxy Lease

Proxy List (Bounded by p)	*Lease*
P1	T_{p1}
P2	T_{p2}
⋮	⋮
PN	T_{pn}

Table 5.2: Per Document Lease

Document List	*Proxy List*	*Lease*
(Bounded by d)	*(Bounded by p)*	
D1	P1, P2, · PX	T_{d1}
D2	P1, P2, · PY	T_{d2}
⋮	⋮	
DN	P1, P2, · PZ	T_{dn}

If the state information is organized as in Tables 5.1 and 5.2, the server can efficiently manage clean up of the state information for proxies that are unreachable. There is no need to search through the list to handle the clean up. If a proxy should go down, we should no longer maintain state information for all documents for that proxy.

To determine how much space to give to each proxy, we must be fair to give more space to the more popular proxies, and less space to the less popular proxies. We maintain a counter of the access history and adjust the value based on the number of previous accesses by the proxy. We initialize these values based on the estimates using prior access log files on the server. This can be done by the algorithm inspecting previous log files and computing the number of accesses per client. We are aware that some of the clients in a log file are not proxies. We assume the lower access counts are for non-proxy clients.

Maintaining Temporal Control

The server must control how much time to maintain state information. This is what the currently proposed lease algorithms provide to maintain the duration of state information. To ensure that state information is not maintained for an indefinite amount of time, we use a Per-Proxy Lease (T_p) and a Per-Document Lease (T_d). Both of these terms can be fixed or variable (adaptive). The Per-Proxy Lease overrides the Per-Document Lease. The Per-Proxy Lease keeps a dead or unpopular proxy from holding space unnecessarily. We propose

that T_p be fixed. The Per-Document Lease T_d is the generic lease (proposed by others in the literature) that tells the proxy when the server will no longer contact it for a particular document. We use the Exponential Weighted Moving Average (EWMA) to dynamically compute T_d. The EWMA algorithm is exponentially weighted so that recent data is given more weight than older data. To compute this, our algorithm monitors the last change time, and the current change time. We then input the values into the EWMA equation to compute an average change time. The EWMA equation is the estimation of the time to next poll, $E(t)$, and is given by

$$E(T) = \alpha(ts_t - ts_{t-1}) + (1 - \alpha)E(t - 1)$$

where $(ts_t - ts_{t-1})$ represents the last observed time between two consecutive significant updates. The value of α determines how much weight is given to the last observed value and how much to the previous estimates. The smaller the value of α, the more the weight is given to the older values in the series. Larger values of α result in more weight being given to more recent observations. This average change time is the value that is used for the Per-Document Lease (T_d).

State Table Data Structures

The use of hash tables allows the user to quickly and efficiently store and retrieve data. Each hash table uses the IP address of the entry as a key. Each of the data structures uses a basic set of database-like operations, which allow the user to perform the following functions.

- insert_entry - insert an entry
- delete_entry - delete an entry
- find_entry - lookup an entry
- delete_all - delete all of the entries
- timout_entry - time out the entry

These take specific parameters based on what operations we want to perform. For example, if we want to delete all documents cached by a particular proxy, then we would call the *delete_all* function. This would be useful if we wanted to perform cleanup operations before shutting down the server. If a lease expires, we would call the *timeout_entry* to clear the entry.

5.4.4 The Message Exchange

The messages exchanged between the proxy and server are managed using the HTTP1.1 protocol. The message specifications and format are given below.

Using HTTP/1.1 to Handle Leases

The interaction between the proxies and the server occurs through HTTP/1.1 request and response messages. Our implementation is based on HTTP/1.1, which allows for user-defined extensions as part of the protocol's request and response header. Header extensions are used to allow proxies to initially request and renew leases from a server. Lease requests and responses are included in the normal HTTP requests and responses. The lease renewal and invalidation requests are sent as HTTP request header extensions. A new Lease Control header field was introduced in the entity header. This header allows lease information to be exchanged between the proxies and server. Shown below are the possibilities for the entity header extensions.

Table 5.3: Lease Entity Header Extension

Message Type	*Header Field*	*Header Value*
Request	Lease-Control	Grant | Renew | Replace Old-URL
Response	Lease-Control : Push	#-Lease-Entries | Deny | Lease-Period where Lease-Period = start-Date - end-Date
Invalidate	Lease-Control	Invalidate #-Lease-Entries
Invalidate-Ack	Lease-Control	Ack

The *Invalidate* message can be used to invalidate the proxy, meaning all documents at the proxy are no longer valid, or to invalidate a document only. The first case occurs when the server is performing state table maintenance due to space concerns.

An example of an HTTP request message with the lease control extension is given below.

```
GET If-Modified-Since http://csgrad.cs.vt.edu/~doswell/
If-Modified-Since: Sat, 29 Oct 2003 19:43:31 GMT
Content-Length: 1800
Content-Type: text/html
Lease-Control : Grant
```

An example of a normal response message with the lease control entity headers is given below

```
http://csgrad.cs.vt.edu/~doswell/
```

```
response
HTTP/1.1 200 OK
Date: Wed, 02 Oct 2004 19:05:45 GMT
Server: Apache/1.3.20 (Unix)
Last-Modified: Wed, 21 Jul 2004 21:09:51 GMT
ETag: "4e9cb-732-3d35dd1f"
Accept-Ranges: bytes
Content-Length: 1842
Content-Type: text/html
Version: XYZ10203040
Lease-Control: Push: Wed, 17 Jul 2004 21:09:51 GMT -
                     Fri, 18 Jul 2004 21:09:51 GMT
```

Message Sent to Request a Lease Entry: Figure 5.3 gives the fields that are present in a SINC server invalidation request and response exchange.

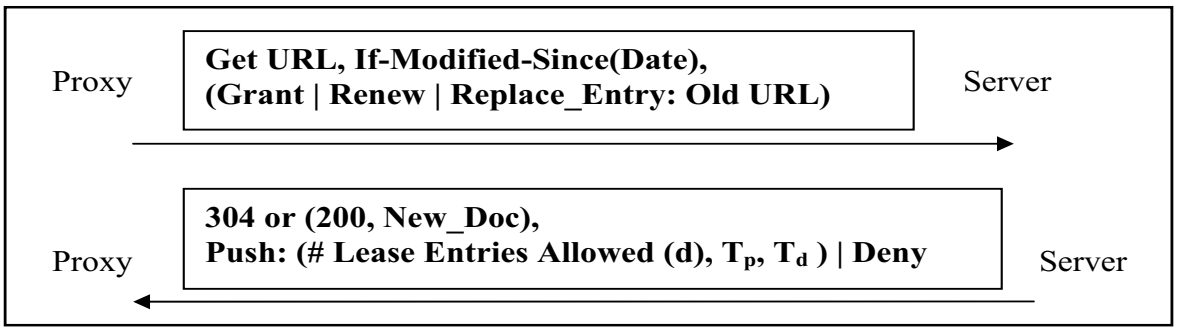

Figure 5.3: Request-Response Exchange

Message Sent to Invalidate or Return an Update: Figure 5.4 gives the fields that are present in a SINC server invalidation message to notify a proxy that its copy is no longer valid. An acknowledgement (ACK) is expected from the proxy in order for the write to proceed.

On start up, the proxy allows the client accesses to populate the G-Cache using client poll with TTL. The proxy uses the **GET** message or the **conditional GET** with **if-modified-since** message with the *Grant* field value to let the server know that it is open for using invalidation and to request a lease. In addition, we use those message types to move a document between the G-Cache and the SR-Cache. The *Replace_Entry* field is used to specify the URL that we are swapping out of the SR-Cache. We call this entry the Old_URL. The normal URL field is used to specify the URL that we are requesting to be moved into

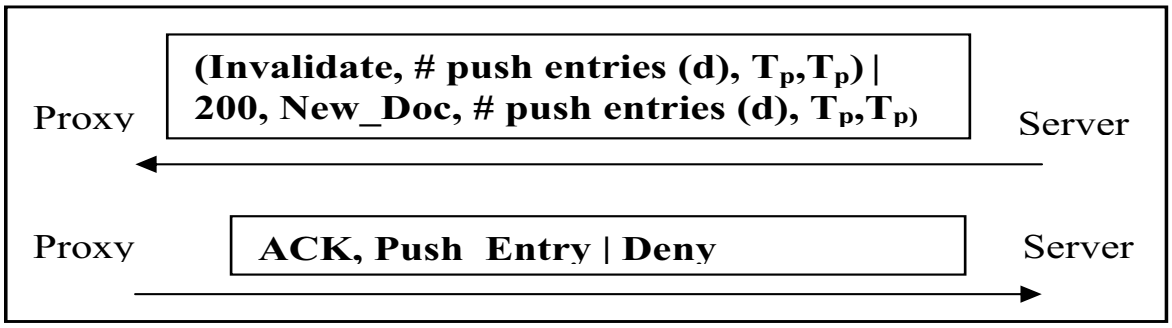

Figure 5.4: Invalidate-ACK Exchange

the SR-Cache because it has become popular based on a threshold. The server will grant or deny a lease for the URL and stop invalidating the *Old_URL*.

5.5 Validation and Verification

In verifying the operation of the consistency algorithms, we have to verify that the algorithms perform as expected. This process involves source code debugging, execution of various tests, and visual inspection of the results to ensure that the expected metrics were calculated. All aspects of this system implementation were verified in stages. Each stage either involved implementation of scripts to automate the process or use of existing tools to verify operations.

The proxy and server were actual implementations, not simulations, and required more visual inspection as well as manual execution than the other components. We have tested the solutions described above by several activities and measured the output of these activities. This was done for workloads captured in the CS, CNS, and NLANR modified logs.

5.5.1 Tools Used

Log Parser Request Generator

The Log Parser Request Generator (LPRG) was used as a request generator to the proxies and to spawn each proxy on various cluster nodes to eliminate development of client software that simulates requests. We implemented this as a driver of the emulation that was to occur using the cluster nodes.

Telnet and Commercial Browsers

Two approaches were used to test the proxy and server. We used *telnet* to simulate a web browser by telneting to the host and port where the proxy is running, and typing the request by hand (i.e., GET http://www.yahoo.com/). This type of testing was used initially as unit testing for each component of the proxy and server when we were not using an environment that had a commercial browser. However, final testing was done with a commercial browswer to test the communication between the proxy and server.

Web Server Access Logs

The Web server access log was used to determine the amount and size of data sent by the server to the clients, both browsers and proxies. It also was used to generate the emulation of proxy client requests.

Analog

Analog is a basic log analysis tool, available as freeware, that allows the user to pull various pieces of data from its log files such as number of requests, bytes transferred, hosts contacted, etc. This software was used to reveal some characteristics of the log files used in the performance analysis.

5.5.2 Testing Methodology

Testing was done in phases to provide verification that each unit functions properly. The tools mentioned above were used, to verify at the unit level. Integration of each component was done in pieces by adding one component at a time. The Log Parser Request Generator was verified, regarding that it spawned the correct number of proxies per cluster node, and that each proxy could receive its lists of requests. The proxy was then verified that it worked with the server by observing the log files generated at the proxy and at the server. Finally, the statistical analysis that was produced by the Proxy and Server software was verified by using the Analog tool on the generated log files to determine if the same results occurred. The items that were tested and compared when using the various consistency algorithms were as follows.

- number of requests handled per proxy
- average stale count
- average stale duration

- percentage of requests handled by client polling
- number of unnecessary polling requests
- percentage of requests handled by invalidation
- number of invalidation messages sent
- server state size

Results using these criteria are presented in Chapter 6.

5.6 Chapter Summary

The cache consistency negotiation scheme provides support for improving the amount of traffic that is generated unnecessarily due to the current method of achieving cache consistency, client polling. The need for a better consistency scheme is proven by the percentage of *304 Not Modified* responses that occur in log files. In addition, the new scheme addresses the issue of maintaining a large state space at the server when other invalidation with leases schemes are used. The novel approach does not just limit the amount of time that data is maintained at the server, but how much data is maintained at that server per proxy client. We aim to eliminate the maintenance of data that is no longer relevant, such as less popular documents that may never be requested again. Monitoring the message count reduction, staleness possibilities, and the server state reduction when comparing our algorithm to others proposed in the literature reveals the advantages of the approach.

This chapter describes a web caching architecture using GDSP as a predictive placement algorithm in the proxy cache and EMWA as a prediction algorithm in the server. In this model, multiple proxies generate requests to be sent to the server based on a multiple-instantiation consistency algorithm. We fuse two different consistency approaches into one framework. The proxies communicate with a single origin server. GDSP, which stands for Greedy Dual Size Popularity algorithm, is used to determine the popularity of the Web pages. It runs over every cache and inserts the documents in an ordered cache list based on the utility value. The future surge in requests is predicted on the basis of previous statistics. This algorithm also is used as the replacement algorithm when the cache becomes full and items need to be evicted. This chapter concludes by discussing the design and implementation details wherein the components are integrated and tested to obtain the web caching statistics.

Chapter 6

Performance Analysis

6.1 Overview

In Chapter 5, we presented the design and implementation of SINC. In this chapter, we evaluate the performance of SINC and compare it to the performance of three existing and proposed cache consistency policies using emulation. The three algorithms used for comparison are Client Polling, Client Polling with Adaptive TTL, and Server Invalidation.

6.1.1 Workload Characterization

Proxy

A proxy typically sees more user characteristics than origin Web servers, because it is located between a set of users and all Web servers. However, a proxy may inherently group together users with some overlapping interests (e.g., users of corporate proxy may be more likely to view the same content) [21]. We use one proxy trace in our experiments. The proxy log we used is from the NLANR cache hierarchy, which is a set of top level caches that are part of the IRCache project [31]. Currently, logs from the last seven days are available upon request. They are free of charge for academic use. We used 1 week of logs from the NLANR sd server cache starting on November 15, 2002, for a total of 3,950,198 requests. We refer to these logs as NLANR sd. However, not all clients identifed from proxy traces with unique IP addresses are only individual users. Since proxies can be configured into a hierarchy of proxy caches, we have to be concerned with the possibility that proxy traces could have clients which are really proxy caches themselves, with multiple users. Since this implies the occurrence of overlapping request patterns, we attempt to compensate for the redundancy. In the past, we have ranked the clients by total numbers of requests, and ignored the top twenty, and instead selected the second twenty as the representative set of active users. However, with

the knowledge that we have dated logs which have shown a 30% increase over the past 3 years, we reran our study without the previous assumption. The generated site visits did contain a significant number of duplicates, i.e., repeat accesses to the same URL in a single visit with return code "200". These accesses may be due to changed pages or due to servers that are not set up to support the **Get-If-Modified-Since** feature in HTTP. While some of the duplicates may belong to different end clients, we believe that most belong to the same client, as was shown in the feasibility study given in Chapter 4 using different workloads. In the following, we report results with duplicates included.

Server Logs

For the server logs, we analyzed logs from two sources. The first set of logs, VT-Server, records requests from the Virginia Tech Communications Network Services (CNS) which includes all Campus-wide traffic. We used logs for the period of August 1, 2003 to September 30, 2003, which corresponds to a total of 42,153,040. The second set of logs, CS-Server, records requests for the Virginia Tech Computer Science Department Web Server. We had logs available from October 2002 to September 2003. We chose to use 2 months out of the available logs that involved a time period that coincided with our first set of server logs. Therefore, we used the August 1, 2003 to September 30, 2003, a two month period, which corresponds to a total of 1,266,307 requests.

We have extracted individual request histories from the Virginia Tech CS server traces, CNS server traces, and NLANR traces.

Table 6.1: Log File Statistics

Category	*VT-Server 2 months*	*CS-Server 2 months*	*NLANR 1 week*
Successful requests:	42,153,040	1,266,307	34,759,977
Average successful requests per day:	691,033	3,647	4,965,737
Successful requests for pages:	6,377,834	353,058	11,983,363
Average successful requests for pages per day:	104,554	1,016	1,711,902
Failed requests:	287,881	81,246	874,302
Redirected requests:	1,327,360	7,961	54,189
Distinct files requested:	3,128	13,003	7,530
Distinct hosts served:	405,469	43,416	1,250,056
Corrupt logfile lines:	71,176	61,447	1,012
Data transferred:	214.31 GB	27.66 GB	476.66 GB
Average data transferred per day:	3.51 GB	81.58 MB	68.09 GB

6.1.2 Background Study: Assessment of Frivolous Requests

In Chapter 2, we discussed previous work on cache consistency and caching of frequently changing objects. However, certain aspects of the Web that are important for this dissertation were not addressed by previous work. Also, while some characteristics of the Web have already been studied, we wanted to validate our workloads to determine if the findings are still valid.

In this chapter, we present a background study investigating the amount of unnecessary traffic that is generated on the Web today. Caches issue many unnecessary requests to servers in an attempt to determine if they maintain a consistent copy of the document. Our background study determines whether significant network savings would occur if invalidation were available in the Web today.

6.1.3 Algorithm Performance

We have evaluated the efficiency of the SINC implementation by performing a benchmark study. We measured the control message, byte, server state, and staleness overhead incurred for grant, renew, and invalidation requests. Our results demonstrate that the overheads in grants, renewals, and invalidations are comparable with existing mechanisms.

6.2 Background Study: Assessment of Frivolous Requests

This section describes the special characteristics of server workloads that are beneficial in the performance analysis of the consistency algorithms. We get an upper bound possible reduction of the number of unnecessary requests sent to the proxy. We studied two months worth of web server logs from Virginia Tech Communication Network Services and the Computer Science Department collected from August 1 to September 30, 2003. These logs were generated in extended log format. For the purpose of our analysis, we determined the request stream that would most affect how consistency influences the amount of communication between the proxy and server. Therefore, we were concerned with the **GET** requests, and specifically the ones that generate *304 Not Modified* responses from the server. Performing some pre-analysis on the log files, we found that reducing the *304 Not Modified* messages would significantly reduce the amount of wasteful traffic that is transmitted in the attempt to maintain consistency.

6.2.1 Methodology

Frivoulous Request Assessment We need know how many messages are sent prior to our algorithm usage to estimate how many are surpressed by our algorithm. This is done by implementation of code to parse log files and extract relevant data to be used in the experiments. First, we needed to know how many requests resulted in **Get** and **Get-If-Modified-Since** requests. From log files, we deduce how many were **Get-If-Modified-Since** by observing the **Get** requests and counting the second **Get** requests for each IP address. We then observe the number of those requests that have a return code of *304 Not Modified* response or *200 OK* response. We assumed that each *304 Not Modified* response in the logs could be eliminated if a *200 OK* or another *304 Not Modified* response came from the same server. We must be careful when analyzing the data from the server logs because some of the "200" return codes are **Get-If-Modified-Since** and some are simply **Get** requests. We do know that the *304 Not Modified* responses are definitely in response to **Get-If-Modified-Since** request messages. One way to determine this information from the log files is by rewriting the files as follows.

- First access (same IP, same document, with "200" code) is a regular **Get** request messages
- Second access (same IP, same document, with "200" code): Convert to "1001" or some other special code. These are the **Get-If-Modified-Since** request messages.

Potential Request Per Proxy

Next, we needed to know the average number of accesses made by each proxy. This information can aid in estimating the amount of traffic that will come from each proxy in the experiments. We want to determine the percentage of proxies that access 1-100, 100-200, 200-500, and so on.

Types of Requests

Finally, we want to know the file types that occur most often in the Web pages. Knowing the characteristics of the log files based on file type gives some indication of whether the document will change or not. For example, a substantial amount of traffic can be saved if most of the objects are images, because images do not change often. It is also a good indicator of whether the objects are likely to be first level pages or links embedded in Web pages.

6.2.2 Results

Results for each of the log files are shown in Tables 6.2 and 6.3. The percentage of *Not Modified* responses is high. Table 6.2 shows that between 37% and 40% of all requests result in a *304 Not Modified* response, which is consistent with previously published results. Wills and Mikhailov [69] reported that only 15% to 32% of their proxy logs result in *304 Not Modified* responses. Yin, *et al.* [76] revealed that 20% of requests to the server are due to revalidation of cached documents that have not changed. These results are consistent with the results found in our logs as discussed in Chapter 4. However, with the current logs, the number of *304 Not Modified* responses has increased a little. This may be due to the duration of the analysis being longer for this study or to the use of different logs. It is assumed that a large fraction of these frivolous requests are due to embedded objects that do not change often, but HTTP/1.1 sends in response to a request for the container Web page.

Table 6.2: Logfile Analysis by Response Message Type

	VT-Server		CS-Server		NLANR	
Response Code	# Request	% Request	# Request	% Request	# Request	% Request
200 OK	24189816	55.28	705689	52.06	27125668	76.01
206 Partial content	274395	0.63	57982	4.28	472279	1.32
301 Document moved	53096	0.12	7961	0.59	440	0.00
302 Document found elsewhere	1274264	2.91	0	0.00	54189	0.15
304 Not modified since	17688829	40.41	502636	37.08	7161586	20.07
400 Bad request	1350	0.00	782	0.06	2097	0.01
401 Authenticate required	0	0.00	58782	4.34	17692	0.05
403 Access forbidden	2010	0.00	162	0.01	56	0.00
404 Document not found	284482	0.65	21062	1.55	666193	1.87
406 Document not acceptable	13	0.00	0	0.00	7	0.00
408 Req timeout	8	0.00	1	0.00	9569	0.03
412 Precondition failed	2	0.00	0	0.00	0	0.00
414 Filename too long	5	0.00	0	0.00	0	0.00
416 Request range not valid	8	0.00	5	0.00	997	0.00
500 Internal server error	3	0.00	452	0.03	176227	0.49

Potential Request Per Proxy

The LPRG is required to generate separate request files for each client located in the log files. It is useful to know the number and percentage of requests for which proxies actually access the server. From our analysis, most of the proxies accessed under 10,000 requests over a two month period. This averages to a daily request of 170 requests a day. This gives an estimate of how much traffic is generated to a server, and how much state information is needed if we allow all documents to be invalidated. It is advantageous to limit each proxy to a threshold of space for its documents.

Table 6.3: Number of Request and Bytes Held by Clients

	VT-Server		CS-Server	
# Requests	# Clients	Bytes	# Clients	Bytes
$R > 20000$	7	29.34%	25	87.78%
$10000 < R \leq 20000$	2	58.31%	63	1.57%
$5000 < R \leq 10000$	6	1.86%	442	5.83%
$1000 < R \leq 5000$	45	3.40%	471	4.82%
$500 < R \leq 1000$	51	1.21%	0	0.00%
$100 < R \leq 500$	592	4.81%	0	0.00%
$0 < R \leq 100$	298	1.07%	0	0.00%

Types of Requests

Our analysis of the log files indicates that 75% of requests for the CNS-Server and VT-CS server are images, which implies that they are embedded images that result when a page is requested. This provides significant value as to the type of traffic that is generated. The NLANR logs indicate 23.46% Web pages (HTML,HTM), 64.67% image pages (GIF, JPEG, PNG), and 11.87% other pages (PDF, PPT, TXT, and others).

6.2.3 Summary of Background Study Results

This section presented a study that investigated various aspects of the Web log characteristics that feed into this dissertation. We determined an upper bound on the savings that would be gained when we eliminate the extraneous requests that are sent to maintain consistency. We revealed that there is significant value in eliminating this traffic. We also revealed an upper bound on how much state information on average a proxy would require at the server. This is mainly useful in providing a threshold for initially setting the proxy states prior to obtaining the write patterns that subsequently drive the server portion of the consistency algorithm. Finally, we confirmed that most of the documents that result in *304 Not Modified*

are due to embedded items that are validated as a result of a front page being validated. If we could somehow reduce the checks for such items then the *304 Not Modified* messages can be further reduced by not validating embedded objects, which happen to be images that do not change often. With the findings, the next section describes the performance analysis of our framework in comparison to existing algorithms. It makes use of the findings reported in this background study.

6.3 Algorithm Performance Analysis

The purpose of this research is to evaluate and compare the performance of the cache consistency policies that have been proposed in the research literature. We have evaluated the performance of the implementation. First, we measured the overheads incurred during grant, renew, and invalidation requests. Next, we use a workload consisting of three sets of log files to measure the overhead of our algorithm in comparison to other proposed algorithms. Our results demonstrate that the overheads in grants, renewals, and invalidations are comparable with existing mechanisms. In addition, the state overhead at the server is reduced by using the write frequency to invalidate only a portion of the request. This section describes the experimental setup, our methodology, the cache consistency policies that we studied, the performance metrics used, and the results of our benchmark comparison study.

6.3.1 Experimental Setup

The testbed for our benchmarks consists of the Log Parser Request Generator (LPRG), the SINC Web Server (SWS), and the SINC Proxy Server (SPS) running on the Virginia Tech Anantham Cluster. The hardware for the benchmark includes a cluster of 200 nodes. These nodes run Linux 6.1 and have a 1GHz Athlon AMD processor, 1GB of main memory, and a 20GB disk. Data can be sent between any two processors in the cluster over a Myrinet network at rates of over 2 billion bits per second. The cluster nodes were used to develop and install the SINC Proxy and Server software and the LPRG to model the Internet.

Table 6.4 gives the configuration data needed to initialize the proxies and servers.

6.3.2 Methodology

In order to evaluate the performance of SINC and compare it to that of other cache consistency policies, we developed the Log Parser Request Generator (LPRG) to drive the emulation of the Web accesses. We assume an infinite capacity cache and no cache replacement policy for our experiments. However, the algorithm used for placement of items in the cache can be configured to replace items in the cache.

Table 6.4: Configuration

Parameter	*Description*	*Default*
N_{max_conn}	Maximum Connection Proxies Allowed	10
$Algorithm$	Cache Consistency Algorithm	TTL, INV, SINC
TTL	Fixed TTL of TTL Consistency Algorithm	5 minutes
$Placement$	Placement Algorithm for Consistency	[GDSP, LFU, LRU]
$NumberofCPs$	Number of Cluster Proxies	[5, 10, 15, 20, 40]
$Threshold$	Pruning Threshold (Percentage)	90
$Clean_{freq}$	Clean Frequence (minutes)	10
$TraceFile$	Location of trace file	/tmp/SINC
$TotalTraceTime$	Total trace time (hours)	24

Assumptions

Unless explicitly mentioned, all experiments have the following:

- Trace used: About 250,000 requests due to resource limitations.
- Number of proxies: 10 Proxies.
- Lease Duration: We employ a fixed lease duration of 30 minutes for all leases followed by a lease of 1 day.
- Renewal Policy: Leases are renewed upon arrival of a request and the determination that the lease has expired.
- Lease Termination Policy: If a proxy does not receive a request for an object for 30 minutes, it concludes it is not interested in the object.
- GET Message: If an object is not locally available or if an object has been invalidated, this message is issued.
- IMS Message: If an object exists in a proxy cache, but there exists no authority within the cluster to serve it, i.e., its lease has expired.
- Renew Messages: This serves as IMS messages also, but it is counted separately when this message is sent. This is done to address where an object changes at the server between the time the renew request is issued and the time the request reaches the server.
- Terminate Messages: These are issued from proxies to server when the replacement policy chooses to evict a cached document that is in the SR-Cache.

Log Parser Request Generator

LPRG is a shell script that spawns a predefined number of client processes to act as proxies. The LPRG simply parses the log files, separates the information into separate log files based on IP Address, and passes each resulting request file to a different process operating a proxy. Each proxy generates HTTP traffic to the Web server and collects performance data during a predefined period of time. After all proxies finish running, LPRG gathers the data collected by each proxy and generates an overall performance report that represents the behavior of the server during the execution of the workload, from the proxies viewpoint. In our experiments, we spread proxies over 20 nodes within the cluster.

Each request in the trace provides information such as the client making the request, time of the request, the requested URL, and the size of the object. Determining when objects are modified is usually crucial to cache consistency mechanisms. Since our framework is concerned with the frequency of requests and modifications, we ignore the time related issues of rate of request. Load generation is performed by successively requesting pages and files from the server as fast as it can answer the requests. A new request is sent out to the server immediately after a proxy receives the response from the previous request and all allowable connections are in use.

SINC Server

The SINC Server is responsible for collecting performance data at the server level. It is based on a trace of events that occur during the handling of HTTP requests. The server process accesses internal counters and timers that keep state information in relation to consistency and data related to the server's performance. The SINC Server measures the number of bytes transmitted and connections established during a period of time, read and write operations, and lease information. Our implementation supports the following functionality.

- Creation and maintainence of state information for all objects on the server. This state information is stored in memory to facilitate faster granting and renewal of requests and invalidations.
- Granting leases based on lease duration policies The following two adaptive lease duration policies are supported: (1) Frequency-based leases based on update history using the EWMA algorithm and (2) fixed leases. The policies are defined in a configuration file. The configuration also allows for specification of the threshold parameter associated with individual policies.
- Renewing leases based on lease duration policies.
- Invalidating all proxies of an document. Invalidations are sent in parallel to a maximum of ten concurrent invalidations to facilitate faster invalidations. The lease server

associates each invalidation with a timeout that can be configured. If an invalidation is not acknowledged by the timeout, the invalidation is retried a configurable number of times. A successful invalidation is acknowledged by the proxy sending an OK response to the server.

Server Message Exchange The protocol for SINC ensures that the server responds to grant and renew request from proxies. It also handles write requests by sending invalidations to the proxies.

- If the request is a grant request then the server assigns a new lease based on the lease computation. It sends the document, D, accompanied by a lease. It records the state information maintained for the proxy.
- If the request is a renew request with the IMS message, then the server assigns a new lease and updates its state. If D has been modified, it sends a 200 response with new D and a lease. Othewise, it just sends a 304 response accompanied by a lease.
- If a write request occurs at the server, the server sends invalidations to all valid proxies for D. It waits for an acknowledgement from each of the proxies and then allows the write to proceed.
- If the proxy has not responded it make two retries until it receives an acknowledgement. A Non-acknowledgement NOK is sent if one or more proxies are unreachable. If the acknowledgment is a NOK, then the server only risks writing the object without guaranteeing strong consistency.

State Space Overhead The state space overhead is governed by how the leases are computed. Our implementation computes the leases based on the EWMA algorithm that uses write history to determine a lease. When a server grants a lease to a proxy, it has to maintain some state information about the proxy. More popular objects usually require larger state. If longer leases are granted for all documents at the server, then state may exceed available limits. By granting shorter leases to popular objects, the server can adaptively control the amount of state required. Our implementation assigns a lease based on the write history of the document being requested. The lease is computed as follows. At any time instance t, the estimation of the time to next poll, $E(t)$, is given by

$$E(T) = \alpha(ts_t - ts_{t-1}) + (1 - \alpha)E(t - 1)$$

where $(ts_t - ts_{t-1})$ represents the last observed time between two consecutive significant updates. The value of α determines how much weight is given to the last observed value and how much to the previous estimates. The smaller the value of α, the more weight is given to

the older values in the series. Larger values of α result in more weight being given to more recent observations. We set our value for α at 0.9, a high value, because we want to give more weight to the recent observations.

SINC Proxy

The current proxy cache consistency mechanism is a weak consistency mechanism using TTL-based client polling. In our implementation, we augment this mechanism by splitting the cache and allowing invalidation on a portion of the cache, the documents that are shown as highly popular based on a threshold value and the GDSP algorithm. The other portion of the cache will continue to use a weak consistency algorithm. Specifically, we support the following functionality.

- Coexistence of a strong consistency mechanism with a weak consistency mechanism that automatically provides the support. We do not have to the configure the cache to use one mechanism and when we wish to switch, change the configuration and use the other mechanism.
- Issuing grant requests and storing lease expiration information in primary memory
- Issuing renewal *If-Modified-Since* requests and updating lease expiration information
- Responding to invalidation requests by negating the document and the lease in the cache
- Use of the GDSP algorithm to support management of the placement of cached items based on popularity

The protocol for handling the leases ensures that the client never receives a stale object for documents cached in the SR-Cache, the local portion of the cache that uses server invalidation. Assuming a proxy receives a request for document D,

- If the lease on D is valid then the proxy serves the document from its local cache.
- If the request is the first access to D then the proxy issues a grant request to the server.
- If the request is the first read following an invalidation then the proxy issues a grant request.
- If the lease on D has expired then the proxy issues a renew request The renew request is sent as an If-Modified-Since (IMS) request. If the response is a new lease with a Not-Modified response then the proxy can serve D from the SR-Cache. If the response is a HTTP OK 200 response, then the proxy stores the latest document piggybacked with the 200 response.

6.3.3 Discussion of Existing Cache Consistency Algorithms

As the proxy processes the requests, it uses the implemented cache consistency policies. In addition to SINC, we implemented three other policies. The three algorithms used for comparison are Client Polling, Client Polling with Adaptive TTL, and Server Invalidation. The Client Polling policy provides strong consistency with high network cost because it validates every read request sent to a proxy by forwarding the request to the server. The Client Polling with Adaptive TTL algorithm bounds the number of requests forwarded to the server by assigning a time when the documents should need validating by the server. The Server Invalidation algorithm invalidates the cached copies whenever the server modifies the document. The last algorithm would be the perfect solution if it did not have such high server state overhead in maintaining all proxies and documents that need invalidating.

6.3.4 Performance Metrics

The three main categories of performance metrics are staleness, control message, and server state overhead. Table 3.1 summarizes the measures being analyzed in this study. Here, we present where and how the performance metrics are collected.

The SINC proxies measure the number of control messages and file transfers that occur due to consistency. Also, the number of unnecessary updates, percentage of stale documents, and the period of staleness are measured. The workload is defined by the number of proxies, and by the request files that are passed to the proxies to generate requests to the server.

The SINC Server measures the average number of invalidations that occur per document, the average number of proxies contacted revealing the state size per proxy, the average lease length per proxy, and the average lease length per document.

We maintain two thresholds in the SINC Server. The S_{load} is the server load threshold. We use the number of leases maintained at the server to measure the load on the server. The N_{load} is the network load threshold. We use the number of messages received and sent by the server to measure network load.

6.3.5 Results

Control Message Overhead

The results for the number of control messages exchanged between the proxy and the server are shown in Table 6.5. We compare the performance of each strong consistency algorithm to the SINC algorithm to reveal the message count reduction. The performance of the client polling (CP), invalidation (INV), and SINC policies are measured across all workloads in terms of the average number of requests that the server received and the average number

of kilobytes that the server returned per document retrieval. The results indicate that the INV policy avoids transferring 55% to 60% of bytes as compared to the CP policy. The INV policy also transfers only a 5% difference in the number of bytes transferred by the SINC policy. The results further indicate that the INV policy issues 13 to 26 more requests to the server than the SINC policy. From this information, our maximum N_{load} threshold value is 14 with an average byte count of 166KB for the SINC Algorithm.

Table 6.5: Control Messages

	CM(Count)			*CM(KB)*		
	CP	*INV*	*SINC*	*CP*	*INV*	*SINC*
VT-CS	51.3	21.6	4.1	174.7	69.7	53.5
VT-CNS	106.8	27.5	14.3	430.0	174.0	165.6
NLANR	89.3	39.3	12.5	405.3	182.0	134.3

Proxy Cache Staleness Overhead

The server sends information on changes to the proxies it has listed in its state table. An experiment with 20 proxies and one server resulted in an average staleness value of 0.175% across all the proxies. This reveals that the server invalidation policy with the EWMA algorithm is highly scalable.

The amount of historical data affects the staleness measurements at the proxy. The proxy relies on the update information from the servers to make predictions on the time of the next significant update for a document in the SR-Cache. To quantify the importance of historical information availability, we perform a comparison between the Moving Average Estimator when all updates are available, and the Moving Average Estimator when only the last-update is available. The graphs in Figures 6.1 and 6.2 compare the various functions at the server for the two different scenarios. The all updates case gives the lowest staleness value as it has the maximum update information. The last updates case has incomplete update information and therefore underestimates the update frequency at the server. As a result, it performs fewer updates that results in a higher staleness period.

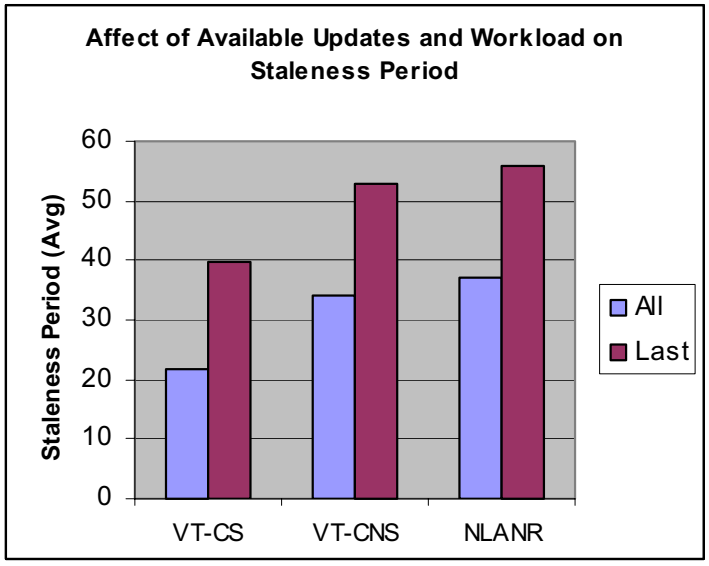

Figure 6.1: Staleness Period

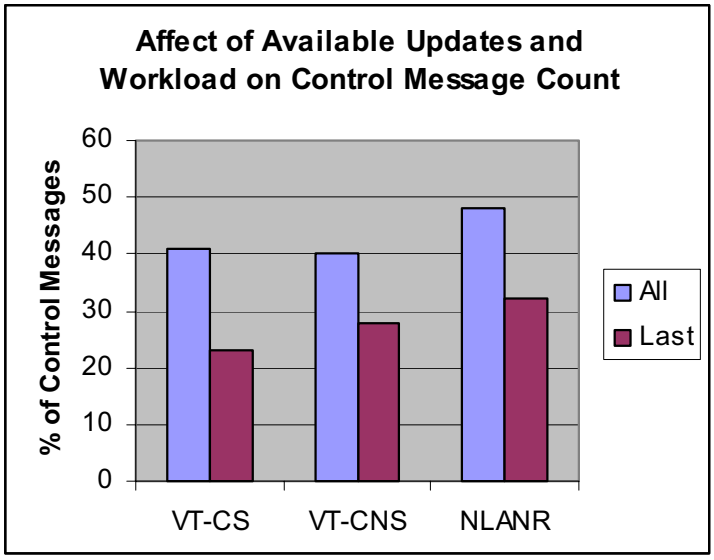

Figure 6.2: Staleness Count

Server State Overhead

The SINC and the Invalidation (INV) algorithms maintain state information for documents cached by a proxy. SINC maintains per-proxy leases and per-document leases. The server invalidation (INV) algorithm, to which we are comparing it, maintains the equivalent per-document leases. We capture the overhead associated with per-proxy leases. However, since the overhead is higher for the per-document leases, we present that data in Table 6.6. We measure the overhead of document leases by recording the number of per-document leases held for a client after it makes requests. Table 6.6 shows the average and maximum number of leases held for one client at each site. The highest daily number of leases for the SINC algorithm is 218 and was observed for the VT-CS site. The other workloads gave similar results for the total daily number of leases. Our results show that INV must maintain over 246 document leases per-proxy, and also must maintain leases even in instances where there is little update activity occurring. For the SINC Algorithm, the S_{load} threshold value is around 220.

Table 6.6: State Overhead Based on Lease Count

	Updates	*SINC Lease*		*INV Lease*	
		Avg	*Max*	*Avg*	*Max*
VT-CS	151	218	288	246	657
VT-CNS	88	180	454	270	770
NLANR	100	162	302	391	941

This experiment also consisted of measuring requests and responses occurring between the proxy and server. We first acquired results with server invalidation instrumented as the cache consistency algorithm, followed by SINC, our combined algorithm, as the consistency algorithm. The results show that the SINC framework decreases the server state overhead by 30% because about 70% of the documents needed state maintenance based on our workload and the calculations of the GDSP algorithm that ranks the cache documents in order of popularity. In Figures 6.3, we show the number of updates that occur that result in messages being sent to the proxy.

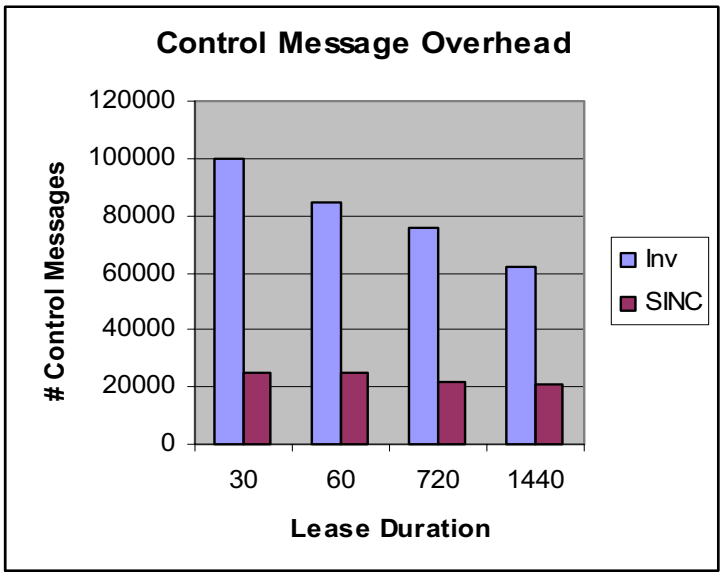

Figure 6.3: Server Control Message Overhead

In Figure 6.4, we show the percentage of documents that have information maintained for them based on the duration of the document-level lease values that are computed. The server space measurements have been made by periodically updating the server state and freeing space for proxies whose lease has expired. The plot in Figure 6.4 shows the variation in the state space measured by the number of valid leases issued. The plot for the SINC 1 day slowly rises to the maximum space as state information is kept indefinitely for the server invalidation algorithm. But in case of SINC, the plot depends on the lease duration as well as the request pattern. The maximum state information for a 30 minute lease that the SINC server has to store is one-fifth of server invalidation. This shows that the shorter lease tradeoff for server space still holds true in comparison with the number of control messages.

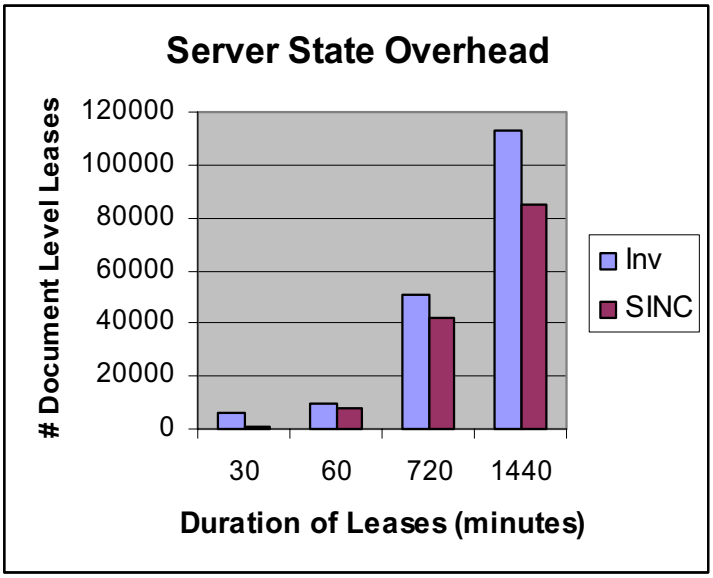

Figure 6.4: Document Level State Overhead

Finally, Figure 6.5 shows us the number of proxies that require state information. Together, these figures give a view of the state information maintained at the server.

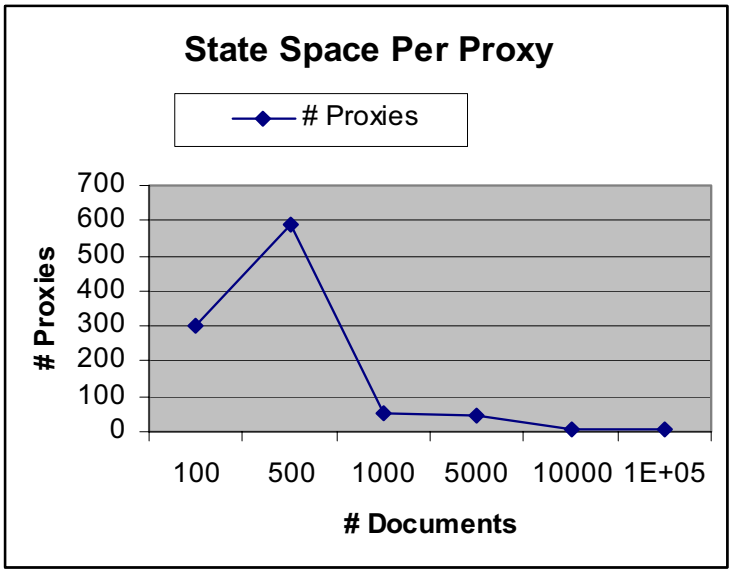

Figure 6.5: Proxy Level State Overhead

6.3.6 Summary of Results

This report presents the results of our investigation of several cache consistency techniques, broadly divided into an adaptive client poll (using TTL values), or server invalidation methods. In addition, we propose using a combination of adaptive client polling and server invalidation, which our evaluations show improved consistency. More specifically, we focused on the following techniques:

1. Client Poll with Adaptive TTL is presented in Section 5.3.1. The main idea is to poll the origin server whenever the TTL expires. TTL is an estimation of the duration during which the file is less likely to change. We use an adaptive TTL algorithm where the clients calculate TTL values based on the polling history. We refer to this algorithm as Adaptive Client Polling.

2. Server Invalidation is presented in Section 5.3.1. Whenever a file gets modified at its origin server, we construct an invalidation message and send it to every proxy for which we maintain state information. Proxies that cache the documents check their local copies upon receiving the invalidation message, and change their local copies' status to stale if they are older than the server copy.

3. Synchronous Nodes for Consistency (SINC), our algorithm, is presented in Section 5.3.1. It simulaneously employs both Client Poll with Adaptive TTL and Server Invalidation algorithms to provide consistency. The novel part of this negotiation scheme is that we do not switch between two cache consistency protocols, but actually employ them both simultaneously based on the read and write frequency of documents.

We derive the following conclusions from the results.

1. The number of invalidations represents those invalidations for which the lease was valid and hence results in a network message. This depends on the lease duration. Server Invalidation (INV) requires the maximum number of invalidations. As the lease duration decreases, the number of invalidations falls. The total number of control messages for SINC has been shown to be less than for server invalidation. One reason may be that the INV requires approximately 10% more grant requests than the SINC server invalidation. Grant requests are dependent on the number of invalidations which in turn depend on the lease duration. A larger number of invalidations leads to a higher number of grants.

2. Server Invalidation alone achieves almost nonexistent staleness when the update rate is lower than the request rate. SINC achieves only 10% better staleness than pure Invalidation when the update frequency and request frequency are comparable. With the assumption that there are normally much less updates than requests in the Web, we expect Server Invalidation alone to be sufficient in guaranteeing strong consistency. However, the server state overhead makes this consistency method less worth the risk.

3. Client Polling with Adaptive TTL does not guarantee strong consistency. As update frequency decreases by half, Client Polling with Adaptive TTL gives 20% better consistency. The poll message overhead is relatively consistent when the update frequency changes and is in general at least 5% lower than the invalidation message overhead when using Server Invalidation or SINC. In applications where the consistency requirements are less stringent, the Client Polling with Adaptive TTL algorithm is a good choice.

4. The TTL value determines the reach of validation request messages to the server. The staleness of cached copies increases as TTL values increase. When the TTL value gets large enough, both Invalidation and SINC achieve lower staleness rates.

5. Request rate does have an effect on staleness when using the SINC algorithm. It is the driving force of when to use server invalidation for the popular objects.

6. The invalidation message overhead depends on the update rate. If the update rate is low relative to the request rate, then the invalidation message overhead is also lower than the request message overhead.

7. The SINC algorithm results in significantly lower state space overhead by limiting how many documents and proxies to maintain the information, and applying dual leases to allow for proxy state release when needed.

6.4 Chapter Summary

The SINC framework achieves a good balance between stong consistency and weak consistency. With this negotiation scheme, we achieve very good consistency. For example, both the staleness period and staleness rate measured in the experiments are comparable with the strict invalidation scheme and require less state overhead. The Client Poll with Adaptive TTL algorithm has the most control message overhead. For example, its control message overhead could be higher than Client Poll if the TTL is longer than the request frequency. We argue that the update rate is generally much lower than the request rate in the Web. In this scenario, the invalidation message overhead could be much lower than the request and reply message overhead. On the other hand, Client Poll with Adaptive TTL alone provides less strong consistency, but has the advantage of the least control message overhead. Based on these, we conclude that SINC is a good choice for Web, and with the predictive algorithms at the proxy and server, the user does not have to decide when or which algorithm to choose. The algorithm is based on the request and update frequency at the proxy and server.

Chapter 7

Conclusions

This dissertation presents an overview of the design and management of a cache consistency algorithm and examines the proposed solutions in recent literature, many of which have not been deployed due to the potential for increased network traffic, and server load. With emerging new applications, cache management for web proxies remains a fertile research area, and many newly arising issues have not been well addressed. To conclude this dissertation, we list several possible research directions for proxy caches and the management of cache consistency.

7.1 The Importance of Proxy Caching Today

Web proxies currently play a vital role in efficient distribution of web content over the Internet. However, there are different opinions about the importance of caching in the Web. For example, Rabinovich and Spatscheck [56] argue that sizes of hard drives and memory grow very fast so that the cache capacity is not a constraint. Thus, the need for tuning of caching algorithms is not of great importance. Jin and Bestavros [36] argue against this and state that

1. the byte and hit ratios grow in a log-like fashion, and therefore increasing them by some small factor is equivalent to increasing the cache size;
2. the growth rate of Web content is much higher than the growth rate of memory sizes suitable for web caching;
3. small improvements in web caching algorithms might lead to savings in network traffic.

With the availability of large web browser caches, many assume that most repeated accesses to the same page from the same user will be satisfied from the browser cache and will never

be seen by the proxy cache. However, this does not support the fact that many documents are shared by different users and these request require proxy caching to reduce network bandwidth utilization. We feel that the importance of Web caching lies somewhere in the middle between these extremes. On one hand, some large commercial servers cache all its contents in memory and seem not to experience problems in all three points mentioned above. On the other hand, for proxies, such an approach is impractical, since the size of potential cacheable documents is large. Another area where caching may be applied is streaming media. In this case, the second and third points given by Rabinovich and Spatscheck might be of great importance.

7.2 Discussion of Research

WWW applications place diverse consistency requirements on proxies. This suggests the need for stronger consistency mechanisms to supplement weaker ones. Currently, the proxies in the WWW offer only weak consistency mechanisms. Previously proposed strong consistency mechanisms are not appealing for the Web environment, due to their large state space and message overhead. We have developed a new approach for providing strong and weak consistency based on leases that balances these tradeoffs. We present adaptive policies for determining suitable lease duration. A framework has been implemented that allows the co-existence of strong and weak consistency mechanisms. The SINC Web proxy and server have been implemented, and tested with leases and TTL mechanisms. Our experimental results validate our hypothesis that the leases algorithm represents a suitable tradeoff between server state space and network messages when we allow the proxy to influence how much state space is really needed at the server. Including read access calculations at the proxy and write access calculations at the server aids in a negotiation between the proxy and server while using the existing HTTP protocol.

A novel feature of SINC is that its' proxies make use of server invalidation and client polling simultaneously on a per-object basis. Earlier studies have analyzed the benefits of server invalidation versus client validation, but their comparisons were in a context where strong cache consistency was not imperative. More importantly, their evaluation was focused on assessing stale hit-rate using trace-driven simulations. In our work, we evaluate the competing mechanisms to keep strong consistency, at least in part, by analyzing the problem at the level of individual reads and writes of each document, which we believe yields substantial insight. The authors of [78] study the efficacy of server invalidations using a scalable distribution infrastructure, and provide several insights into the general problem of cache consistency. Our work advances the state-of-the-art beyond [78] in three major respects. First, proxies in our system dynamically choose the consistency mechanism based on their own observation of the request rates and update rates of objects. Prior knowledge of these statistics is not assumed. Second, our analytical results help us define the thresholds of the cache consistency policy at which proxy caches can apply the mechanism without having

to switch from one mechanism to another, but rather to employ both aspects of weak and strong consistency by minimizing the overhead produced by both.

7.3 Required HTTP Proxy and Server Changes

We make use of the HTTP protocol to support existing technologies that have proven themselves with a wide range of deployment. There are no changes to the HTTP protocol, but server administrators using the SINC technology must be aware of the message requirements to support SINC server invalidation. These messages are discussed in Chapter 5 of this dissertation. Two basic ideas are incorporated in the SINC framework: Server-driven invalidation and Client-driven validation. Some requirements of SINC server and proxy nodes are given below.

7.3.1 Server-driven Invalidation

The SINC server sends document or proxy invalidations to the SINC proxies. This allows the server to control the SINC activity according to the server's own load. The connection between the SINC proxy and server can be established by either node, and could be persistent. Persistent connections are specified in HTTP/1.1 and facilitates monitoring of the update guarantee through positive acknowledgements. The entity extensions of HTTP/1.1 are used in request and response messages provided by HTTP/1.1 for the server to notify the proxy of modifications. A SINC server knows the state of the content and generates document updates. An invalidation is indication from a SINC server to a SINC proxy that the server copy has changed.

7.3.2 Client-driven Validation

The SINC proxy queries the SINC server for the freshness status of a subset of cached objects. This allows the proxy to control the SINC activity according to the proxies own load and configured preferences. The SINC server responds with the latest changes since the last time the proxy polled the server. It makes use of the HTTP Etag, timestamp, and/or version number. Whether and when the proxy sends request to the SINC server is determined by the consistency guarantee that the proxy is committed to provide. A SINC proxy knows the acess pattern and rate of requests, needs to knows the state of the contents at the server, and receives document updates from the SINC server. A SINC proxy must not serve an invalidated cached document without an HTTP **Get-If-Modified-Since** message. SINC must guarantee that either a SINC proxy sees an update as intended or be able to detect that it might have missed an update. While participating in a SINC message exchange, a

proxy cache should never return potentially stale data without a warning that the data is possibly stale. This can be done via the *110/111 HTTP Warning* response code.

7.4 Research Contributions

This research has made the following contributions.

- We have performed an analytical investigation of four proposed consistency algorithms to identify how read and writes could affect the operation of these algorithms and to provide a taxonomy of when the algorithms are most appropriate.
- We have used an approach for studying Web resources and understanding how they change, which we discussed in Chapter 4. Instead of examining proxy and server logs, as was done by previous studies, we identify a set of Web resources to study and then actively retrieve them over a period of time. This approach allowed the investigation of several different kinds of resources that were separated into categories, instead of being constrained to the data available in logs or traces. A study was done using these retrieved resources to determine the rate at which documents on the Web change and to determine if the types of documents affect that rate of change.
- We examine a group of server logs to reveal how many messages are required to communicate changes to proxy caches that hold copies of the resources. This reveals the number of Web sites that should be notified of changes in a document maintained by the server if invalidation were to be used. We also measure the number of references made to each server document by each proxy. This aided in determining the amount of per-proxy state information that is required at the server if invalidation were used. The overall intent of this contribution was to discover the limits within which server invalidation would be beneficial.
- We investigated the amount of unnecessary traffic that is generated on the Web today. This provides estimates of the possible reduction of unnecessary messages that could occur if our approach were available for use in the Web today. Our background study determines whether significant network savings would occur if server invalidation played some part in the maintenance of consistency. These evaluations show the potential for substantial improvement offered by our algorithm.
- We have devised a novel framework, called SINC, that chooses the most appropriate consistency strategy for a set of documents cached at proxies. The framework assesses the read and update frequency of documents and allows the request to drive the consistency at both the proxy and server. SINC achieves strong consistency for a subset of documents, the most popular objects that are cached at the proxies. The objects marked as less popular, based on the read frequency, use the current weak consistency

method employed on the Web today. The novel part of this approach is that we employ two predictive algorithms to access the popularity and modification frequency of documents. The algorithms have been in existence, but not applied in the manner stated in this dissertation.

- We have designed and implemented our SINC algorithm in a system involving three major components. A log parser drives the proxies to propogate request to the server. The proxy employs the idea of a split cache allowing two consistency mechanisms to be used in the Web simultaneously. A server uses a predictive calculation to access update frequency by used historical data about prior modifications. We employed the existing HTTP/1.1 protocol infrastructure to accomplish the communication between proxies and the server. The implementation is a major contribution because it validates the idea that we can provide strong cache consistency without having substantial network and server cost associated with it and without incorporating change into the existing practices.
- We evaluated our algorithm and compared it with existing and proposed algorithms using a testbed of cluster nodes to run an emulation of the Web. This included the used of actual Web proxies and servers that are driven by a workload of server and proxy logs.

7.5 Future Research

7.5.1 Future Direction of Proxy Caches

Large-scale distributed systems often address performance and quality of service by caching and replication. Caching aims to move content closer to users to help decrease load on origin servers, eliminate redundant data traversal through the network, and reduce user-perceived latency. Traditional caching has limited effectiveness due to diversity of resource access, increasing dynamic content, and concerns about consistency of cached responses. In today's scenario, each site needs additional mechanisms to deliver acceptable performance amidst flash crowds and Denial of Service attacks.

In recent years, due to growing user needs and high performance application development, strong consistency is desired for its improved performance. Existing cache consistency research mainly focuses on time-based consistency policies. Applications that use dynamically generated web objects are increasingly used to provide web content. Replicating a dynamic web object requires replicating both the code and the data that the code acts upon. Consistency for such applications has not received much attention and more research work can be carried out to provide a scalable solution. Similarly, multimedia applications are contributing much to Web traffic, and providing scalable consistency mechanisms for them is also an

area that needs to be considered. Some of the aspects of proxy caches and cache consistency that require additonal focus are described below.

Caching Dynamic Content

A large portion of web traffic today is dynamic content. These types of data are normally marked as "uncacheable". However, it has been found that many resources that are marked as dynamic are static resources with a dynamic component added. For example, a simple static page with an access counter added in the content would deem the entire page as dynamic and therefore "uncacheable". An alternative approach could be in the creation of the page. If the page was developed as a static page with the counter as an embedded link, then the page would be cacheable and only the counter would be marked as changeable. Proxy caching techniques can be modified to recognize this type of phenomena and to cache portions of pages that are considered dynamic, but are mostly static with dynamic portions. To further improve web performance, reverse caching has been suggested to make dynamic content cachable and manageable. However, the challenge remains for how to efficiently maintain consistency between the cached content and the dynamically changing source data.

Caching Streaming Objects

It is predicted that streaming media like music or video clips will represent a significant portion of web traffic over the Internet. Due to the distinct features of streaming objects, such as large size, long duration, intensive use of bandwidth, and interactivity, conventional proxy caching techniques may not be entirely applicable, or may perform inefficiently. Most current proxies do not cache streaming media due to their large size. Without the benefit of caching, access latency in retrieving documents remain a concern. The difficulty lies in the decisions that must be made such as how to optimally choose the portions of the data to cache and how the proxy and origin server can jointly deliver streaming objects to clients. One possibility is to cache a portion of the streaming data in the same way that a portion of dynamic data can be cached. Caching the static portion of this data would provide some advantage and if used with a prefetching technique, we can satisfy the request with reduced delay.

Security

The use of proxies also complicates the task of maintaining the security of data transmitted between clients, proxies, and servers. For standalone proxy caches, there is a need for protection against privacy concerns and denial-of-service attacks. For cooperative caches, there is a need to establish a trust relationship among participants. To alter data, an attacker

can now target a proxy in addition to the origin server. Therefore, it is crucial to ensure the integrity of the cache content in a proxy.

Denial of Service is a relatively new type of Internet attack where a group of users of a specified service deny access to another group of users. This makes the service unavailable to the latter group for a period of time that exceeds the intended service time. This loss of service may range from the suspension of network services such as e-mail to that of an entire browser or server having to cease operation. A security attack can occur at the server in the form of a denial of services to clients. An attack can occur on the network causing it to limit transmission of required messages to all clients or a subset of clients. There is also the possibility of flooding the network with enough messages to restrict its proper use. Another concern in the use of server invalidation is in allowing hackers to saturate proxy entries in a server's state table which would exclude other proxies from being invalidated. We handle this problem in our research by bounding the amount of state that the server maintains on a per-proxy basis. The bound is $O(p * d)$. In addition, we can easily purge assumed hacker proxies by revoking their per-proxy lease. Further analysis is required to assess the impact of bounding the proxy storage during malicious attack attempts. We can also employ the idea of certificates to authenticate whether a proxy is valid or invalid.

Content Distribution Networks

CDN's such as Akamai [3] achieve scalability through replication. Documents requested by clients can generally be categorized as static or dynamic documents. It is the goal of CDN developers to design an efficient scheme for updating documents of any category and maintaining consistency of documents at replicas with the origin server. The exact cache consistency mechanism and degree of consistency employed by proxies depends on the nature of cached data, because all types of data need not employ the same level of consistency guarantees. Our SINC framework would work well in such an environment because it includes the separation of cached objects within the algorithm design thereby further reducing server load and reducing the increased network bandwidth that is inherent in such a network where multicast technology is used to maintain consistency among replicas.

Cache Management in Mobile Environments

Caching techniques have been successfully employed to overcome some of the problems posed by disconnection and/or limited bandwidth in mobile environments. Demand driven and prefetching techniques have been found to have limited success in such environments. Mobile environments involve accessing data from a wireless network connection which is highly dynamic, unreliable and insecure. One of the major issues that wireless networks face is frequent disconnections of clients which may result in loss of data packets. Caching is used in mobile networks to deal with disconnections, but it is essential to ascertain that the

client caches have updates that are consistent with the servers. Some proposed algorithms include polling based or server invalidation based techniques. All of the advantages and disadvantage inherent in each technique applies to the wireless environment as well. Use of the SINC algorithm would overcome some of the disadvantages of general consistency algorithms, but require changes to allow for resumption of network activity when mobile nodes are not connected.

7.5.2 Cache Consistency Management

Invalidation Algorithm Modifications

In addition to the basic algorithms we discuss in this dissertation, there are many variants of invalidation that can be investigated such as Piggyback Server Validation [39], Invalidation with Delta Encoding [50], and Volume Lease Invalidation [75]. Our research allows the propogation of invalidates to the proxy when a write occurs. This is sufficient to prevent proxies from continuously contacting the server for updates upon request from clients. However, this is mostly useful when the popularity of an object is low. If an object is popular, and a server propagates only invalidates, then every other request for the object generates GET messages for the objects. This results in extra network traffic to get the document. Therefore, if an object is popular, it is more efficient to send the updates rather than just the invalidates. More investigation is needed to reveal the actual savings when updates are sent to the proxies which would alleviate the proxy from having to send GET requests on subsequent reads. Our future work involves combining the best elements of existing consistency mechanisms to allow proxies and servers to adapt to conditions of the Web and decide which mechanism to invoke.

Co-existence of Cache Consistency and Replacement Algorithms

Another area of future work is in determining the effect of cache replacement policies on the management of consistency in proxies. Since replacement policies could interfere with the application of a consistency mechanism, some research can deal with how to best incorporate both to coexist in Web proxy caches. For example, the current Web caches use Least Recently Used (LRU) to evict documents from the cache. The analysis here does not address the possibility of a replacement policy removing documents that may have otherwise been considered a part of the cache consistency split cache scheme. Although we make use of the GDSP replacement policy to place documents in the cache in order of popularity, we do not address the possiblity of documents being removed. For the SINC framework, it may be as simple as removing the bottom items of the G-Cache or application of a more elaborate replacement scheme. Evaluations are required that consider this issue.

7.5.3 Proxy Cache Evaluation

Effects of Workload Characterization

One important aspect of any experimental work is the workload used in the experiments. While we introduced the workloads that we used in our studies, the type of workload is an evaluation issue. Usually, we are concerned with methods that learn models of typical Web usage. However, those models often only identify common occurrences among resources with the ultimate goal of generating accurate predictions about resources that might be requested. However, there are multiple types of relationships between Web resources that might occur in Web logs. One possible relationship is that of an embedded object and its referring page. Embedded objects include image, audio, or source code that are automatically retrieved by the browser when the referring page is requested. Another relationship is that of traversal, when a user clicks on a link from the referring page to another page. The first traversal is a result of how the content was prepared. The second traversal, embedded links, while also existing because of placement by the content provider, also includes a function of how users navigate through the Web pages. Many researchers distinguish between such relationships and choose not to make predictions for embedded objects. They are concerned with the analysis of the sequence of requests made by the user and not the set of objects automatically requested as additional resources. We include limited analysis of embedded objects in this research. Employing the split cache theory and separating the documents based on content type, referring page, or embedded links could be a possible enhancement to the consistency algorithm. This would require further evaluation using these types of categories.

Nature of Document Changes

Although we recommend the updating of very popular and frequently changing Web documents using SINC, a combination of a strong and weak consistency algorithm that capitalizes on the strength of both, we believe our study could benefit from an analysis of the nature of the document changes. How much of a change constitutes the need for a fresh document? Were there small changes like a date or spelling error? Or were there huge paragraphs being rewritten? These types of decisions can help us to determine when the stale copies are actually acceptable. This type of data could then be incorporated into determining the staleness of a document and affect which type of consistency algorithm is invoked on documents. Our prediction algorithm could then be driven by the amount of staleness as well as the popularity of the document.

7.6 Chapter Summary

We hypothesized in this dissertation that we can improve upon existing invalidation-based techniques so that cached Web documents can maintain consistency efficiently. Our goal in this dissertation is to minimize the return of stale documents, reduce the network bandwidth through reduction in unnecessary requests, and to lower server overhead. In the previous chapters, we have presented our approach, design, implementation, and evaluation of a new framework that addresses this issue. In the end, we validate our claim that the framework improves upon existing and proposed approaches.

This chapter summarizes the contributions of this dissertation and presents ideas for future work. In this chapter, we discuss the importance of proxy caching today, summarized the contributions of this dissertation, and pointed out why we believe our contributions are important. We also listed and discussed a few ideas for future work. We believe exploring these directions may result in new extensions to SINC and may lead to new applications of the ideas in SINC to a different architecture.

Bibliography

[1] 100hot.com. <URL: http:www.hot.com>.

[2] Pc magazine top 100 listing. <URL: http:www.zdnet.compcmagspecialweb100index.html>.

[3] Akamai technologies, inc. <URL: http:www.akamai.com>.

[4] Apache web proxy cache. <URL: http:httpd.apache.org>.

[5] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of SIGMETRICS*, pages 126–137, Philadelphia, PA, April 1996. ACM.

[6] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking java against c and fortran for scientific applications. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 97–105, Palo Alto, California, 2001.

[7] K. Candan, W. Li, Q. Luo, W. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of ACM SIGMOD 2001*, pages 532–543. ACM, May 2001.

[8] L. Cao and M. Ozsu. Evaluation of strong consistency web caching techniques. In *Proceedings of World Wide Web: Internet and Web Information Systems*, volume 5, pages 95–123, December 2002.

[9] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of USENIX Symposium on Internet Technology and Systems*, December 1997.

[10] P. Cao and C. Liu. Maintaining strong cache consistency in the world wide web. In *Proceedings of ICDCS'97*, pages 12–21, May 1997.

[11] V. Cate. Alex - a global file system. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1–12, May 1992.

[12] A. Chankhuthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrel. A hierarchical internet object cache. In *Proceedings of the Usenix Technical Conference*, January 1996.

[13] L. Cherksova. Improving www proxies performance with greedy-dual-size-frequency caching policy. Technical Report HPL-98-69 (R.1), Hewlett Packard Computer Systems Laboratory, 1998. <URL: http://www.hpl.hp.com/techreports/98/HPL-98-69R1.pdf>.

[14] B. Choi and N. Bharade. Network traffic reduction by hypertext compression. In *In Proceedings of the International Conference on Internet Computing*, pages 877–882, 2002.

[15] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving end-to-end performance of the web using server volumes and proxy filters. In *In Proceedings of the ACM SIGCOMM'98 Conference*, 1998.

[16] D. Comer and D. Stevens. *Internetworking With TCP/IP*, volume 3. Prentice-Hall, London, 1997.

[17] I. Cooper, P. Gauthier, J. Cohen, M. Dunsmuir, and C. Perkins. The web proxy auto-discovery protocol. *IETF Internet Draft*, November 2000. <URL: http://www.wrec.org/Drafts/draft-cooper-webiwpad-00.txt>.

[18] C. Cunha. *Trace Analysis and its Applications to Performance Enhancements of Distributed Information Systems*. PhD thesis, Computer Science Department, Boston University, 111 Cummington St, Boston, MA 02215, 1997.

[19] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of www client-based traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, 111 Cummington St, Boston, MA 02215, July 1995.

[20] B. Davidson. A survey of proxy cache evaluation techniques. In *Proceedings of the Fourth International WWW Caching Workshop*, March 1999.

[21] B. Davidson. *Learning Web Request Patterns*, pages 435–460. August 2004.

[22] B. den Boer and A. Bosselaers. Collisions for the compression function of md5. In *Advances in Cryptology, Proceedings Eurocrypt '93*, pages 293–304, 1994.

[23] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. *Proceedings of the International WWW Conference (10)*, May 2001.

[24] F. Doswell and M. Abrams. The effectiveness of cache coherence implemented on the web. In *Proceedings of the Workshop on Caching, Coherence and Consistency (WC3 01)*, June 2001.

[25] 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 Internetworking Technologies and Systems*, pages 147–158, December 1997.

[26] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. In *Proceedings of the IEEE Infocom'00*, pages 834–844, March 2000.

[27] Z. Fei. A new consistency algorithm for dynamic documents in content distribution networks. *Journal of Parallel and Distributed Computing*, 63:916–926, October 2003.

[28] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and Tim Berners-Lee. Hypertext transfer protocol - http/1.1. Request for Comment RFC 2616, Network Working Group, June 1999. <URL: http://www.w3.org/Protocols/rfc2616/rfc2616.html>.

[29] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 141–152, January 1996.

[30] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[31] J. Gwertzman and M. Seltzer. An analysis of geographical push-caching. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, pages 51–55, Orcas Island, WA, May 1995.

[32] J. Gwertzman and M. Seltzer. World-wide web cache consistency. *USENIX Symposium on Internetworking Technologies and Systems*, pages 141–152, January 1996.

[33] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 2nd edition, 1990.

[34] C. Huang and T. Abdelzaher. Towards content distribution networks with latency guarantees. In *Proceedings of IEEE International Workshop on Quality of Service*, pages 51–55, May 2004.

[35] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley, New York, 1991.

[36] S. Jin and A. Bestavros. Popularity-aware greedy dual-size web proxy caching algorithm. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, April 2000.

[37] A. Kahol, S. Khurana, S. K. Gupta, and P. K. Srimani. A strategy to manage cache consistency in a distributed mobile wireless environment. volume 12, pages 686–700.

[38] B. Krishnamurthy and J. Rexford. *Web Protocols and Practices.* Addison Wesley, 2725 Sand Hall Road, Menlo Park, CA 94025, 2001.

[39] B. Krishnamurthy and C. E. Wills. Piggyback server invalidation for proxy cache coherency. In *Proceedings of the Seventh International World Wide Web Conference*, volume 30.

[40] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Transaction on Networking*, 8(5):568–582, October 2000.

[41] R. Lancellotti, M. Colajanni, and B. Ciciani. A scalable architecture for cooperative web caching. In *Proceeding of the Workshop in Web Engineering, Networking 2002*, May 2002.

[42] E. Leontiadis, V. Dimakopoulos, and E. Pitoura. Cache updates in a peer-to-peer network of mobile agents. In *Proceedings of the 4th IEEE International Conference on P2P Computing (P2P '04)*, Zurich, Switzerland, August 2004.

[43] D. Li, P. Cao, and M. Dahlin. Wcip: Web cache invalidation protocol. *IETF Internet Draft*, November 2000.

[44] D. Li and D. Cheriton. Scalable web caching of frequently updated objects using reliable multicast. pages 1–12, October 1999.

[45] P. Lorenzetti, L. Rizzo, and L. Vicisano. Replacement policies for a proxy cache. Technical Report LR-960731, University di Pisa, December 1996.

[46] A. Luotonen. *Web Proxy Servers*. Prentice-Hall, London, 1998.

[47] A. Luotonen and T. Berners-Lee. CERN httpd 3.0 guide. <URL: http://www.w3.org/hypertext/WWW/Daemon/User/Guide.ps>, October 1994.

[48] R. McGrath. Caching for large scale systems.

[49] M. Mikhailov and C. E. Wills. Evaluating a new approach to strong web cache consistency with snapshots of collected content. pages 599–608, May 2003.

[50] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta-encoding and data compression for http. In *Proceedings of the ACM SIGCOMM '97 Conference*, September 1997.

[51] Message passing interface (mpi). <URL: http:www..llnl.gov/computing/tutorials/mpi>.

[52] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Scalable consistency maintenance in content distribution networks using cooperative leases. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):813–828, July 2003.

[53] A. Pashalidis. A cautionary note on automatic proxy configuration. In *Proceedings of IASTED International Conference on Communication, Network, and Information Security*, pages 153–158, New York, December 2003.

[54] V. Paxon and S. Floyd. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, August 2001.

[55] J. E. Pitkow and M. M. Recker. A simple yet robust caching algorithm based on dynamic access patterns. In *Proceedings of the 2nd International WWW Conference*, pages 1039–1046, Chicago, October 1994.

[56] Michael Rabinovich and Oliver Spatscheck. *Web Caching and Replication*. Addison Wesley, 2725 Sand Hall Road, Menlo Park, CA 94025, 2002.

[57] L. Ramaswamy and L. Liu. Cache clouds: Cooperative caching of dynamic documents in edge networks. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS) 2005*, Columbus, Ohio, June 2005.

[58] R. Rivest. Rfc 1321 - the md5 message-digest algorithm. Request for Comment RFC 1321, Network Working Group, June 1999. <URL: http://www.faqs.org/rfcs/rfc1321.html>.

[59] P. Rodriguez and E. W. Biersack. Continuous multicast push of web documents over the internet. *IEEE Networks*, 21(2).

[60] L. Boszormenyi S. Podlipnig. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, December 2003.

[61] Mukesh Singhal and Neranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, New York, 1994.

[62] Squid web proxy cache. <URL: http:www.squid-cache.org>.

[63] R. Sundaresan, M. Lauria, T. Kurc, S. Parthasarathy, and J. Saltz. Adaptive polling of grid resource monitors using a slacker coherence model. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, pages 260–269, June 2003.

[64] M. Tanenbaum. *Principles of Modern Operating Systems*. Prentice-Hall, London, 2001.

[65] B. Urgaonkar, A. Ninan, M. S. Raunak, P. Shenoy, and K. Ramamritham. Maintaining mutual consistency for cached web objects. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, AZ, April 2001.

[66] Z. Wang, S. Das, H. Che, and M. Kumar. A scalable asynchronous cache consistency scheme (saccs) for mobile environments. *IEEE Transactions on Parallel and Distributed Systems*, 15(11):983–995, November 2004.

[67] D. E. Wessel and K. Claffy. Icp and the squid web cache. *IEEE Journal on Selected Areas in Communication*, 16(3):345–357, April 1998.

[68] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for world-wide web documents. In *ACM SIGCOMM'96 Conference*, pages 293–305, Stanford University, California, August 1996.

[69] C. E. Wills and M. Mikhailov. Toward a better understanding of web resources and server responses for improved caching. pages 1231–1243, 1999.

[70] R. Wooster and M. Abrams. Proxy caching that estimates page load delay. In *Proceedings of the 6th International World-wide Web Conference*, pages 325–334, Boston, April 1997.

[71] K. Worrell. Invalidation in large scale network object cache. Master's thesis, University of Colorado, Boulder, 1994.

[72] J. Xu, B. Li, and D. Lee. Placement problems for transparent data replication proxy services. 20(7):1383–1398, September 2002.

[73] J. Xu, J. Liu, B. Li, and X. Jia. Caching and prefetching for web content distribution. 6(4):54–59, July 2004.

[74] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using leases to support server-driven consistency in large scale systems. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 285–294, May 1998.

[75] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume leases for consistency in large-scale systems. *IEEE Transaction on Knowledge and Data Engineering*, 11(4):563–576, January 1999.

[76] J. Yin, M. Dahlin, and A. Iyengar. Engineering server-driven consistency for large-scale dynamic web services. In *Proceedings of the 10th International World-wide Web Conference*, pages 46–57, Hong Kong, May 2001.

[77] N. Young. Online caching as cache size varies. In *Proceedings of the Symposium on Discrete Algorithms*, pages 46–57, Hong Kong, 1991.

[78] H. Yu, L. Breslau, and S. Schenker. A scalable web cache consistency architecture. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 163–174, September 1999.

Vita

Felicia R. Doswell

Felicia Doswell received her M.S. degree in Computer Science from Georgia Tech in September 1991 and her B.S. in Computer Science from Norfolk State University in May 1990. She is a member of Upsilon Pi Epsilon Computer Science Honor Society. Her professional memberships include IEEE and ACM organizations. For the past two years, Mrs. Doswell has worked in academia at Virginia State University and Norfolk State University. Prior to pursuing her doctoral studies, she also worked in the Computer Science industry for 5 years with AT&T Bell Laboratories and Nortel.

Publication List

[1] F. Doswell. *Web-Related Careers.* Prentice-Hall, 1998.

[2] F. Doswell. A framework for providing strong cache consistency on the web. In *SIGCSE Doctoral Consortium*, February 2003.

[3] F. Doswell and M. Abrams. The effectiveness of cache consistency on the web. Technical Report TR 00-02, Virginia Tech, August 2000.

[4] F. Doswell, M. Abrams, and Srinidhi Varadarajan. The effectiveness of cache coherence implemented on the web. In *Proceedings of the Workshop on Caching, Coherence and Consistency (WC3 01)*, June 2001.

[5] F. Doswell, H. Harley, T. Lewis, C. Seals, and Dr. G. Scales. Adapting to life as a graduate student: Getting up to speed on information technology. In *Student Guide to Graduate and Professional School Success*, 2000.