COOPERATIVE WEB CACHING:

A VIABILITY STUDY AND DESIGN ANALYSIS

APPROVED BY SUPERVISING COMMITTEE:

Dr. Kay A. Robbins, Supervising Professor

Dr. Clinton L. Jeffery, Co-Advisor

Dr. Samir R. Das

Dr. Richard F. Sincovec

Dr. Weining Zhang

Dr. Lawrence Williams, Outside Member

Accepted:

Dean of Graduate Studies

Copyright 2000
Sandra Goles Dykes
All Rights Reserved

Dedication

Writing a dissertation requires persistence and hard work, however, it is also undeniably fun. The research allows you to explore your own interests, to experience the thrill of uncovering something new, and to piece together the answers to interesting questions. I was fortunate to have advisors who, while not neglecting the work component, emphasized and added to the fun component. In particular, I want to thank Kay Robbins for the hours we spent probing different aspects of the viability and analysis issues. Her unfaltering enthusiasm, insights, and careful scrutiny of results carried me through the most difficult phases and vastly improved the quality of the work. The other major figure in my academic family tree is Clint Jeffery, who is responsible for leading me into the fields of networking and Web caching. Not only is Clint a pleasure to work with, his wide range of interests and expertise have added important perspectives to all the projects we have worked on.

The topic for this dissertation grew from a cooperative Web cache design originally proposed by Clint and Samir Das. In addition to the original concept, Samir contributed an expert background in networking and a connection to other topics in the field. I also want to thank Samir for the enjoyable and valuable classes he taught in performance evaluation and analysis of algorithms which provided many of the skills necessary for the data analysis in this work.

When immersed in a project, it is often too easy to let details interfere with a clear description of the work. Comments from Richard Sincovec, Weining Zhang, and Larry Williams helped with the focus and clarity of the explanations. I would also like to thank Dmitry Gokhman, Xiaoping Chen, and Roy Schwaerzel for their comments on my original proposal.

Finally, I want to express my deepest appreciation for the support and encouragement of my family: Jim, Travis, Sean, and two sets of wonderful parents, mine and Jim's. My family's understanding and support made it possible to spend the hours required for this work, and their encouragement made it worthwhile.

COOPERATIVE WEB CACHING:

A VIABILITY STUDY AND DESIGN ANALYSIS

by

SANDRA GOLES DYKES, B.S., M.S.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences and Engineering
Division of Computer Science
August 2000

Acknowledgements

This research was supported by the National Science Foundation under grant CDA-9633299, by a graduate fellowship from the NASA Texas Space Grant Consortium, and by support from the University of Texas at San Antonio Division of Computer Science. Additional support for conference and travel expenses was provided by USENIX and the NSF Student Travel Grant program in conjunction with the IEEE Communications Society. I would also like to thank the Department of Computer Science at Texas A&M University and the Department of Computer Science at the University of Houston for access to computer systems used in a portion of the experiments.

May 2000

COOPERATIVE WEB CACHING:
A VIABILITY STUDY AND DESIGN ANALYSIS

Sandra Goles Dykes, B.S., M.S., Ph.D.
The University of Texas at San Antonio, 2000

Supervising Professor: Kay A. Robbins

A cooperating Web cache is a group of HTTP proxy servers that share their cached objects. Although researchers agree that proxy server caching improve performance, there is considerable debate about how, or even if, proxies should cooperate. This dissertation evaluates the viability of proxy cooperation using empirical hit rates and measured Web response times. We find proxy cooperation to be only marginally viable if the sole criteria is average response time. However, a more detailed examination shows proxy cooperation can reduce the variability in response time and the number of pathologically long delays. Moreover, cooperation offers a means of avoiding router congestion and improving network throughput.

Hit rates are determined from traces supplied by the National Laboratory of Applied Network Research (NLANR) global cache hierarchy. As part of the trace analysis, we quantify and correct three often ignored effects: cache warmup, uncachability due to unrecorded HTTP response headers, and sibling hits. The analysis shows clients often repeat requests for uncachable documents, and these repeats substantially inflate the ideal hit rates and Zipf's Law parameters used in cache modeling. Our work establishes far lower estimates of Web sharing than previously reported and reveals statistically significant differences between weekdays and weekends and between different caches.

Because remote caches are not inherently faster than remote Web servers, cooperation creates a speed advantage by allowing clients to select the fastest candidate site. This work evaluates client-side selection algorithms that use dynamic network probes, statistical bandwidth, statistical latency, random selection, and hybrid selectors. Of these, probes result in the fastest download time, least variability, and fewest pathological delays. The advantage of cooperation stems from the ability of probes to improve response time by avoiding congested network routes.

Other contributions of this dissertation include a taxonomy for cooperative Web caches and an analysis of cooperation designs. The analysis supports a mesh organization with independent metadata propagation. Our data on the diurnal nature of Internet load and the longevity of cached objects suggests nighttime metadata exchanges between proxy caches can achieve an efficiency of over 70% while adding negligible overhead to the average response time.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The World Wide Web

This dissertation focuses on the viability and advisability of cooperation between Web proxy caches. Before addressing cache cooperation, we briefly describe the World Wide Web, caching in general, and how caching is currently deployed on the Web.

1.1.1 TCP, routers, and packet loss

The Internet is a global network of computers that communicate using the IP network protocol. Physical links transmit message signals between two network points. Switching devices such as routers decide which links constitute a path between sender and receiver. Built on top of the IP protocol is TCP, the transmission control protocol. While IP controls communication on the link between two points, TCP controls the end-to-end communication. TCP is responsible for decomposing messages into network data packets, reassembling the messages on the receiver, and ensuring messages are delivered without error.

Network congestion occurs when data packets arrive at routers faster than the router can process them and the router queue is already full. When a router queue overflows, the data packets are simply dropped. Routers do not inform senders when packets are dropped: the only mechanism for detecting packet loss is lack of a receiver acknowledgement. When no acknowledge is forthcoming, TCP resends the packet and lowers the sending rate to reduce network congestion. Consequently, a connection's throughput may fall severely as a result of packets drops.

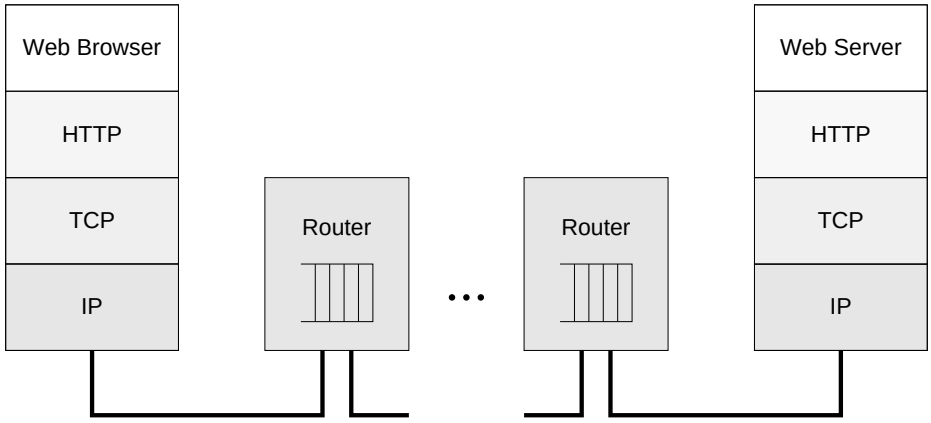

Figure 1.1: *Network communication layers between Web clients (browsers) and Web servers.*

1.1.2 HTTP Protocol

The World Wide Web is the set of client and server applications on the Internet that communicate via the Hypertext Transport Protocol (HTTP) [28]. HTTP is implemented using TCP in the same way that TCP is implemented using IP. HTTP specifies the rules for interpreting and handling text documents and multimedia; however, it relies on TCP to ensure documents are transmitted completely and correctly. Web clients initiate communication by sending an HTTP request to the Web server. Upon receiving the HTTP request, the Web server replies with an HTTP response header and, if the request is successful, with the requested object. Figure 1.1 illustrates the relationship between Web browsers, Web servers, and the underlying HTTP and TCP protocols.

Although HTTP is the dominant application protocol on the Internet [53], the traffic contains other important protocols such as SMTP (Simple Mail Transport Protocol), FTP (File Transport Protocol), and NNTP (Network News Transport Protocol) [60]. Traffic from all applications compete at routers and interfere with each other during periods of high traffic load.

1.1.3 Internet paths

On the Internet, the path between a Web browser and server usually contains 15 – 20 links [18], each of which provides opportunity for packet loss. For example, Figure 1.2 shows the links traversed on the path

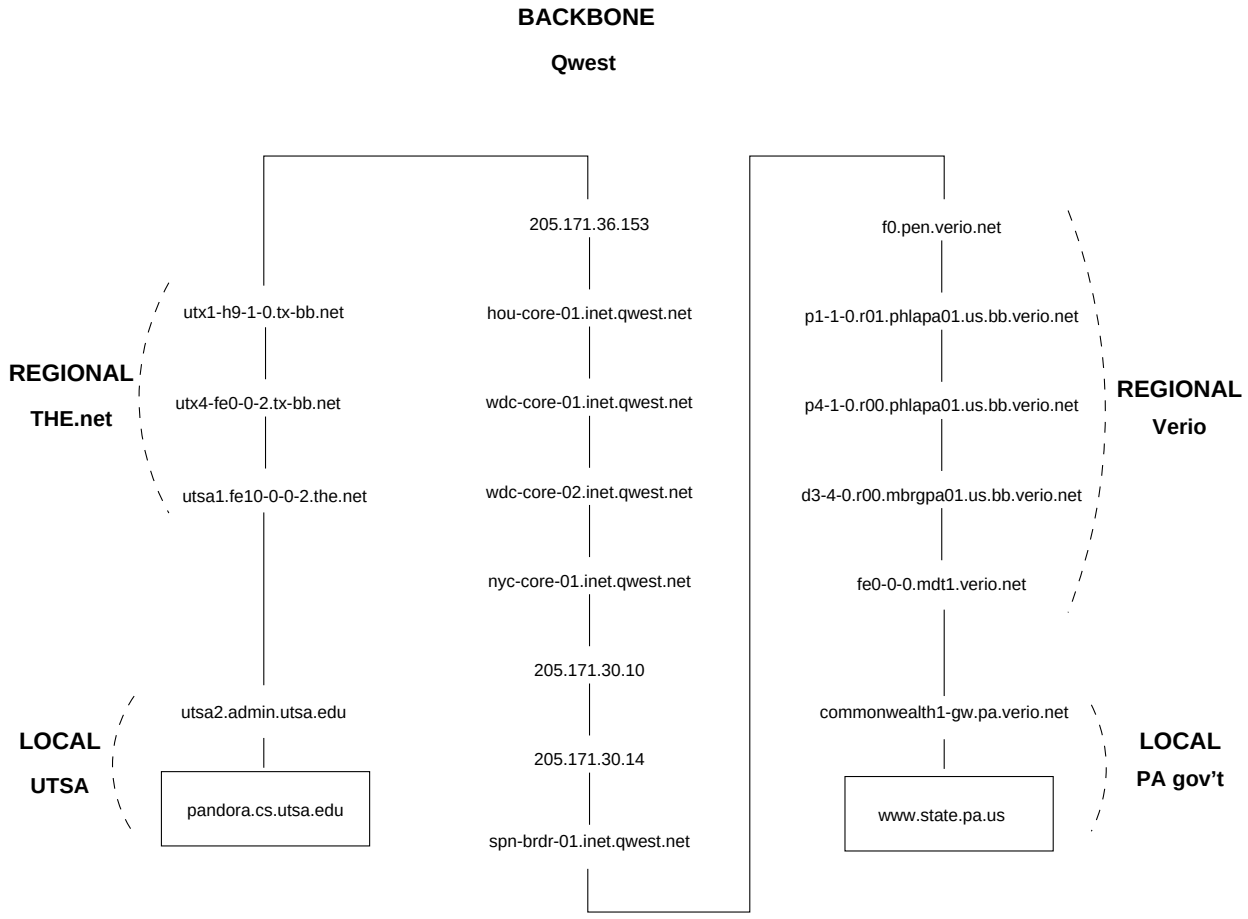

Figure 1.2: *Trace of Internet routers between a client at the University of Texas at San Antonio,* `pandora.cs.utsa.edu`*, and the official Pennsylvania state government Web server,* `www.state.pa.us`*. Data were obtained with the* `traceroute` *utility [33].*

from a client at the University of Texas at San Antonio to the official Web server for the state of Pennsylvania.

Data packets travel through the local area network gateway to the regional network. From the regional network, the packets enter Internet backbones. Although backbone networks have the highest bandwidth, the gateway routers between regional networks and backbones and routers at the backbone exchange points often exhibit the greatest congestion and delay. For this reason, communication within a regional network is typically much faster than communication that traverses an Internet backbone [30].

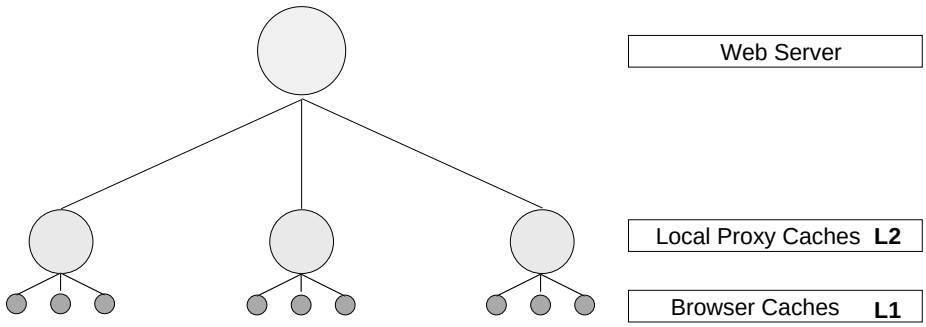

Figure 1.3: *Web browser caches (L1) and local proxy caches (L2).*

1.2 Caching on the World Wide Web

Traditionally, a cache is defined as a fast, temporary store for commonly used items. For example, high speed memory on microprocessor chips cache data from main memory; main memory units cache sections of disk files; and local disks cache documents from the network file server. Caching succeeds in these examples because 1) each cache level is progressively faster, and 2) the data pattern exhibits locality of reference. Request streams that exhibit locality of reference frequently re-access the same or nearby data within a short time interval. Locality in the request stream allows caching algorithms to predict future data references and move that data into faster cache storage during periods when the CPU is busy. By overlapping computation and data transfer, caching hides the latency of data transfer and improves overall system performance. Viability of caching depends upon both locality of reference and a speed advantage for the cache compared to the primary store.

1.2.1 Proxy caches

Caching on the World Wide Web exists at several levels. At the lower levels, Web browsers (L1) maintain a private cache for the user and local proxy servers (L2) share Web documents across users, as illustrated in Figure 1.3. When a user requests a file, the browser first checks its cache on the user machine. If the browser cache does not contain a copy of the requested document, the browser sends the request to the proxy server on the local network. If the proxy server has cached the document, it returns the cached copy, thus avoiding the long delay often associated with remote network transfers.

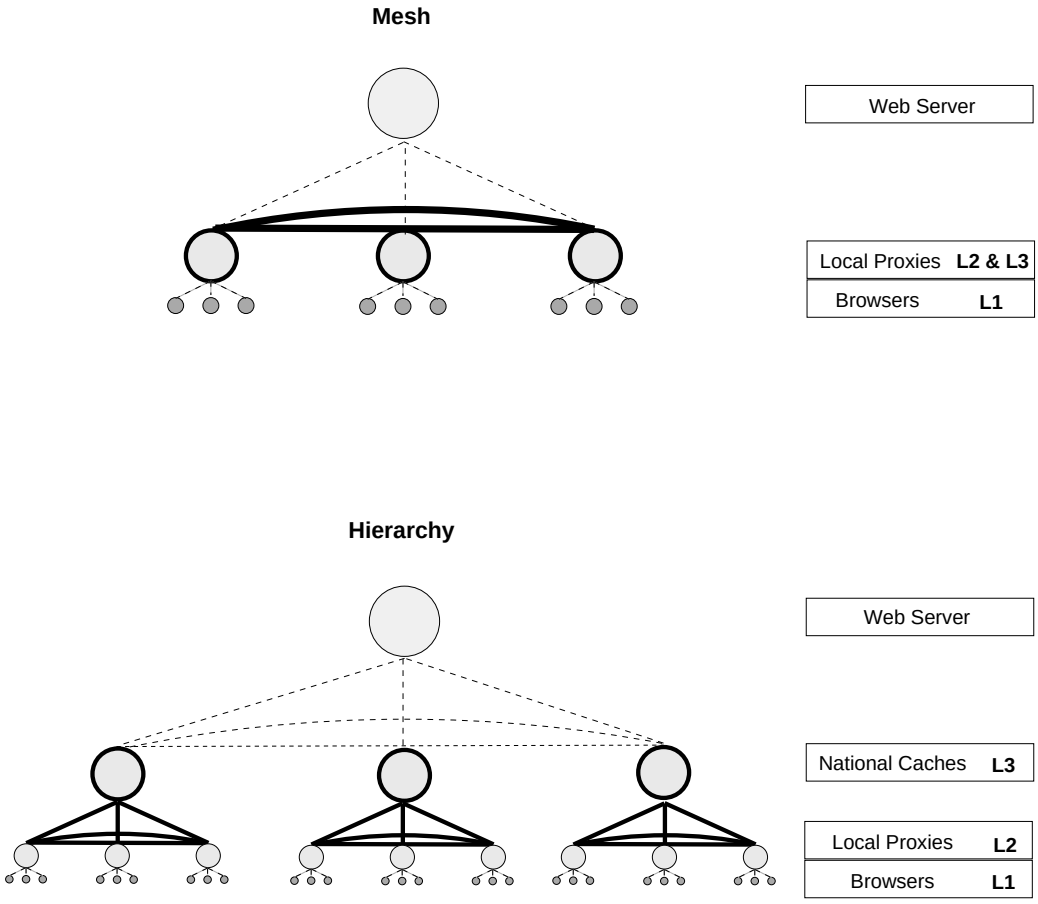

Figure 1.4: *Cooperative proxy caching (L3) with a mesh organization (top) and hierarchical organization (bottom). Bold lines indicate L3 cache communication.*

1.2.2 Cooperative Web caching

Cooperative Web caching (L3) is a mechanism for sharing documents between local proxy caches. If a cache miss occurs at the local proxy server, that proxy may forward the request to another cache instead of to the origin Web server. Cooperating proxies may be organized as a flat mesh or as a cache hierarchy, as shown in Figure 1.4. In a flat mesh, the L3 cache misses are handled by local proxies which directly retrieve the object from the Web server. In a cache hierarchy, the L3 cache misses are handled by parent caches which retrieve the object from the Web server and forward it to the local proxy. Current operational L3 caches use a hierarchical organization with a single layer of parent caches which usually serve as national caches [40][48][61].

Browser and local proxy server caches speed up data retrieval for the same reasons that memory and file

system caches succeed; specifically, 1) local network transfers are faster than remote transfers, and 2) Web documents are often re-requested by the same user or by other users on the same local network. Although researchers agree that browser and local proxy caching improve performance [2][53], there is considerable debate about the fundamental structure and mechanisms for cooperative L3 Web caches [22].

Recently, Wolman, et al., questioned whether any form of large scale cooperative Web caching provides significant benefit over L2 proxy caching [68]. Their study finds the benefits of cooperative caching are due simply to the increase in number of users sharing Web documents. Further, performance improves only up to a certain population size. The authors suggest that large L2 proxies gain little by cooperating with other proxies and that small L2 proxies should cooperate only on a small scale. Consequently, they see no need for large scale proxy cooperation. Wolman's study begins to address the fundamental question in cooperative Web caching: what is the extent and nature of document sharing on the Web?

1.3 Issues in cooperative Web caching

1.3.1 L3 Cache sharing

Successful caching requires that there are multiple requests for the same set of objects, and that these objects are cachable. This locality of reference allows the cache to share documents across requests. Studies consistently find evidence for locality of reference in Web request streams. In server traces, typically 10% of documents receive over 70% of the requests [6][31]. Similar results hold for proxy cache traces [1][12]. Researchers suggest that caching exploits this popularity skew by sharing popular Web documents across users [1] [14]. The problem for cooperative Web caching is that it is unclear how much sharing occurs within a proxy community and whether enough sharing is left to make proxy cooperation viable. L3 cache hits are limited to L2 cache misses, therefore if most popular documents are cached by local proxy servers, there is little need for an L3 level.

1.3.2 Cachability and ideal hit rates

Analysis of Web sharing patterns raises the issue of document cachability. Not all Web objects are cachable, and cachability rules are constantly changing. For example, objects with server tags known as "cookies" were not cached in HTTP/1.0, the earlier Web protocol standard, but are cached in the newer HTTP/1.1. Currently, dynamic and time-sensitive objects such as query results and stock market quotes are not cached, although there is active research in this area. Another development is the extent to which Web advertisers are using techniques to forbid or hinder caching because their revenues depend upon the advertisement's hit count. An ideal hit rates assumes all documents are cachable and counts all duplicate requests as cache hits. Ideal hit rates are useful in cache analysis because they represent the upper bound of cache performance.

1.3.3 Zipf distributions and cache hit rate

Zipf's law is a power-law function that relates the frequency of an event, $R(i)$, to its popularity rank i [71]:

$$R(i) = \frac{\Omega}{i^{\alpha}}.$$

In Web caching, Zipf's law relates the number of requests for a document to its popularity and has been interpreted as a measure of potential document sharing and cache hit rate [3][12][19][29][43][49]. Recently researchers have begun to use Zipf distributions to model cache workloads, although such models have not yet been especially successful in predicting observed hit rates[12][67].

1.3.4 Cache speed differential

A problem with remote network caches is that, unlike other types of caches, they are not inherently faster than the next level of the hierarchy. Remote caches and servers both transfer documents across the Internet, and there is no guarantee the cache will have faster server hardware, higher bandwidth, or a better route to the client. When cache and server have similar resources and communication cost, cooperative Web caching actually increases response time due to cache overhead. If cooperative Web caching is to be viable, there must be a mechanism for selecting remote cache sites such that caches return requested documents faster, on the average, than origin Web servers.

1.3.5 Cache overhead

Assuming cooperative Web caching is viable for the ideal case, the problem shifts to issues of cache overhead. If a real cache is to be viable, the overhead must not exceed the average speedup. Because Web caches are distributed across the Internet, cache management involves network communication. By increasing network traffic, Web caches may slow down unrelated network traffic. Therefore cooperative Web cache designers must be cognizant of how overhead affects both user response time and network cross traffic. Distributed Web caches perform three basic functions: *discovery* (how users locate cached objects), *delivery* (how objects are delivered from the cache to the user), and *dissemination* (where to cache objects). To estimate cache overhead costs, designs should consider how their approach to each function interacts with Internet traffic and Web request patterns.

1.4 Dissertation objectives

The goals of this dissertation are to characterize available sharing between proxy caches, to develop methods that make a set of remote caches faster than remote Web servers, to analyze basic cooperation approaches, and to determine if cooperative Web caching is indeed viable. The work addresses four fundamental questions:

1. Is there enough sharing between Web proxy servers to make cooperation viable? (Chapter 2)
2. Can a set of remote caches offer faster response times than Web servers? (Chapter 3)
3. What are the basic features of cooperative cache designs that best match user sharing and Internet traffic patterns? (Chapter 4)
4. Will cooperative caching speed up response time? If so, what is the relationship between speedup and cache overhead, and what is the maximum overhead for which cooperation is still viable? (Chapter 5)

We analyze proxy traces to determine the extent and nature of potential sharing. To establish a cache speed differential, we empirically compare algorithms for selecting remote Web sites and demonstrate that, given a choice of sites, caching offers statistically faster response times. To determine the best design choices

for cooperative caching, we organize a taxonomy around fundamental design elements and use it to analyze the match between cooperation approaches and Web traffic characteristics. Web characteristics are based upon a literature review and upon our own Web server measurements. The final question consolidates the dissertation through an analysis that address the fundamental viability and usefulness of cooperative Web caching. We develop an expression for expected speedup in user response time parameterized by results from the sharing and site selection experiments. Using this expression, we compute expected speedup as a function of cooperation overhead and efficiency. This analysis provides an estimate of the upper bound on cache overhead beyond which cache cooperation harms rather than helps performance.

1.5 Dissertation contributions

1.5.1 Effect of repeat requests on Zipf parameters and ideal hit rates

Our work shows for the first time how repeat requests for uncachable documents inflate the ideal hit rate and introduce errors in previous estimates of Web sharing. We challenge established models of document popularity and user sharing as measured by the Zipf's law exponent, α. Previous Web cache studies report α's ranging from 0.64 to 1.0, with values typically between 0.8 and 0.9 [3][12][19][29][49][43][68]. These values have been widely accepted as representative of cache sharing and used in simulation and analytical studies. In contrast, Chapter 2 shows that large α's are the result of repeat requests from the same client for uncachable documents, and therefore *do not* reflect potential sharing between different clients. When repeat requests are removed, α drops to $0.2 - 0.3$. The lower α's are a truer indication of document sharing and ideal cache hit rates. Further, if changes to Web applications or protocols remove some cachability constraints, the associated repeat requests disappear because lower level browser caches or proxies store the document and clients do not need to repeat their requests. Under no circumstances do repeat requests reflect document sharing, and α's computed with them misrepresent potential cache sharing and hit rates.

1.5.2 Techniques for correcting cache traces

One important outcome of this dissertation is a set of corrections for unrecorded information in cache traces. This improves the quality of large-scale, publicly-available traces, making them more valuable to researchers and enhancing the quality of other Web caching studies. Available sharing for cooperative Web caches is difficult to measure because the data must contain simultaneous request streams from multiple, independent proxy caches. While data collection is technically straightforward, it raises significant privacy and access issues. Consequently, most Web caching research relies on either trace-driven simulation or models derived from proxy traces. Even isolated proxy traces present a data collection problem because most researchers do not have access to large cache sites or organization gateways. Traces from some large operational caches are available, but these do not allow researchers to specify the type of information that is collected. Thus most researchers must choose between custom monitoring of small, limited request streams and incomplete data from large, diverse request streams.

National caches in Europe, Asia, Australia, and the National Laboratory for Applied Network Research (NLANR) in the United States provide most large cache traces. All use hierarchical cache software descended from the Harvest project [16], the most popular variant being the publicly available Squid caching software [58]. NLANR traces in particular are often cited in Web caching research because these traces are publicly available, up-to-date, and include many diverse users [12][21][26][63]. Our work points out three errors common to previous studies using hierarchical traces and provides correction techniques. These errors and corrections are outlined below.

Repeat requests

Most duplicate requests in the NLANR proxy traces are repeat requests from the same client, of which 90% to 95% are cache misses. Cache studies have difficulty accurately determining cachability because standard trace formats do not record cache directives in the response headers. We show that these directives can be successfully inferred by looking for repeat cache misses in the trace.

Cache warmup

Trace-driven simulations make a second error if the simulation starts with a cold cache. For single day NLANR traces, starting with a cold simulation cache predicts false misses for approximately 20% of the requests. This leads to an interesting interplay between cachability and warmup errors. Errors are of similar magnitude but opposite direction, and ignoring both may predict hit rates that match measured hit rates and appear to validate the simulation model. In fact, such a cache model would be inaccurate and could predict erroneous results for other workloads.

Sibling hits

The third error stems from assuming the requests recorded in the parent cache trace reflect all potential sharing among the child caches. However, hierarchical caches allow their children to obtain documents from each other without informing the parent. These *sibling hits* between the child caches do not appear in the parent traces, and therefore have not been included in previous cache studies. In Chapter 5, we introduce an analytical approach that includes empirical measures of the sibling hits, and show sibling hits add approximately 4% to the parent's hit rate.

1.5.3 Evaluation of site selection methods

When objects are replicated or cached at multiple remote sites, caching will be beneficial if the client selects a cache site that offers faster response time than the original Web server. However, fluctuations in network congestion and server load make it difficult to predict response times, especially during busy times of day when network load is the heaviest. The dissertation compares client-based methods for site selection and explores factors that influence their effectiveness.

Our work empirically evaluates six selection algorithms. The study compares two statistical algorithms, one using median bandwidth and the other median latency, a dynamic probe algorithm, two hybrid algorithms, and random selection. The server pool includes a topologically dispersed set of United States state government web servers. Experiments were run on three clients in different cities and on different regional networks. The study examines the effects of time-of-day, client resources, and server proximity.

The results provide two important contributions. First, client-side selection algorithms far outperform random site selection, demonstrating that cooperative Web caching can provide a statistical speedup by offering clients a choice of remote sites. Second, our work conclusively establishes that dynamic network probing performs as well or better than the other algorithms under all conditions, and shows a hybrid probe-bandwidth algorithm can be used to ensure both high performance and low communication overhead. Hybrid algorithms offer a practical solution for cooperative cache design that scales without congesting network routers and slowing network cross-traffic.

1.5.4 Taxonomy

Numerous designs for cooperative Web caching have been proposed, with several accompanying performance evaluation studies [9][16][31][44][58][62][70]. However, it is difficult to compare competing designs because because they contain multiple, interacting features and performance depends upon the entire design. In this work, we organize proposed designs into a taxonomy based upon methods for document discovery, dissemination, and delivery. The taxonomy is useful for comparing caching projects because it separates basic design choices from implementation details. Further, it helps focus on factors that may not be measured in simulation or performance studies. We analyze how alternative approaches in the taxonomy match characteristics of the Web. Although our analysis is qualitative, it serves as a guideline for basic design decisions.

1.5.5 Viability of cooperative Web caching

A major contribution of this work is an analysis that estimates cache speedup for various metadata propagation efficiencies, and suggests an upper bound for cooperative Web cache performance. Parameters in the analysis are determined from the cache sharing study and site selection experiment. The analysis directly addresses whether cooperative Web caching can speed up user response times and establishes bounds on cache overhead.

1.5.6 Design

Guided by the taxonomy analysis, we describe a cooperative Web caching system designed to reduce both response time and network congestion. Our design organizes proxy servers into a flat mesh, which incurs much lower delivery cost on wide-area networks than do hierarchical organizations [62]. Flat organizations have the problem of discovering where objects are cached. Our answer is to use local metadata directories that store information about data in other caches, and to propagate the metadata during quiescent network periods, with the option of piggybacking small updates onto object transfers. When a user requests an object, the proxy server looks in its local metadata directory to determine where the object is cached, then uses dynamic probing to select the remote site with the fastest expected response time.

1.5.7 Standard terms and metrics

Web caching is a relatively new area and suffers from lack of standard definitions. For example, consider the case where a Web cache returns its cached copy after exchanging short verification messages with the original Web server. Some authors consider this a cache miss because the cache connected to the original server [67], while others consider it a cache hit because the cache did not have to re-fetch the object [48]. Moreover, some studies do not include these requests in their analysis [14] or their definitions are unclear [12]. Differences also exist among authors with regard to the definitions of cache request, response time, and response latency. Throughout this work, we attempt to carefully define terms, propose standard comparison metrics, and offer rationale for using these terms and metrics. Such standardization would make the comparison of results from different studies easier and more accurate.

1.5.8 `httpget` and `tcping`

Utilities `httpget` and `tcping` are general-purpose measurement tools developed during the course of this work. These tools are applicable to other projects inside and outside the area of Web caching. The `httpget` tool measures components of Web download time, including DNS lookup, connection establishment, latency to the arrival of the first packet, remaining packet delivery time, and delivery bandwidth. It displays HTTP request and reply headers, and optionally saves the returned Web document. This utility is

HTTP/1.1 compliant, allowing users to attach cookies and ETAGs to the request.

The `tcping` tool measures network round trip latency by sending a TCP SYN header to an arbitrary port on the server and waiting for the returned RST. `tcping` is similar to the standard `ping` program in that it measures network latency. However, `ping` uses ICMP echo messages, therefore must execute with root privilege while `tcping` runs under user permissions. Consequently, `tcping` can be adapted for a user-level library function and called without the overhead of using `system()`. It can also be adapted to send concurrent probes to multiple hosts without threads or forks.

Both utilities are publicly available from the Web site given in Appendix C. Appendix C also contains the source code listings and a `Makefile` for compiling on either Solaris or Linux.

1.6 Publications

Portions of this dissertation appear in four papers:

- "Taxonomy and Design Analysis for Distributed Web Caching" [22]. This paper describes the taxonomy for cooperative Web caching and design analysis from Chapter 4 and our proposed cache design from Appendix A. The paper appears in the *Proceedings of the 32nd IEEE Hawaii International Conference on System Sciences (HICSS'99).*

- "An Empirical Evaluation of Client-Side Selection Algorithms" [23]. In this paper, we report the site selection experiment in Chapter 3 and explain how it applies to other elements of the Internet infrastructure. The paper is published in the *Proceeding of IEEE Infocom'2000.*

- "Uncacheable Documents and Cold Starts in Web Proxy Cache Simulations: How Two Wrongs Appear Right" [25]. The work described in this paper includes our analysis uncovering the effect of repeat requests on document sharing models (Chapter 2) and reports the interaction between cache warmup and repeat request errors.

- "A Viability Analysis of Cooperative Proxy Caching" [24]. This paper is based upon the viability analysis in Chapter 5 and contains the conclusions discussed in Chapter 6.

Chapter 2

Characterization of Sharing

2.1 Introduction

This chapter analyzes L3 cache traces from the National Laboratory for Applied Network Research (NLANR) global cache hierarchy to determine the extent and distribution of sharing across proxies. NLANR caches serve as parent caches to a set of L2 proxies, and therefore provide a real measurement of L3 sharing. However, there are three problems with using these traces to estimate potential proxy sharing: 1) repeat requests, 2) cache warmup, and 3) sibling hits. The analysis in this chapter addresses repeat requests and cache warmup effects. A correction for sibling hits is developed in Chapter 5 for converting L3 parent hit rates to total L3 hit rates.

Cache viability depends upon the average sharing rate across users and upon how that sharing is distributed. Studies consistently find that most Web requests are for a small fraction of documents, and that the distribution of requests approximately fits a power-law function known as the Zipf distribution [3][12][19] [29][49][43][68]. This popularity skew serves as the main argument to support Web caching: if most requests are for a small fraction of documents, then caching these popular documents near users would eliminate much remote traffic and server load.

Let us momentarily agree with this argument. The viability of *cooperative* caching then rests upon the distribution of document popularity across user groups. If all user groups request the same small set of documents, then local L2 proxies will handle most requests and there is little to be gained from proxy cooperation. Cooperation is only viable if a sufficiently large fraction of documents are shared across proxies.

2.1.1 Repeat requests, Zipf distributions, and ideal hit rate

In the course of this work, we discovered a flaw in the viability argument. Using request distributions to predict sharing across users overlooks the fact that, if a document is uncachable, then a single user may issue many repeat requests for the same document. Clearly repeat requests from the same user do not reflect potential sharing across users. We find repeat requests dominate document popularity distributions. The Zipf exponent, α, is bounded between 0.0 and 1.0, where larger α's indicate greater sharing. Removing repeat requests from the NLANR traces lowers α from 0.8 to 0.2 – 0.3 and significantly reduces predictions of potential cache sharing.

Repeat requests similarly inflate ideal L3 hit rates. Ideal hit rates assume all documents are cachable. However, if all documents were cachable, then browser caches would handle repeat requests and they would disappear from proxy and L3 traces. Past studies incorrectly treat repeat requests as ideal cache hits. Because under the assumption of ideal cachability these repeat requests would not be seen by L3 caches, the correct approach is to remove repeat requests from the trace, counting them as neither L3 hits nor misses.

2.1.2 Cache warmup

The NLANR traces span a single day. Computing hit rates based solely upon duplicate requests in the trace ignores documents already in the cache. This is known as a "cold cache" start. To correctly compute ideal hit rates, we must first warm up the cache. The analysis described in this chapter examines the effect of cache warmup on computed hit rates. Further, it shows how ignoring document cachability and cache warmup can generate errors of similar magnitude but opposite sign, resulting in an apparent match between predicted and measured hit rates. Such a match could lead to the acceptance of an inaccurate model and possibly result in misleading predictions.

2.2 Related work

Cao and Irani [14] found a large number of repeat requests in traces from DEC, the University of Virginia, and Boston University. Their data show users often re-access the same document, and that the fraction of

repeat requests peaks on a 24 hour cycle. Wolman et al. report that 40% of all requests to shared documents at the University of Washington are repeats by the same client [67], and Nishikawa et al. [49] observe that users tend to access the same document repeatedly at short intervals. Nishikawa et al. hypothesize that removing such requests from the trace might improve their simulation results.

Wolman et al. [68] collected traces that include response header information by installing custom monitoring software at the University of Washington's Internet gateway. The authors also obtained concurrent traces from proxies at Microsoft Corporation. Because they controlled data collection, the authors could avoid cachability and warmup errors; however, this approach is limited to sites that grant access. Wolman extrapolates the University of Washington trace to multiple sites by creating virtual proxies according to client IP address. In this study, all users were located in the same geographic area (Seattle) and either attended the same university or worked for the same company, limiting the diversity of the user population.

Breslau et al. [12] take the opposite approach, analyzing traces from six large proxy caches that include L3 caches at NLANR and Questnet in Australia, and L2 caches in Italy, Finland, and the United States. This approach ensures diversity and measures actual cache behavior, but allows no control over the logged information. For the NLANR trace, Breslau et al. include successful GET requests for both cachable and uncachable documents in the analysis. The authors report Zipf parameters of $\alpha = 0.64$ and $\Omega \approx 4000$. (Ω is the proportionality constant in the Zipf distribution and is approximately equal to the number of requests for the most popular item). As NLANR cache sites typically service less than 100 clients and the authors used four NLANR sites, we estimate the number of clients to be less than 400. If there were no repeat requests, Ω would not exceed the number of clients, yet in Breslau's data it is greater by a factor of ten. Breslau's study is the most comprehensive to date on the relationship of Zipf's law to Web caching; however, the inclusion of repeat requests adversely affects hit rate predictions that depend upon the Zipf parameters.

Duska et al. [21] examine inter-trace sharing across six proxy caches and report results as percentage of shared URLs rather than as Zipf distributions. Using their reported data, we determined Zipf parameters for the combined traces. Assuming no repeat requests, and no capacity or consistency misses, we computed least squares fit parameters of $\alpha = 0.24$ and $\Omega = 10$. These parameter values are far lower than those reported in other studies and match results reported here.

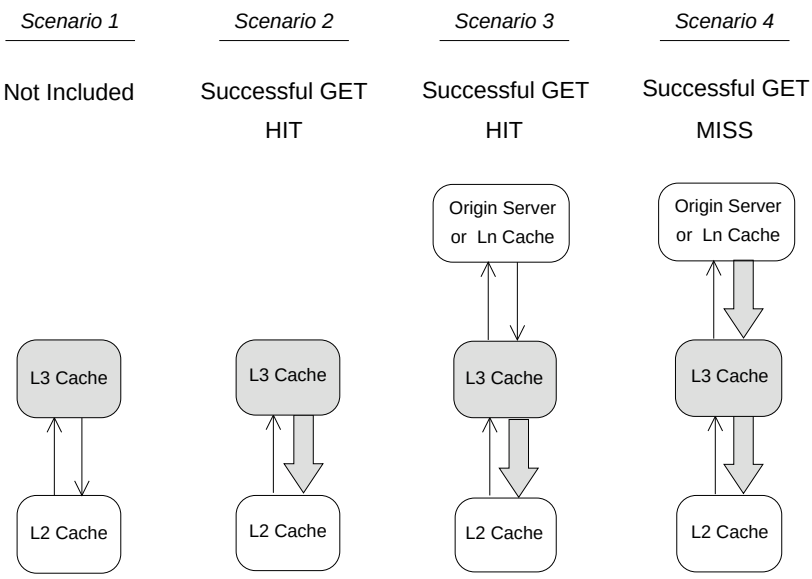

Figure 2.1: *Web cache scenarios illustrating the definition of hit rate. Narrow arrows are HTTP headers; wide arrows are document transfers. The left scenario is not a successful GET and is not included in the analysis. The middle scenarios are L3 cache hits, and the right scenario is an L3 cache miss.*

2.3 Issues in Web cache modeling

2.3.1 What is a cache hit?

Results from different Web caching studies are difficult to compare because there is no standard definition for cache hit rate. Some researchers report results based upon number of successful GET requests [6][12], while others base it upon all requests, including those with errors [68]. This distinction is critical because unsuccessful requests are common: on the average, we found successful GETs comprise only slightly over half of all requests in the NLANR traces. Likewise, there is no agreement on the definition of a proxy cache hit. Figure 2.1 illustrates the four possible scenarios listed below.

Scenario 1 Request is not a successful GET, so no document is returned.

Scenario 2 Proxy returns the cached document without validation.

Scenario 3 Proxy returns the cached document after exchanging HTTP validation headers with the origin server.

Scenario 4 Proxy retrieves the document from the origin server and forwards it to the client.

Figure 2.2: *One of the most requested documents in the NLANR traces: a* 1×1 *gray GIF image.*

Scenario 3 is neither a cache hit nor a miss in the traditional sense; however, in terms of both response time and resource usage, it is more comparable to a cache hit because HTTP headers are far smaller than most documents and file I/O is not involved. We therefore define hit rate based upon document transfer. Hereafter, the term "request" should be taken to mean a successful GET request, and cache hits are defined as in Figure 2.1.

2.3.2 Repeat requests and Web advertisers

We define a repeat request as a request from the same client for the same URL. Our analysis of the NLANR traces shows that 90% to 95% of repeat requests are cache misses. Web advertisers supply some of the more egregious examples of such requests. Figure 2.2 displays the document returned most often from the NLANR BO1 cache on Jan. 30, 2000: a 1×1 gray GIF background from a Web advertiser. The same gray GIF image was also the 2nd and 5th most frequently returned document. The proxy cache returned these three documents almost 20,000 times, *yet only 23 of the requests were cache hits.* Equally striking, these thousands of retrievals were for at most 26 clients. The most popular document was retrieved 14,210 times for only 4 clients, and no document was returned to over 16 clients. We noticed other advertisers use an identical gray GIF image. In a different trace, the cache fetched this image from RealMedia a total of 2730 times, returning it each time to the same client.

Req.	*Hits*	*Clients*	*Bytes*	*URL*
14210	0	4	409	`m.doubleclick.net/viewad/817-grey.gif`
4791	2	6	244	`ad.doubleclick.net/viewad/817-grey.gif`
1394	0	1	12821	`m.doubleclick.net/viewad/385809-family...`
939	0	1	2130	`m.doubleclick.net/viewad/28902-lycoshop...`
915	21	16	359	`messenger.netscape.com/images/pixel.gif`

Table 2.1: *Five most popular documents in BO1 trace for Jan. 30, 2000.*

Table 2.1 lists the five most popular documents in the Jan. 30 trace, together with the number of requests, cache hits, and clients, and shows a pattern of repeat misses that is typical of the NLANR traces. Throughout the traces, we observe repeat requests often access Web advertisement sites. Of the five most popular documents in the Jan. 30 trace, four are to a DoubleClick site and the fifth to Netscape.

It is interesting to examine why a small static image is uncachable. The log file indicates that cache control directives are attached to requests for two of the documents, but the log does not explain why the third document is uncachable. To test if response headers are responsible, we retrieved the documents with our `httpget` utility (Appendix C). All three origin servers attach `ETag` headers with entity tags that change every minute or two, despite the fact the document is unmodified. Entity tags are designed to assure cache consistency: the server attaches an entity tag to the document and the client includes this tag with subsequent requests. If the document is unmodified, the server should respond with an HTTP 304 Not Modified message instead of the object. Our tests, however, show the servers improperly resend the document and thereby generate cache misses. Even if origin servers respond correctly, it remains unclear why a static 1×1 GIF image should require consistency checking on every request.

2.3.3 Zipf distribution

A Zipf distribution relates number of requests for an object, R, to its popularity rank i:

$$R(i) = \frac{\Omega}{i^{\alpha}}.$$

The exponent α reflects the popularity skew, and the constant Ω approximates the number of requests for the most popular document ($i = 1$). Because α increases as popular documents receive a greater fraction

of requests, it has been interpreted as a measure of the potential sharing in a Web cache. An α near one suggests a strong potential for document sharing and a high cache hit rate.

Beginning with Glassman in 1994 [29], numerous studies report Web requests follow a Zipf or Zipf-like distribution [3][12][19][43][49] with α ranging from 0.64 to 1.0. However, all previous Zipf's law studies in Web caching that we are aware of include repeat requests. The analysis in Section 2.5.5 shows *Zipf's law behavior and large α's are caused by repeat requests for uncachable documents, not by potential sharing between clients.*

When repeat requests are removed, α drops to $0.2 - 0.3$. The lower α's are a truer indication of the potential cache sharing and hit rates. Further, if changes to Web applications or protocols remove some cachability constraints, the associated repeat requests will disappear because lower level browser caches or proxies will store the document and clients will not need to repeat their requests.

2.4 Methodology

2.4.1 Traces

We use publicly available traces [48] from the NLANR Boulder (BO1) and Urbana-Champaign (UC) caches for the 16 days from Jan. 27 to Feb. 11, 2000. NLANR caches are hierarchical L3 caches which service only L2 proxies. Approximately 90 proxies use BO1 as their parent cache and 60 use UC. Client IP addresses are randomized daily and are consistent within a trace but not between traces. Each trace spans one day and contains from 400,000 to 900,000 total requests for a combined total of 9 million requests for BO1 and 11 million for UC. Log files are in the Squid native format which records the cache action, allowing computation of actual hit rates.

2.4.2 What is cachable?

We determine cachability from explicitly logged information and from cache behavior. A request is considered uncachable if any of the following are true:

Cgi-bin (C)	URL contains “cgi-bin”.
Query (Q)	URL contains “?”.
Pragma (P)	Request header contains a no-cache pragma or other cache control directive (Squid code `TCP_CLIENT_REFRESH_MISS`).
Partial Content	Request is for partial content (Status 206).
Inferred (X)	None of the above apply, yet the document receives ≥ 2 requests and all are cache misses. Inferred uncachability is due to cache control directives in the response header and to less obvious mechanisms such as frequently changed entity tags.

2.4.3 Cachability models

Four different models are used to measure the effects of cachability and cache warming. All treat the initial request for a document as a cache miss. Duplicate requests for cachable documents count as cache hit, and duplicate requests for uncachable documents count as cache misses. Models assume infinite cache size, and, with the exception of the **Recorded** model, assume no consistency misses. Results show these assumptions introduce little error in the computed hit rates. The four cachability models are:

All_requests	All documents are considered cachable.
Cachable CQP	Cgi-bin, query, pragma, and requests for partial content are considered uncachable; all else is cachable.
Cachable CQP-X	In addition to CQP, any document that receives ≥ 2 requests but no cache hit is considered uncachable.
Recorded	Duplicate requests are assigned the cache action recorded in the log file. In the warmup experiment, the **Recorded** hit rate will eventually equal the measured hit rate if no documents are replaced by the physical cache.

	B01	*UC*
Traces	16	16
Length of each trace	1 day	1 day
Documents per trace	198363	290923
Total requests per trace	563956	691770
Successful GET requests per trace	276481 (49%)	397363 (57%)
File size, median		
Cachable	3.0 KB	3.3 KB
Uncachable	2.6 KB	2.8 KB
The following fractions are based on successful GET requests:		
Cachable	**0.72**	**0.76**
Initial	0.65	0.70
Duplicate	0.08	0.07
Uncachable	**0.28**	**0.24**
Initial	0.06	0.04
Duplicate (different clients)	0.02	0.01
Repeat (same client)	0.20	0.18

Table 2.2: *Trace statistics for NLANR caches, Jan 27 - Feb 11, 2000. Fractions are based on successful GET requests. Values are averages unless otherwise noted.*

2.5 Results

Table 2.2 contains per-trace statistics for the two cache sites, and shows the fraction of successful GETs that are initial requests for an object, duplicate requests from different clients, and repeat requests from the same client. Approximately half of all requests are successful GET requests; the remaining requests are primarily GET errors or cache consistency validations. At BO1, 28% of the successful GET requests are for uncachable documents. Of these, 6% are initial document requests, 2% are duplicate requests for the same document from different clients, and 20% are repeat requests for the same document from the same client. The ratio of initial requests to repeat requests shows that, on the average, *clients request an uncachable document not once, but 4.3 times.* The proxy cache must re-fetch the document in over 90% of these cases. The UC cache averages a slightly higher ratio of 5.5 requests and over 5 fetches per client for each uncachable document.

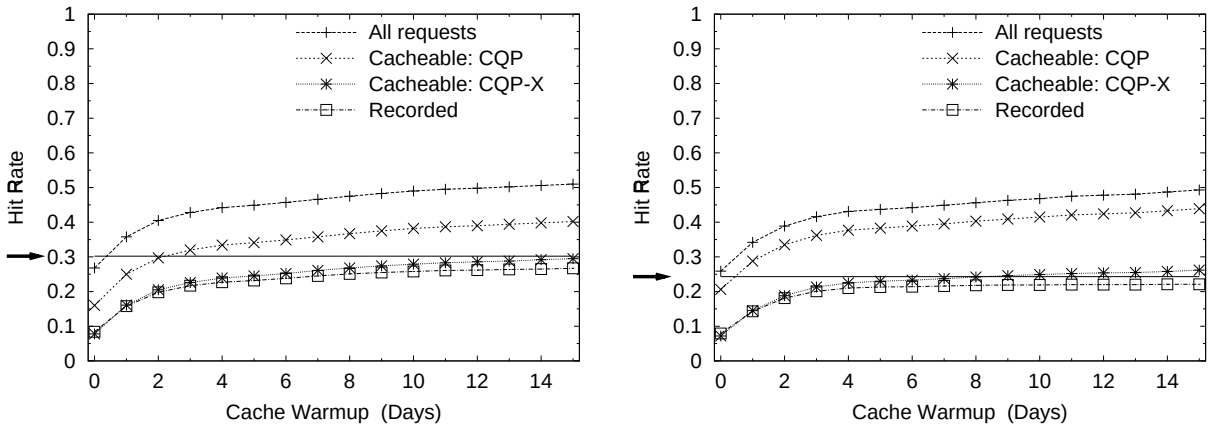

Figure 2.3: *Effect of cache warmup on computed hit rates at BO1 (left) and UC (right), Feb 11, 2000. Arrows indicate the measured hit rate as recorded in the trace.*

	B01		**UC**	
	Cold	*Warm*	*Cold*	*Warm*
Computed hit rates				
All_requests	0.27	0.51	0.26	0.49
Cachable CQP	0.16	0.40	0.21	0.44
Cachable CQP-X	0.08	0.30	0.07	0.26
Recorded	0.08	0.27	0.08	0.22
Measured hit rate	0.30		0.24	

Table 2.3: *Hit rates for different cachability constraints with and without cache warmup. Hit rates measured for Feb 11 trace, with cache warmup from Jan 27 - Feb 10 traces (15 days).*

2.5.1 Cache warmup

The cache warmup experiment begins by assuming an empty cache. Hit rates are computed for the Feb. 11 trace using each cachability model. Next, documents from the preceding day's trace are added to the model cache, and the Feb. 11 hit rates are recomputed. This process continues until, at the final data point, the model cache contains all documents successfully returned by the proxy during the 15 days from Jan. 27 to Feb. 10. At each data point, only requests in the Feb. 11 trace are used to compute hit rates. Requests from earlier traces serve merely to warm the cache.

Figure 2.3 and Table 2.3 present cache warmup results for the four cachability models. Measured hit

rates of 0.30 for BO1 and 0.24 for UC are denoted by solid lines in the graphs. As Figure 2.3 illustrates, the **Recorded** hit rates closely approach the measured rates after an adequate warmup period. For UC, warmup requires around 4 days, while BO1 requires 8 to 10 days. After this point the recorded hit rates level off and additional warmup is unnecessary. Actual warmup length may differ for other days or other caches; however, these results indicate it is on the order of several days and that simulations driven by single day traces cannot ignore it.

2.5.2 Hit rates for different cachability assumptions

For the ideal cachability model, the predicted hit rates greatly overestimate the actual hit rates. With a warm cache, the All_Requests model predicts hit rates of 0.51 for BO1 and 0.49 for UC, as compared to the measured rates of 0.30 and 0.24. This overestimate is consistent: assuming all documents are cachability falsely predicts cache hits for approximately 20% of the requests throughout the warmup period.

Large errors for the **CQP** results demonstrate that request data is insufficient for modeling cachability. With a warm cache, the **CQP** model predicts hit rates of 0.40 for BO1 and 0.44 for UC. Thus, using explicit information in the log file on document cachability corrects only half the errors in BO1 hit rates and a fourth of the errors in UC hit rates. Accurate models of cache behavior must account for cache misses due to the actions of HTTP response headers, even though this information is not recorded in most standard log files.

As Figure 2.3 illustrates, the **CQP-X** hit rates closely overlap the **Recorded** hit rates throughout the warmup period, and, after adequate warmup, correctly predict the measured hit rate. The excellent agreement between **CQP-X** predictions and the measured hit rates establishes that inferring uncachability from repeat requests produces a correct model, even if log files do not contain response header information. Our approach makes publicly available traces more useful and reduces the need for collecting custom traces.

A more subtle but important point is the agreement between predicted and measured results indicates few cache misses occur are caused by documents replacement. If requested documents were replaced, the **CQP-X** model would predict cache hits while the recorded data would indicate cache misses.

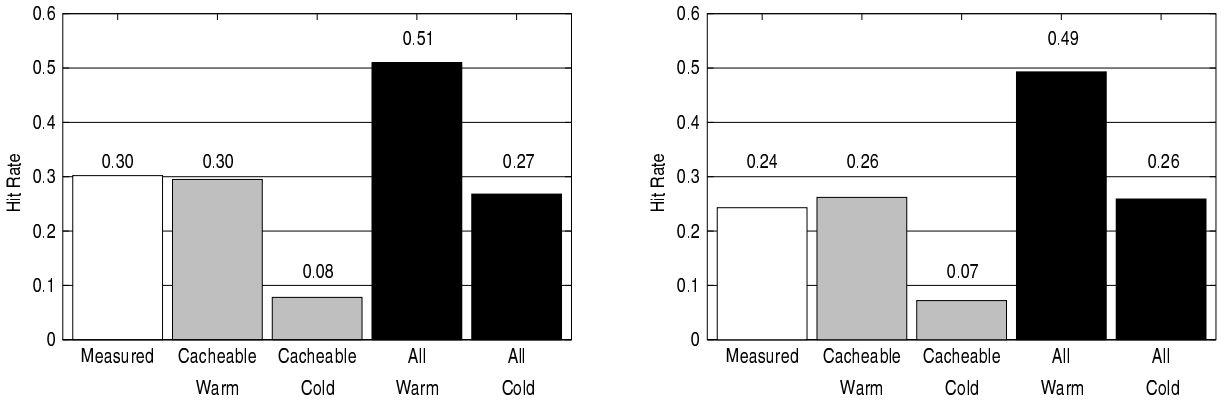

Figure 2.4: *Measured and calculated hit rates for BO1 (left) and UC (right).*

2.5.3 How cachability and warmup errors may cancel

Our analysis highlights an unexpected result: when a cache model makes the dual errors of starting with a cold cache and assuming all documents are cachable, the predicted hit rate approximately equals the measured hit rate. Figure 2.4 illustrates the separate and combined effects of these two errors. Correctly modeling uncachability but using a cold cache *underestimates* the hit rate by 22% for BO1 and 21% for UC. Warming the cache but treating all documents as cachable *overestimates* the hit rate by 17% and 25%. When both corrections are made, the computed hit rates closely match the measured rates. Unfortunately, the two errors are of similar magnitude and opposite sign, therefore when *neither* correction is made, the computed hit rates also appear to match the measured rates. This could lead researchers to believe that a cache model is correct, when in fact these two errors will not necessarily cancel when the workload is changed. These errors also contribute significantly to the wide variance of hit rate estimates reported in the literature [3][12][19][29][49].

2.5.4 HTTP cache control and static documents

Repeat misses occur when the document is uncachable, as in a cgi-bin or query response, or when caching is prevented by HTTP headers. Some headers prevent the cache from storing the object, while others force the cache to validate and possibly re-fetch a document before reusing it. Table 2.4 gives reasons for uncachability and the corresponding fraction of requests. Values are reported for the Feb. 11 traces and are

	B01	*UC*
Cachable	**0.75**	**0.76**
Uncachable	**0.25**	**0.24**
206: Partial Content	0.00	0.00
Cgi-bin (C)	0.01	0.00
Query (Q)	0.08	0.01
Request pragma or directive (P)	0.07	0.07
Response header (X)	0.10	0.16
Repeat requests	**0.17**	**0.17**
206, Cgi-bin, Query, or Pragma (CQP)	0.09	0.04
Response header (X)	0.08	0.12

Table 2.4: *Attributing causes of document uncachability. Values are the fraction of successful GET requests in NLANR traces, Feb 11, 2000.*

representative of the other traces in the study. The data show requests for partial content and cgi-bin documents are negligible. Uncachability is primarily due to HTTP header cache control directives, which appears in around 17% of the requests. Cache control headers are attached primarily to static objects rather than to queries or cgi-bin documents. Recalling the example gray GIF image, it appears Web advertisers are using HTTP cache control mechanisms to prevent caches from storing and sharing static documents.

2.5.5 Repeat requests and Zipf parameters

Figure 2.5 shows log-log Zipf plots for BO1 trace of Feb. 7, 2000, and is representative of Zipf plots for the other traces. One plot includes all requests and the other shows the Zipf distribution without repeat requests. Values for α and Ω are determined by a nonlinear least-squares (NLLS) Marquardt-Levenberg algorithm, and the best-fit function is superimposed upon the trace data. Because requests are discrete and the data are dominated by points with 1 or 2 requests, each point in the NLLS is weighted by $1/d$, where d is the number of documents with the same request count. This is equivalent to using the average document rank for each request value and avoids an artificial flattening of the fitted curve due to the stair-step nature of the data.

Table 2.5 reports medians and ranges of Zipf parameters for the 16 daily traces for each cache. Two features stand out: repeat requests greatly inflate both α and Ω, and repeat requests create large variations in the measured Zipf parameters. With repeat requests, the NLANR α ranges from 0.66 to 1.01 for BO1, with

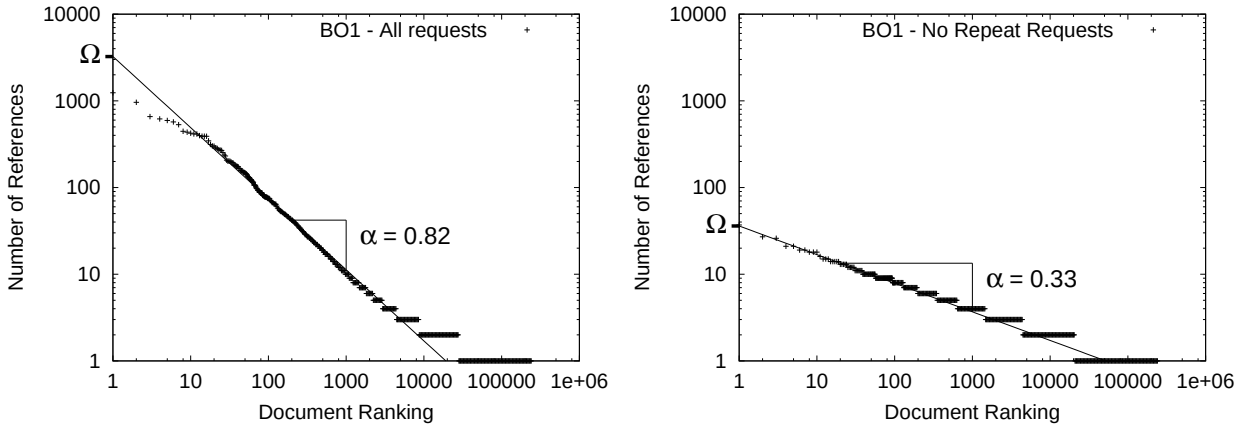

Figure 2.5: *Zipf distributions with repeat requests (left), and without (right) for BO1, Feb 7, 2000.*

	BO1		**UC**	
	α *median*	α *range*	α *median*	α *range*
All requests	0.83	0.66 - 1.01	0.80	0.70 - 0.92
No repeat requests	0.33	0.29 - 0.46	0.22	0.19 - 0.31
Mon - Fri	0.31	0.29 - 0.35	0.21	0.19 - 0.25
Sat, Sun	0.42	0.38 - 0.46	0.29	0.27 - 0.31
	Ω *median*	Ω *range*	Ω *median*	Ω *range*
All requests	3120	1250 - 8700	3500	1450 - 6240
No repeat requests	33	31 - 38	21	18 - 23

Table 2.5: *Zipf parameters for NLANR traces, Jan 27 to Feb 11, 2000.*

a median of 0.83, and ranges from 0.70 to 0.92 for UC, with a median of 0.80. The values for α in these 32 traces overlap the range of previously reported α's for NLANR traces [3][12][19][29][43][49]. The high variability across traces is due to random factors in repeat requests and explains the spread in previously reported values. When repeat requests are removed from the traces, the median α falls dramatically to 0.33 for BO1 and 0.22 for UC. Moreover, the ranges tighten to where the α for BO1 is significantly greater than the α for UC, indicating BO1 clients share more documents than do UC clients.

Eliminating the noise of repeat requests also uncovers an interesting sharing pattern in weekdays *vs.* weekends: α's for M-F are clearly smaller than the weekend α's, suggesting there is less diversity during

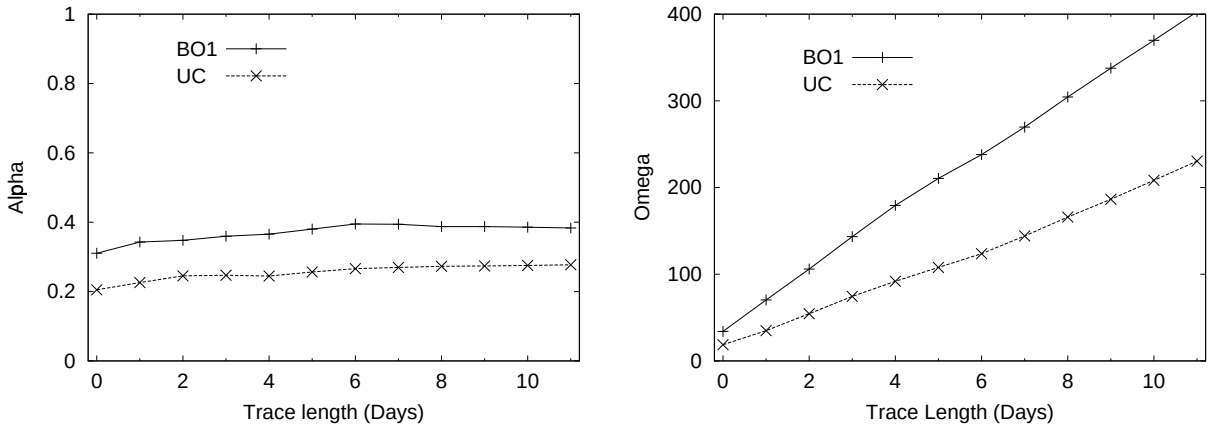

Figure 2.6: *Effect of trace length on Zipf parameters α (left) and Ω (right). Repeat requests were removed from the traces.*

weekends than during the week.

Zipf parameters computed without repeat requests offer better insights into the underlying data. For example, BO1 traces contain around 90 clients, and UC traces around 65 clients. An Ω of 3120 has little meaning because it is the sum of pathological events. In contrast, the value of Ω computed without repeat request indicates the most popular document is typically shared by 33 different clients, or roughly one-third of BO1's population. Likewise, the $\Omega = 21$ for UC shows its most popular document is shared by about one-third of its 65 clients.

We demonstrate how Zipf parameters vary with trace length by iteratively merging a preceding day's trace file and computing α and Ω. Repeat requests are removed from all traces. Results in Figure 2.6 show α remains fairly constant as the trace length increases, while Ω grows in a linear fashion with slope of approximately 36 for BO1 and 18 for UC. An intuitive interpretation lies in the observation that combining two Zipf distributions with the same α and Ω produces a Zipf distribution with the same α, but with an Ω twice as large. Because Zipf parameters do not vary much from day to day, adding a day's trace has little effect on α but adds a constant amount to Ω.

2.6 Sharing summary and discussion

Exploring repeat requests and cache warmup effects uncovers several interesting results with implications for cooperative cache hit rates. The most important result is the observation that repeat requests have distorted conclusions about the popularity of Web documents, and have led researchers to previously overestimate the extent of document sharing and potential cache success. This error manifests both in the ideal hit rates and in Zipf parameters, which are beginning to be used in analytical and simulation models for Web cache studies. It is critical to correct these errors before the results of such studies become ingrained in the literature. Repeat requests, cache warmup, and other results are summarized below.

- **Repeat requests artificially inflate ideal cache hit rates.**

An ideal cache assumes all documents are cachable and treats repeat requests as cache hits. Many researchers use the ideal cache hit rate as an upper bound on proxy sharing and the Web cooperation hit rate. The difference between ideal and actual hit rates is primarily due to repeat requests because these constitute a substantial fraction of the requests and almost all are cache misses. In the traces we examined, the percentage of repeat requests is 17%, compared to a difference of 21% to 26% between ideal and measured hit rates. *However, if all documents were cachable, most repeat requests would disappear because of caching by the Web browser and L2 proxies, therefore an ideal hit rate computed using repeat requests is meaningless.*

- **Repeat requests artificially inflate Zipf parameters.**

Zipf parameters are inflated by repeat requests for the same reasons as the ideal hit rate. Traces that include such requests do not accurately portray the potential proxy cache sharing between clients or between requests. After removing repeat requests, the Zipf exponent, α, drops from around 0.8 to 0.2 for the UC cache and to 0.3 for the BO1 cache. The constant Ω drops from around 3500 to near 30. These corrected parameters offer precise interpretations: α reflects the duplication in requests from different clients, and Ω reflects the number of clients that share the most popular document. Parameter variability also drops sharply,

exposing a difference in α between the UC and BO1 cache sites, and between weekdays and weekends. With repeat requests, these differences are hidden by noise in the data.

- **Trace analysis must adequately warmup the cache.**

We found 4 days was sufficient for one cache, but the other cache required over a week. With 1 day traces, a cold cache model predicts false misses for around 20% of the requests.

- **Trace analysis must correctly model cachability.**

Assuming all documents are cachable predicts false hits for 20% to 25% of the requests. Determining cachability from request header information while ignoring response header information predicts false hits for 10% to 20% of the requests. Using recorded request header information and inferring response header control directions correctly predict the measured results.

- **Uncachability can be inferred using repeat misses.**

If a trace does not contain explicit information about response headers, uncachability can be inferred from repeat misses. We would, of course, prefer that Squid and other cache software record response header uncachability in their standard log format.

- **Publicly available traces are essential, but should be corrected for cachability and cache warmup.**

NLANR offers Web researchers an invaluable resource by providing up-to-date, publicly available traces of a large operational proxy cache. These traces are not just a convenience, they are a necessity because most researchers do not have the access required to collect comparable traces. However, researchers should be aware that NLANR and most other Squid traces need to be corrected for response header and cache warmup effects. These two errors may cancel for the test workload, but that does not assure they will cancel for other workloads. Ignoring these pitfalls may therefore result in erroneous conclusions from the analysis, and may produce poor parameters for analytical or simulation models.

- **HTTP headers are being used and misused to prevent cache sharing of static documents**

In addition to cache control directives, some content providers are using HTTP validation mechanisms such as entity tags to prevent caches from sharing static documents, even across requests from the same client.

- **Can repeat requests be eliminated from proxy caching?**

Yes - Web browsers and proxy caches should notice when a document is not being cached, and send subsequent requests directly to the origin server. This would reduce the load on upper cache levels by nearly 20% and remove excess network traffic by eliminating useless document transfers through the cache hierarchy. These relatively minor modifications to current proxy caching software could substantially impact performance and yield traces that more clearly portray Web sharing.

Chapter 3

Evaluation of Site Selection Algorithms

3.1 Introduction

3.1.1 Site selection and Web caching

In traditional caching scenarios, accessing a cached copy is many times faster than accessing the original data. On wide area networks where the choice is between a remote server and a remote cache, there is no natural speed differential. One way of creating a speed differential for Web caches is to offer the client a choice of cache sites for the object. Assuming caches and servers are randomly located, the cache set statistically includes at least one site that is faster than the origin server. The problem lies in determining which cache site is faster as fluctuations in network congestion and server load make it difficult to predict response times. The purpose of these experiments is to 1) compare the performance of site selection algorithms, and 2) determine the attainable speed differential between a cache set and the origin Web server. That is, for the best site selection algorithm, what is the performance difference between the fastest Web cache and the origin Web server?

3.1.2 Other applications

Server selection algorithms and their corresponding performance estimates are applicable to areas other than Web caching. Distributed systems rely on replicated objects and services to improve performance and reliability. When objects are replicated on multiple servers, the client would like to select a server that offers fast response time. Estimators of site performance would enable multimedia providers to offer clients a

choice of content fidelity for different estimated transfer rates. For example, a client-side monitor could use performance estimators to negotiate a video fidelity such as image resolution in order to maintain a fixed frame rate for real-time display. In mobile networking applications, changes in response time for client-server connections depend upon current client location as well as server load and network cross traffic. Mobile clients would benefit from a dynamic method of selecting servers as location changes.

3.2 Selection criteria

Selection criteria, or performance estimators, fall into three classes: *static, statistical, and dynamic.* Static estimators are based upon hardware resources and configuration, such as number of hops, connection bandwidths, and server hardware. These estimators take into account resource capacity, but not availability or contention. Response time, however, depends upon both resource capacities and loads.

Statistical estimators are computed from past performance data such as latencies and bandwidths, therefore they reflect typical levels of contention for a server and the network connection. Statistical estimators are less reliable when the data exhibit high variability, as observed for Internet traffic and HTTP server loads [8][17]. When variability is high, more measurements are required to determine a reliable estimator of expected value. Even with a reliable estimator, the high variability produces large errors in performance predictions. For Web transfers, this problem is compounded because latency depends upon time-of-day, and effective bandwidth depends upon both time-of-day and file size [46]. Still, previous work suggests median latencies or bandwidths might be a viable method for server selection [4][55]. Our experiments include two statistical algorithms, one based upon median latency and the other upon median bandwidth.

Dynamic run-time estimators use small probes to detect current network and/or server conditions. Probes estimate current resource availability, but do not include all performance factors. For example, a `ping` probe measures network latency but does not measure server delay or the effect of dropped packets on TCP/IP mechanisms. Another problem arises if conditions fluctuate more rapidly than the time for the document transfer. In this case, the conditions measured by a dynamic probe will not extend over the lifetime of the transfer. Finally, probes can add run-time overhead, both in terms of traffic and elapsed time. Our experiment includes a dynamic probe that reflects network and TCP/IP stack conditions, but not HTTP server load.

Category	*Selection metrics*	*Example*	*Ref.*
Client-side	Geography, Hops	Push-caching	[31]
	RTT	ICP, NLANR	[48][66]
	BW	SPAND	[57]
	Random, Server load	Smart Client	[69]
	Prior response time	empirical study	[37]
	Hops, RTT, BW	simulation study	[15][18]
	Hops, RTT, Latency	empirical study	[55]
	RTT, BW, Latency	empirical study	*this work*
Server-side	Server load	HTTP Redirect	[4]
		IP address rewrite	[20]
DNS	Round-robin	RR-DNS	[13][38]
Routers	Router metric	IPv6 Anycast	[35][50]

Table 3.1: *Classification and examples of server selection methods. RTT is network round trip time, BW and latency are statistical estimators derived from historic data*

Site selection algorithms may rely on a combination of estimators. Two algorithms in the study are hybrids that combine a statistical bandwidth estimator with a dynamic network probe. We anchor the study with a sixth algorithm that uses random selection to choose the Web site.

3.3 Related work

Server selection methods fall into four categories: router, DNS, server-side and client-side methods (see Table 3.1). Our experiments focus on client-side methods in which the clients or their proxies select the servers. Client-side selection is appropriate when the group of servers is heterogeneous or widely dispersed across the network. Conversely, server-side selection methods focus on server clusters. Server clusters typically contain members with similar resources and a shared local network. In clusters, the primary concern is balancing request load across servers: when resources are equal and the load is balanced, a client receives similar response times from all servers. One popular approach for load distribution uses DNS aliasing [13][38]. A site is assigned multiple IP records in the local DNS table. Upon receiving a translation request, the DNS Bind program selects from among these records in a round robin fashion (DNS-RR). While

simple, DNS-RR offers only crude load balancing and is often combined with a server-side mechanism. In server-side mechanisms, clients send requests to a dispatching module that tracks current load conditions and assigns servers accordingly. The dispatcher may direct a client to the chosen server via an HTTP Redirect [4]. Alternatively, the dispatcher may change IP addresses on all incoming packets to route them to the appropriate servers [20]. This algorithm requires the dispatcher keep track of all the site's TCP streams, earning it the name TCP router.

The last category of server selection methods relies on network routers. IPv6 allows an Anycast address to be assigned to one or more network interfaces. Routers choose among the interfaces by deciding which is "nearest", where nearest is defined by a router metric such as hop count. The problem with Anycast lies in determining a router metric that effectively predicts user response time. Several studies have shown that, in general, hop count shows little correlation with bandwidth or response time [18][46][55]. Yoshikawa, et al., argue that "in many cases the client, rather than the server, is the right place to implement transparent access to network services" [69]. They propose a general implementation framework for client-side selection known as the Smart Client. This is similar to work by Bhattacharjee, et al. on application-layer Anycasting [11]. Both systems are flexible enough to incorporate different selection metrics, as are several distributed Web caching designs [22][58][62]. Consequently, all are complementary to this experimental work and could benefit from the results.

The studies most similar to this work are client-side selection studies by Carter and Crovella [15][18], Seshan, et al. [57], and Sayal, et al. [55]. Carter and Crovella use trace-driven simulation to compare HTTP transfer times for selection algorithms based upon geographical distance, hop count, `ping` probes, and an available bandwidth probe called `cprobe`. They find the fastest transfer times are obtained for the dynamic `ping` algorithm and for an algorithm that combines `ping` with `cprobe` data. Carter and Crovella conclude `cprobe` is unsuitable as a dynamic probe because of its high overhead, both in terms of time and network traffic. However, `cprobe` provides information similar to bandwidth estimators derived from historic data, so results in [15] augment findings for statistical bandwidth algorithms. In the SPAND project, Seshan, et al., passively collect bandwidth data for transfers from a server to all clients on the local LAN [57]. They show observed bandwidths are reasonably stable; that is, 90% of the observed bandwidths are within a

factor of 4 of the predicted bandwidth. However, they do not report total time, nor do they compare their statistical bandwidth predictor to other metrics. Sayal, et al., directly compare performance of different selection metrics, but they use HTTP "request latency" as a cost function instead of response time [55]. Consequently, their results indicate only that past request latency is the best predictor of current request latency, and do not necessarily extrapolate to total response time. Further, [15] and [55] report metrics using means, and [57] uses bandwidth means and standard deviations for the selection criteria.

This work expands upon the related studies by presenting results for different times-of-day and for clients on different regional networks, and by examining the relative costs of connection establishment, read latency, and bandwidth. We also describe the distributions of connection times, latencies, bandwidths, and total response time, and explain why median and semi-interquartile range (SIQR) are more appropriate statistics than mean and standard deviation for both statistical estimators and the performance metric.

3.4 Server selection algorithms

3.4.1 Response time components

From the client's perspective, user response time T is the sum of DNS lookup, connection establishment, read latency, and remaining read time:

$$T = T_{DNS} + T_{Connect} + T_{Latency} + T_{Remaining} \tag{3.1}$$

where T_{DNS} is the DNS lookup time, $T_{Connect}$ is the time to establish a TCP/IP connection, $T_{Latency}$ is the time from sending the request to receiving the first packet of the reply, and $T_{Remaining}$ is the time to receive the remaining reply packets. DNS lookup exercises the DNS hierarchy and is independent of server load. It can be avoided if the client knows the server's IP address through caching or an independent metadata transfer mechanism [22][62]. For these reasons we ignore the DNS term. Connection time depends upon network latency and the wait in the server's application queue. Read latency includes the server delay incurred retrieving the object from disk, plus network delays due to transmission latency of the object's first data packet. The remaining read time depends upon available bandwidth for the client-server connection and upon server load. Results in Section 3.6 show $T_{Remaining}$ dominates response time for moderate file sizes.

However, all four components depend upon network conditions and, as we observed, can cause pathological behavior when packets are dropped by congested network routers.

3.4.2 Algorithms

The six selection algorithms in this study are based upon different response time components. The best performing algorithm will be the one which is based upon the most dominant component(s) after accounting for selection overhead. The algorithms are defined as follows:

Random (static): Select a server randomly.

Latency (statistical): Select the server with lowest median latency, $T_{Latency}$, in prior transfers.

BW (statistical): Select the server with highest median bandwidth in prior transfers:

$$BW = \frac{bytes}{T_{Latency} + T_{Remaining}}.$$

Probe (dynamic): Send dynamic probes to all servers and select the first to reply. Immediately request the object from that server without waiting for replies from other probes.

ProbeBW (hybrid): Consider only n servers with the fastest median bandwidths. Send dynamic probes to these servers and select the first to reply. Immediately request the object from that server without waiting for replies from other probes. ($n = 3$ in our experiments.)

ProbeBW2 (hybrid): Send a dynamic probe to all servers. Upon receiving the first probe reply, delay for half the reply time, then select the server with the highest median bandwidth among those who have replied.

3.5 Methodology

3.5.1 Performance metric

Client-side selection methods are designed for a set of heterogeneous, topologically-dispersed servers, whose response times depend upon both server and network effects. Even for homogeneous server sets with well-balanced load, response times can differ significantly because network routes between the client and the servers have different bandwidths and congestion patterns. Server load does not reflect route differences, therefore it is not an appropriate metric for client-side methods, although it can be used to compare server-side selection methods [4][20]. Conversely, high bandwidth connections do not guarantee fast response times if the server has a long queue or slow file system. The appropriate metric for client-side selection algorithms is user response time because it reflects both server and network effects. Unfortunately, response times cannot be compared directly if objects are of different sizes. Other researchers approach this problem in several ways. Karaul, et al., [37] measure response times but restrict their server set to 10 mirror sites with identical objects. In the SPAND project [57], the performance evaluation includes data for many HTTP servers and different object sizes, but uses bandwidth rather than response time as the metric. Sayal, et al., [55], also use an unrestricted server set, but compare selection algorithms on the basis of latency and disregard bandwidth. Rather than restricting either the server set or the metric, our solution is to normalize response time.

3.5.2 File size normalization

Let T be the measured response time for retrieving an object of size N. Let $\widehat{T}$ be the computed normalized time to retrieve the object if it were of size $\widehat{N}$. To normalize the response time, we measure T and its components $T_{Connect}$, $T_{Latency}$, and $T_{Remaining}$. We also record the total number of bytes, N, and number of bytes in the first read, $N_{Latency}$. Assuming the bit rate for remaining reads is independent of file size, the response time normalized to reference size $\widehat{N}$ is:

$$\widehat{T} \;=\; T_{Connect} + T_{Latency} + T_{Remaining}\left(\frac{\widehat{N} - N_{Latency}}{N - N_{Latency}}\right) \tag{3.2}$$

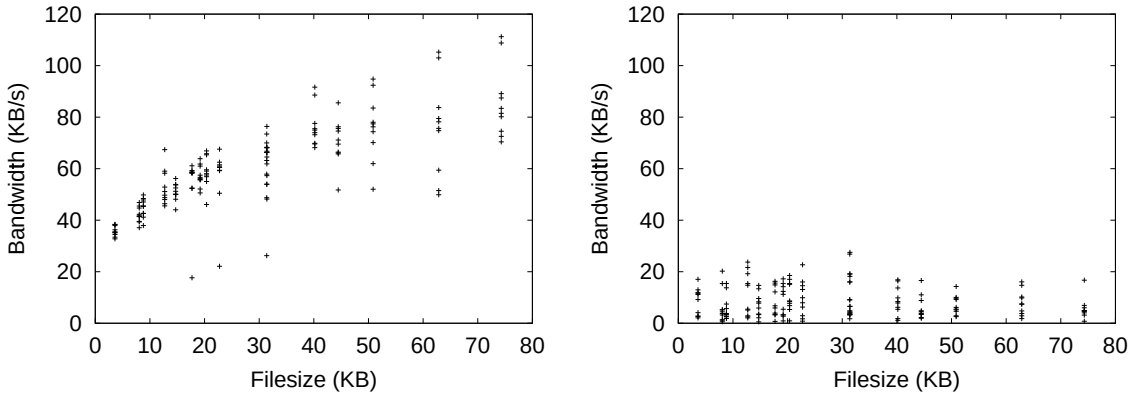

Figure 3.1: *Effect of file size on HTTP bandwidth. Graphs show HTTP GETs from* `www.state.ca.us` *to UTSA at night (left) and during the day (right).*

How valid is the assumption that post-latency bandwidth is independent of file size? In the absence of traffic congestion, the sliding window and slow start mechanisms of TCP/IP tend to produce higher bandwidths for larger objects. Under congested conditions, the packet drops and timer expirations reduce this effect. To test the validity of this assumption, we measured download times for various files between individual client-server pairs. Fig. 3.1 plots bandwidth for remaining read packets as a function of file size for downloads between the California state Web server and UTSA. Although the graphs show only one client-server pair, they are representative of other measured pairs. During nights and weekends when the network is quiescent, bandwidth tends to increase with file size (Fig. 3.1, left). However, during the day when the network is busy, bandwidth appears independent of file size (Fig. 3.1, right). This result indicates normalization is valid for busy periods, but it must be restricted to small size ranges for quiescent periods. Fortunately, we are most interested in high traffic periods where there is the most need for performance improvement.

3.5.3 Overhead

Dynamic and hybrid algorithms include network communication, therefore they are likely to have higher overhead than statistical algorithms which rely solely upon local table lookup. To insure fair comparisons, the overhead is measured and added to the normalized time:

$$\widehat{T}_{Total} = \widehat{T} + T_{Overhead}. \tag{3.3}$$

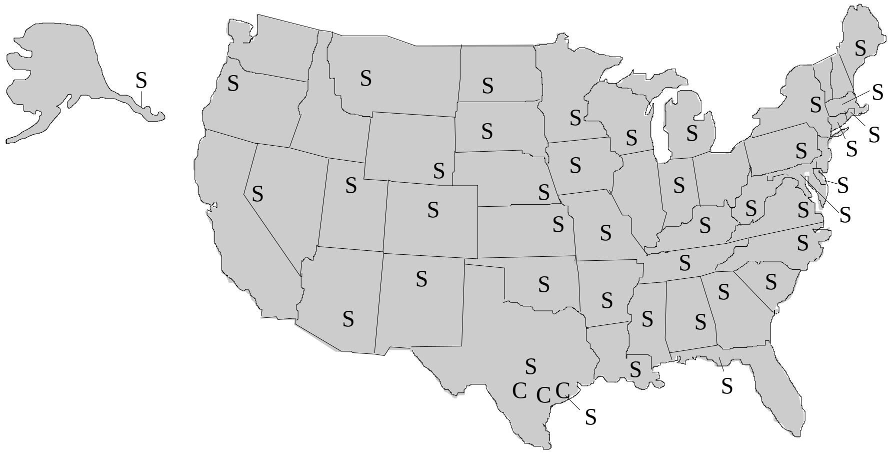

Figure 3.2: *Location of servers (S) and clients (C).*

3.5.4 Clients and servers

The server selection experiments were run concurrently on clients at three Texas universities (see Table 3.2). Although clients are located in the same state, they are in different cities, are connected to different regional networks, and use different primary Internet backbones. All have T1 or better connections to the Internet. To model a widespread caching or replication system, the server set must be distributed across the country and across Internet backbones. Servers should be stable and reasonably well-connected. We fulfill these conditions by constructing the server set from official state government Web servers, `www.state.**.us`, where `**` denotes the United States postal abbreviation. We also included two special servers that are near the clients, both geographically and topologically, and that have fast routes to the clients. Dividing servers into national and nearby subgroups allow us to vary the probability of having the object at a nearby server and observe how different selection algorithms respond. Fig. 3.2 shows the locations of the three clients and 42 servers used in the study. A few states were omitted because the test object moved during the calibration or experiment measurements, or because the server did not respond to `tcping` probes (see Section 3.5.7).

3.5.5 Measurement sessions

A measurement session begins by randomly picking ten unique candidates from the pool of 42 servers. The candidate set models a group of replicated mirror sites or cache servers which have copies of the desired

	A&M	**UH**	**UTSA**
Regional network	tx-bb.net	verio.net	the.net
Primary backbone	BBN Planet	Verio	Sprint
High perf. network	vBNS	vBNS	none
Operating system	Solaris 2.6	Solaris 2.6	Solaris 2.6 Linux 2.0.30
Calibration period	12/98 - 3/99	12/98 - 2/99	12/98 - 3/99
Calibration meas.	3309	3839	7178
Experiment period	3/99 - 5/99	3/99 - 5/99	3/99 - 5/99
Experiment sessions	1630	1710	1720
Image Day	290	276	242
Image Night	914	852	907
HTML Day	110	159	86
HTML Night	316	423	485
Connect failure rate			
Day	0.009	0.008	0.008
Night	0.007	0.008	0.008
Read failure rate			
Day	0.003	0.003	0.025
Night	0.001	0.001	0.001

Table 3.2: *Clients in the site selection experiments: Texas A&M (A&M), University of Houston (UH), and University of Texas at San Antonio (UTSA).*

object. As regional servers might return objects more quickly than most national servers, the probability of including one or more regional servers was fixed at 25% for all experiments except those which explicitly investigated the effect of nearby servers. This was accomplished by using different selection probabilities for regional and national servers. Within a session, each selection algorithm chooses a server from the candidate set, and the program immediately downloads the object from that server, recording number of bytes, algorithm overhead, connection time, read latency, and remaining read time. A session loops over the six selection algorithms in random order, pausing three minutes between algorithm measurements to avoid creating network congestion. One problem we constantly encounter is the high variability of repeated measurements. In many cases we observed that when more than one algorithm within a session selects the same server, the measured download times often differ substantially, with no obvious pattern. In order to fairly compare algorithms and eliminate this noise, the analysis uses the first session download time for a server for all algorithms in the session that selects that server.

The experiment consisted of over 1600 measurement sessions spanning a five week period. Table 3.2 lists the number of measurement sessions for each client under each experimental condition, together with failure rates due to connection refusals, connection timeouts, and read timeouts. If the download for any algorithm within a session failed, the entire session was excluded from the analysis. This had little effect, as connection and read requests failed infrequently. Connection failures occur in 0.7% to 0.9% of the measurements, and are independent of both client and time-of-day. Read failure rates are typically even smaller, ranging from 0.1% to 0.3%, except during the day at UTSA where the rate was 2.5%.

3.5.6 Calibration of bandwidth and latency predictors

Statistical selection algorithms require historic data to determine performance estimators such as latency or bandwidth medians. In this experiment, we collected calibration data prior to running the selection experiment, rather than continuously updating the estimators during the experiment. The calibration data consist of several thousand HTTP downloads for the same client-server pairs used in the selection experiment. The distributions of latency and bandwidth for individual client-server pairs are highly skewed, with extremely long, flat tails and exhibit large variability. Because of the skew, we use the median latencies and bandwidths, rather than the means, as statistical estimators [34]. Both time-of-day and file size effects are apparent in the calibration data, and, as Fig. 3.1 shows, these factors interact. This suggests that different statistical estimators be computed for different times of day and file size combinations. For this experiment, we used two time-of-day categories, (**Day** and **Night**), and two file size categories (**Image** and **HTML**). Combining categories creates four experimental conditions: **Image–Day**, **Image–Night**, **HTML–Day**, and **HTML–Night**. The **Day** category contains data for Monday through Friday, 9 am to 5 pm, and the **Night** category contains all other times. HTML files are small text files, normalized to 6 KB. Image files are larger GIF and JPG files, normalized to 50 KB. The calibration and selection experiment used one HTML and one image file from each server. Median latencies and bandwidths for each client-server pair under each of the four conditions were computed from the calibration data. In the selection experiment, the statistical and hybrid algorithms check time-of-day and object type, then use the appropriate estimator for current conditions.

3.5.7 `tcping`

For dynamic algorithms we developed the `tcping` probe, which sends a TCP SYN packet to an unused port on the server and listens for a TCP RST reply. This probe consists of a single round trip exchange of TCP header packets. By contrast, the standard `ping` utility uses the ICMP protocol to send an ECHO_REQUEST datagram to the server's echo port and listens for the ECHO_RESPONSE [59]. Because ICMP datagrams are sent over raw sockets, `ping` requires `root` privilege to execute. On UNIX systems, this permission is granted to users by setting the access mode of the executable file. Using TCP/IP instead of ICMP gives `tcping` the following advantages:

1. `tcping` is called as a user function. `ping` is invoked with a `system()` kernel call which forks a new execution shell and incurs context switching overhead.

2. A user program can send concurrent `tcping` probes to multiple servers and monitor their replies, allowing clients to run dynamic selection algorithms without `root` access or modification of the server.

3. `tcping` exercises a portion of the TCP protocol stack, therefore it includes an element of TCP/IP performance in its measurement.

For security reasons, some servers or their firewalls do not send ECHO_RESPONSE datagrams, rendering `ping` useless. Likewise, some may not send TCP RST replies for unused ports, which renders `tcping` useless. Of the 50 `www.state.**.us` servers, five did not respond to `ping` and five did not respond to `tcping`. Two cases overlapped; that is, two servers did not respond to either `ping` or `tcping`. One method for combining security features with dynamic probing would be to install a probe port on the server. A server probe port could also return information about current load on the application server, thus providing a simple, low-cost method for including load balancing in client-side selection methods. Source code for `tcping` is publicly available. Appendix C contains the source code listing and gives the Web site from which it can be obtained.

3.6 Results

3.6.1 Response time distributions

For each algorithm, the response time distribution exhibits large skew, long, flat tails, and high variability, as illustrated in Figure 3.3. This is not surprising, as these are the same characteristics observed in the calibration data for repeated measurements of a single object, and are consistent with results of other researchers [47]. The *Random* and *Probe* distributions shown in Figure 3.3 are the two extremes: *Random* has the longest tails and greatest variability, while *Probe* exhibits the shortest tails and least variability.

3.6.2 Medians and SIQR

As with latency and bandwidth estimators, the skew and outliers in these distributions make median the preferred index of central tendency and semi-interquartile range (SIQR[1]) the preferred index of dispersion [34]. Mean and standard deviation are inappropriate for response time because they are highly sensitive to the extreme outliers found in response time distributions.

3.6.3 Algorithm comparison

Fig. 3.4 compares performance of the selection algorithms at the three client sites using total normalized response time $\widehat{T}_{Total}$. These bar graphs show results for the four file size and time-of-day categories. Table 3.3 reports medians and SIQR statistics for total response time and for its component terms. Because both typical response time and its variability are important quality-of-service issues to the client, it is important to note which algorithms have low median and low SIQR values. From the bar graphs and Table 3.3, we observe that algorithm performance, from worst to best, is:

$$Random > Latency > BW \approx ProbeBW2 > ProbeBW \approx Probe.$$

$$slowest \longrightarrow fastest$$

[1] SIQR is one-half the difference between the 75% and 25% percentiles.

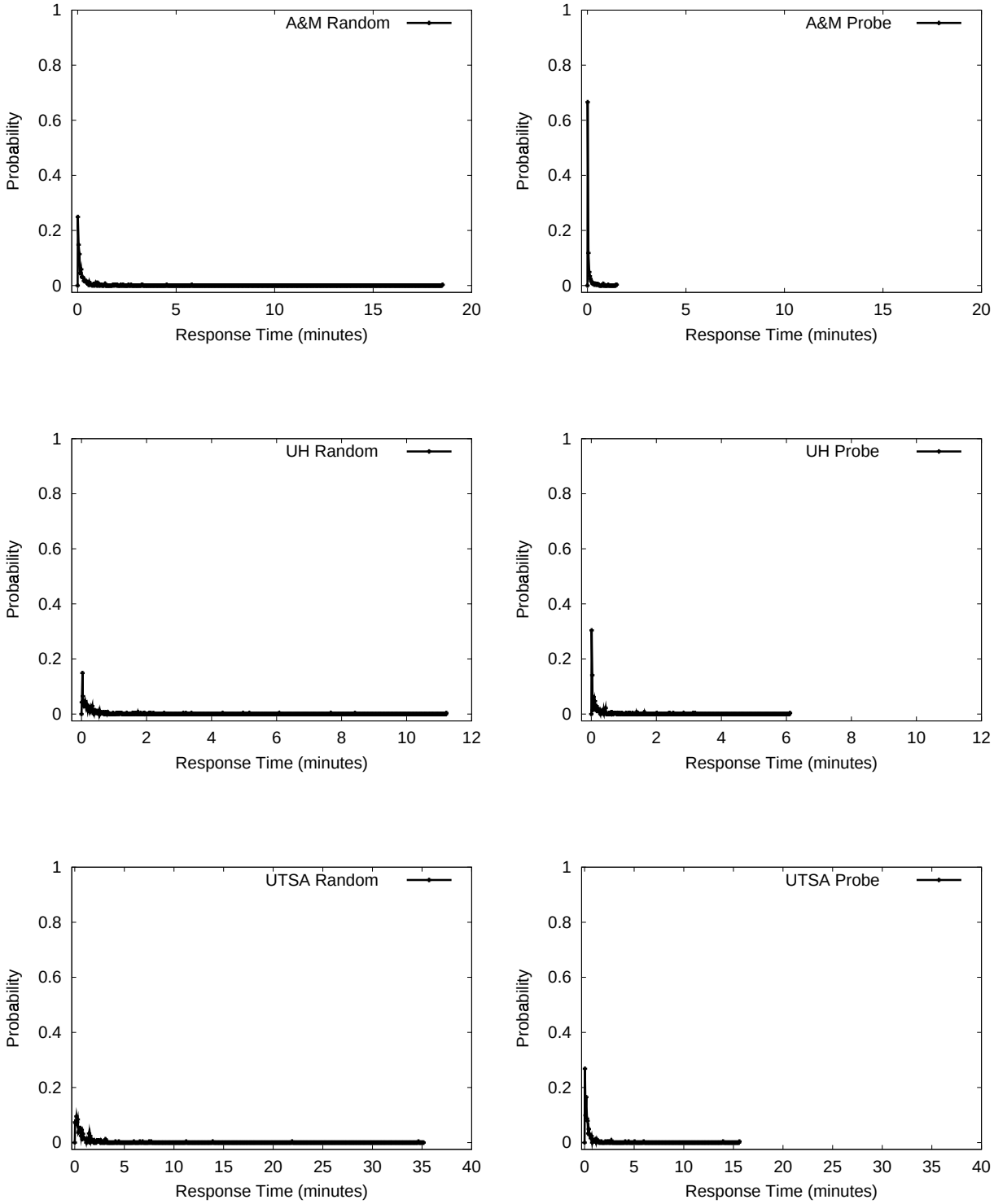

Figure 3.3: *Response time probability distributions for the* **Random** *and* **Probe** *algorithms at A&M (top), UH (middle), and UTSA (bottom).*

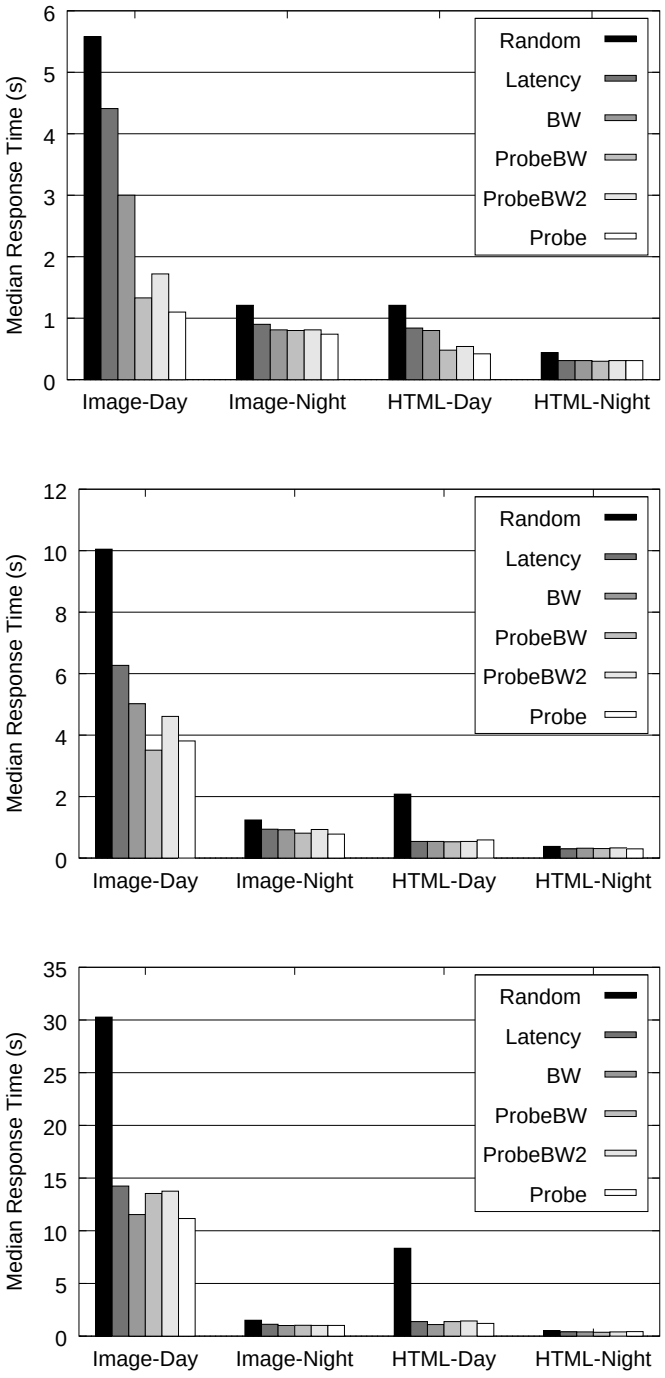

Figure 3.4: *Median response time, $\widehat{T}_{Total}$, for site selection algorithms at A&M (top), UH (middle), and UTSA (bottom).*

Algorithm	$T_{Overhead}$ *med*	$T_{Connect}$ *med*	*SIQR*	$\widehat{T}_{Read}$ *med*	*SIQR*	$\widehat{T}_{Total}$ *med*	*SIQR*
A&M							
Random	0.00	0.2	0.2	5.2	7.6	5.6	8.3
Latency	0.00	0.1	0.1	3.5	4.4	4.4	5.3
BW	0.02	0.1	0.1	2.3	3.2	3.0	3.5
ProbeBW	0.08	0.1	0.0	1.1	2.0	1.3	2.0
ProbeBW2	0.06	0.1	0.0	1.4	2.1	1.7	2.2
Probe	0.08	0.1	0.0	0.9	1.2	1.1	1.4
UH							
Random	0.00	0.1	0.1	9.6	10.3	10.1	10.7
Latency	0.00	0.1	0.0	5.9	5.2	6.3	5.3
BW	0.02	0.1	0.0	4.4	4.4	5.0	4.6
ProbeBW	0.07	0.1	0.0	2.3	5.6	3.5	5.8
ProbeBW2	0.05	0.1	0.0	3.7	5.4	4.6	5.8
Probe	0.06	0.1	0.0	2.9	5.9	3.8	5.9
UTSA							
Random	0.00	0.7	0.8	28.4	25.9	30.3	26.7
Latency	0.00	0.2	0.3	12.6	13.8	14.2	15.5
BW	0.00	0.2	0.2	10.7	11.4	11.5	12.3
ProbeBW	0.20	0.2	0.1	12.5	12.4	13.5	13.4
ProbeBW2	0.17	0.2	0.1	12.7	12.4	13.8	13.0
Probe	0.19	0.2	0.0	10.7	10.6	11.2	11.2

Table 3.3: *Response times components and totals in seconds ($\widehat{N}$ = 50KB, M-F 9am-5pm).*

In addition to the overall performance comparison, we note the following:

- All algorithms have low overhead.
- When the network is quiescent, there is little difference between algorithms.
- *Random* has the highest variability and, in most cases, *Probe* has the lowest.
- *Latency* performs worse than any algorithm except *Random*.
- The hybrid algorithms, *ProbeBW* and *ProbeBW2*, do not improve upon simple probing.
- *Probe* is markedly superior to *BW* at A&M, but less so at the other two clients. This result is explained in Section 3.6.7.

Algorithm	*Fraction of total time*				*Fraction ignoring Overhead*		
	$f_{Overhead}$	$f_{Connect}$	$f_{Latency}$	$f_{Remaining}$	$f_{Connect}$	$f_{Latency}$	$f_{Remaining}$
Image - Day							
Random	0.00	0.03	0.04	0.90	0.03	0.04	0.90
Latency	0.00	0.03	0.04	0.90	0.03	0.04	0.90
BW	0.00	0.03	0.05	0.88	0.03	0.05	0.88
ProbeBW	0.03	0.03	0.06	0.81	0.03	0.07	0.86
ProbeBW2	0.06	0.03	0.06	0.78	0.04	0.07	0.86
Probe	0.03	0.03	0.07	0.80	0.03	0.07	0.86
Image - Night							
Random	0.00	0.07	0.09	0.84	0.07	0.09	0.84
Latency	0.00	0.07	0.08	0.85	0.07	0.08	0.85
BW	0.00	0.07	0.10	0.82	0.07	0.10	0.82
ProbeBW	0.06	0.06	0.09	0.77	0.06	0.10	0.83
ProbeBW2	0.12	0.06	0.09	0.71	0.07	0.10	0.82
Probe	0.07	0.05	0.08	0.77	0.06	0.09	0.84

Table 3.4: *Component of response time as fractions (medians).*

3.6.4 Component fractions: overhead, connection, latency, and read time

Table 3.3 reports medians of elapsed time, in seconds, for algorithm overhead, connection establishment, and read time normalized to a 50 KB reference size. The normalized read time is the measured latency plus the normalized remaining read time as defined in Eq. 3.2:

$$\widehat{T}_{Read} \;=\; T_{Latency} + T_{Remaining}\left(\frac{\widehat{N} - N_{Latency}}{N - N_{Latency}}\right) \tag{3.4}$$

The relative fractions of total time, $\widehat{T}_{Total}$, attributable to overhead, connection, latency, and normalized remaining read time are given in Table 3.4. These fractions were obtained by computing fractions for each measurement and taking medians over all measurements. The smallest contribution comes from overhead. Overhead for statistical algorithms is insignificant because these algorithms rely upon local table lookup. Overhead for dynamic and hybrid algorithms using `tcping` are only slightly higher, with medians ranging from 20 to 70 ms at the two better-connected clients, and from 40 to 200 ms at the other client. Interestingly, connect, latency, and remaining read times tend to follow the same algorithm performance order as total time. Table 3.4 shows that, after correcting for overhead, their relative contributions are nearly constant

across algorithms. For example, selecting servers randomly tends to result in longer connection delays than those for the other five selection algorithms. *Random* also produced the most pathologically long connection delays, while *Probe* produced the fewest: 29 of the 5060 measured downloads for *Random* required over 10 seconds to establish a connection, compared to 21 for *Latency*, 18 for *BW*, and 6 for *Probe*. The same patterns hold for latency and normalized read time.

Although the medians in Table 3.4 are relatively constant, all components except for $T_{Overhead}$ contain outliers that far exceed median total response time.[2] For example, several connects required 70 – 90 seconds while median total time was under 20 seconds. In general, connect, latency, and read time exhibit the same large skews and long tails as $\widehat{T}_{Total}$. We attribute the skews and tails to router packet drops forcing TCP/IP retransmissions and window adjustments.

3.6.5 Time-of-day effects

Differences between server selection algorithms are far more pronounced during busy daytime hours than for quieter night and weekend periods. These differences reflect the periodicity of Internet traffic and the ability of the algorithms to detect network congestion. Internet traffic and server loads increase in the mornings, decrease in late afternoon, and are much lower on weekends than during the week. To simplify the analysis, we separate data into only two time categories, **Day** and **Night**, choosing boundary hours of 9 am and 5 pm CST. The cutoff hours were chosen by plotting medians of $\widehat{T}_{Total}$ *vs.* time-of-day for weekday data, and observing the hours where $\widehat{T}_{Total}$ rises and falls sharply (see Fig. 3.5). In the remainder of this paper, we concentrate on results for the larger **Image** files during **Day** hours, where waits are longer and performance improvement is most needed. The two hybrid algorithms always performed worse than *Probe* and better than *Latency*. These are omitted from Fig. 3.5 for the sake of clarity.

Hourly plots of elapsed time point out an important fact: the performances of the selection algorithms maintain their relative order as the load changes. With minor exceptions, the curves do not cross, implying that as load changes clients need not use different algorithms to achieve optimal performance.

[2]The number and magnitude of outliers are limited by timeout values. Our experiments use a 5 second timeout for `tcping`, 90 seconds for connect, and 10 minutes for read.

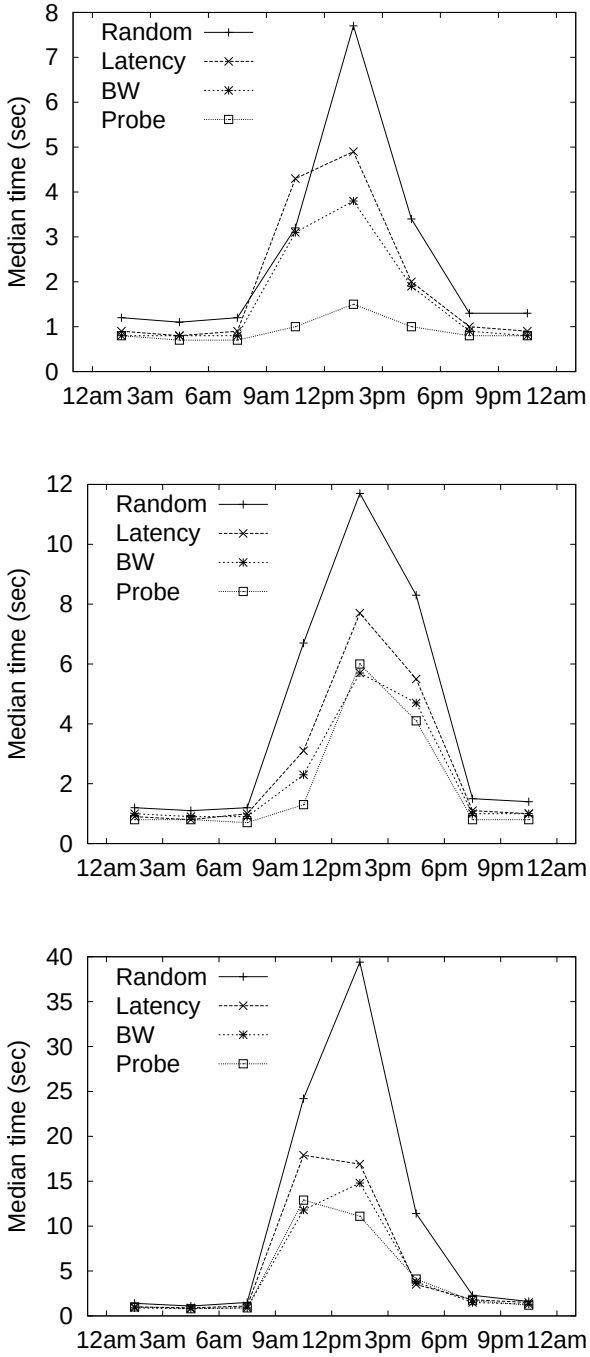

Figure 3.5: *Effect of time-of-day on site selection algorithms for 50 KB files at A&M (top), UH (middle), and UTSA (bottom).*

3.6.6 Pathological delays

One goal of server selection algorithms is to significantly reduce the number of pathologically long wait times. As noted in Section 3.6.4, connect time, latency, and remaining read time can each be responsible for pathologically long delays. Many long delays cannot be predicted by past connection, latency, or bandwidth measurements, nor by `tcping` probes sent immediately prior to the download. In rare cases, the server with the lowest historic latency, highest previous bandwidth, and fastest `tcping` probe timed out on a connection request, or took tens of minutes rather than seconds to deliver the file. To determine which algorithms best reduce pathological behavior, we plot the cumulative distribution function (CDF) of total response times for each algorithm (Fig. 3.6). Once again we find the same ordering of algorithm performance: *Probe* and *BW* typically produce the fewest pathological cases, *Random* produces the most, and *Latency* turns in an intermediate performance. The CDF curves for hybrids *ProbeBW* and *ProbeBW2* are close to that for *Probe* and are again omitted for clarity.

CDF curves also allow us to compare performance distributions at clients with different connection bandwidths. We compare across clients by drawing a vertical line to mark the median time for *Random*, then reading cumulative distribution values for other algorithms at that client. Alternatively, we can expand or contract the horizontal axis according to the ratios of *Random* medians. A&M's median time for *Random* is approximately half that of UH, and one-sixth that of UTSA. When we scale the time axes by these ratios, the CDF curves have similar shapes, indicating the algorithms have similar performance distributions at different clients. The distribution tails of the algorithms are not noticeably longer or flatter at more poorly connected clients, Consequently, if an algorithm produces fewer pathological outliers at one client, it will likely produce fewer pathological outliers at all clients.

3.6.7 Effect of network modifications

The effectiveness of statistical algorithms such as *Latency* and *BW* is limited by the accuracy of their calibration data. If the network connection between a client and server is upgraded, statistical algorithms will underperform until their calibration data are updated. This problem is illustrated in the results for A&M. Subsequent to calibration, a high speed connection was established between A&M and the University of

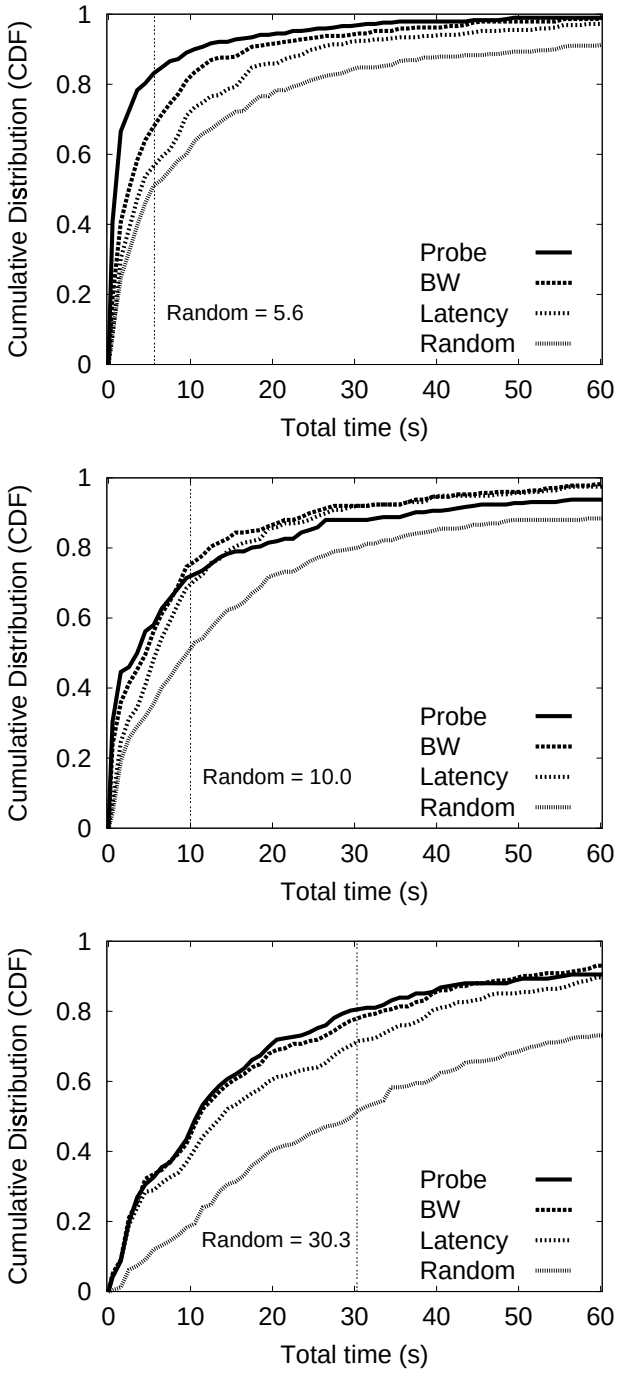

Figure 3.6: *Cumulative distributions of $\widehat{T}_{Total}$ for 50 KB files, M-F 9am to 5pm, at A&M (top), UH (middle), and UTSA (bottom).*

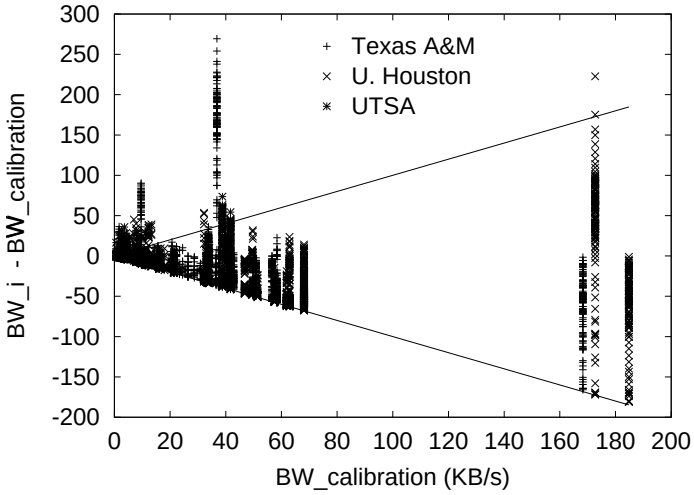

Figure 3.7: *Bandwidth errors for individual measurements.*

Texas Medical Center in Houston, a member of the server set. A second, less dramatic improvement was made to a connection between A&M and another server in the set. Because *BW* and *Latency* relied upon stale bandwidth and latency data, they did not choose these two servers as often as they should have, and consequently did not perform as well at A&M as they did at the other two clients. The upgraded servers stand out when bandwidth errors are plotted against the calibration bandwidth, as in Fig. 3.7. In this graph, bandwidth error for the i^{th} measurement is given by $Error_i = BW_i - BW_{calibration}$. Data points above the upper diagonal line have a measured bandwidth over twice the calibration bandwidth. At 38 KB/s, there is a vertical smear of points well above this upper line, and a shorter series of points at 9 KB/s. These points flag the two upgraded connections, whose identity was easily determined by referring to calibration data.

3.6.8 Importance of nearby servers

In this section, we investigate the importance of locating caches or other services at topologically nearby servers with relatively fast network connections to the client. A topologically nearby connection does not go through an Internet backbone or backbone exchange point. The server set includes two well-connected servers located in the same state as the clients: the University of Texas at Austin and the University of Texas Medical Center in Houston. All three clients connect to these nearby servers through either regional

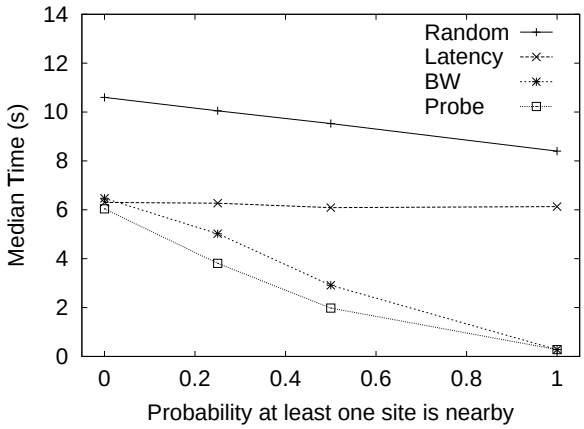

Figure 3.8: *Effect of including nearby servers at UH.*

networks or vBNS. As explained in Section 3.5.5, each session has a 25% probability that its candidate set includes at least one nearby server, or $P = 0.25$. How might results change if we had chosen a different probability? To answer this question, we examined session data to see if a session includes at least one nearby server and, if so, included that session with probability P in the analysis. Fig. 3.8 shows results for $P = 0$, 0.25, 0.50, and 1. $P = 0$ models the case where no nearby servers have a copy of the desired object, and $P = 1$ models the case where at least one nearby server has a copy. The ordering for median total times

$$Random > Latency > BW \geq Probe$$

holds across probabilities. For two clients, the *Probe* and *BW* curves drop more steeply, while *Latency* has an almost flat response, indicating *Probe* and *BW* are more efficient at finding nearby servers. At $P = 1$, the median total times for *Probe* and *BW* fall to well under a second at two clients, and under 3 seconds at the third client. Compared to random selection with no nearby servers, we obtained an order-of-magnitude speedup at all three clients by including at least one nearby server and using *Probe* or *BW*.

3.7 Cache speed differential

Assuming caches and servers are randomly located, the difference in response times between the *Random* and *Probe* algorithms represents the attainable speed differential for Web caching. Because network and server delays are most pronounced for larger objects during weekdays working hours, we consider the **Day-**

	$\widehat{T}_{Random}$	$\widehat{T}_{Probe}$	$\Delta\widehat{T}$	$\widehat{T}_{Random}/\widehat{T}_{Probe}$
A&M	5.6	1.1	4.5	5.09
UH	10.1	3.8	6.3	2.66
UTSA	30.3	11.2	19.1	2.71

Table 3.5: *Response time differences and speedups for* Probe *vs.* Random *selection. Time are medians for 50 KB objects, M-F 9am - 5pm, and are given in seconds.*

Image condition. Under this condition, the median difference in time between not caching (*Random*) and caching (*Probe*) is 4.5 seconds for A&M, 6.3 seconds for UH, and 19.1 seconds for UTSA. In relative terms, the ratio between median cache response time and median server response time is 0.20 for A&M, 0.38 for UH, and 0.37 for UTSA. That is, retrieval from a set of remote caches typically 2.6 to 5 times faster than retrieval from the origin Web server. Table 3.5 summarizes these results.

The magnitude of our results applies specifically to a cache set of size 10 and to a "nearby-cache" probability of 0.25. We expect the cache speedup to increase when there are more cached copies and when more caches are located in the same regional network or high-speed network as the client. Similarly, the cache speedup will likely decrease when there are fewer candidate caches or when there is a lower probability that a cache is nearby.

3.8 Site selection summary and discussion

- **Use dynamic probes instead of statistical estimators to select servers.**

The principal conclusion of this study is that client-side selection algorithms should use simple network probes to select servers, instead of past performance data such as latencies or bandwidths. *Probe* clearly outperforms *Latency*, and performs as well or better than *BW* under all conditions. Dynamic probes are easy to implement, adapt automatically to changes in the network or server, and add little traffic or delay overhead. Conversely, these experiments highlight several disadvantages of statistical algorithms. Algorithms that rely on statistical estimators are limited by the accuracy and variability of the calibration data. Good statistical estimators for Internet bandwidths and latencies are difficult and cumbersome to obtain because 1) clients

must collect data for each potential server, 2) bandwidth and latency depend upon time-of-day and day-of-week, 3) repeated measurements exhibit large skew, long tails, and high variability, and 4) estimators go stale when network or server resources are upgraded.

When the candidate set is large or dynamic, it is difficult to maintain statistical data for each potential server. Time-of-day and file size dependencies force clients to repeat calibration measurements for each server during each time category and for several file sizes. If measured data were stable, this would not be a problem. Unfortunately, the high skew and variability of latency and bandwidth data require clients to collect many repeated measurements for each server under each condition. Even then, the spread in the distributions limits performance of statistical algorithms: the more server distributions overlap, the less accurate the selection.

Finally, statistical algorithms have difficulty adapting to outages and upgrades in the server or network. We saw the consequences in the A&M results, where stale calibration data substantially degraded performance. Stale data can be avoided if clients either periodically recalibrate or continuously collect data from actual downloads and use a sliding calibration window. Both methods have problems. Periodic recalibration adds substantial network traffic, while data from actual downloads are slow to discover upgrades because statistical algorithms tend to avoid servers with poor past performance.

- **Include nearby servers in the candidate set.**

All the selection algorithms, including random selection, perform better when a topologically nearby server is included in the candidate set. For HTTP downloads during the day, we achieve speedups of 8 to 35 by using either *Probe* or *BW* and including at least one nearby server, as compared to randomly selecting a server from a candidate set that did not include any nearby servers.

- **Use total elapsed time rather than latency or server load to evaluate performance.**

In terms of performance, the goal of distributed caching and replication is to reduce user response time. Server load is not an appropriate metric for client-side selection algorithms because it does not reflect

differences in network transmission times. When latency is used both as the estimator and as the metric [55], results prove only that prior latency is a good predictor of current latency. The results of this study show prior latency is not as successful as either prior bandwidth or network probes at predicting response time.

- **Pathologically long delays can be reduced but not eliminated by these algorithms.**

None of the algorithms eliminates pathologically long delays; however, all algorithms reduce the number of such delays by 7% to 20% over random selection. Pathological behavior is difficult to predict because it appears in each component of total time: connect, latency and read distributions all exhibit the same large skew and long, flat tails that characterize total time. Statistical algorithms reduce the number of long delays because some client-server connections statistically experience fewer long delays. Network probes reduce pathological behavior because they discover current congestion problems. Neither method is completely successful. We find connection, latency and read times can each be responsible for pathological behavior. This argues for including all components in the evaluation metric, even though one term may dominate the median value.

- **Use bandwidth data to reduce the number of probes.**

When the number of potential servers is large, we do not wish to send dynamic probes to every server. The hybrid algorithm *ProbeBW* filters out servers with historically poor bandwidth, then sends probes to the remaining servers. Our results show this hybrid method performs almost as well as probing all servers, and, unlike *BW*, responds robustly to network upgrades and poor calibration data. This robust behavior makes it feasible to compute calibration data from a sliding window of prior downloads. Consequently, *ProbeBW* offers a viable method for reducing the number of dynamic probes for large numbers of candidate servers.

- **Consider using server load probes to improve performance**

Statistical algorithms reflect past network and server loads, but offer no information about current conditions. Network probes reflect current network load, but provide no information about current server load.

If the server is overloaded with respect to its CPU or disk subsystem, the delay is unrelated to network congestion. More research is needed to determine how often such situations occur and whether they can be predicted using a dynamic server load probe. One approach to a server load probe would be to install an information port on the server. The HTTP server process would periodically post current load information to the load port through shared memory or other low-cost mechanism. Operating independently of the HTTP process, the information port would reply to load queries with a small UDP packet. (Alternatively, Fei, et al., suggest the server push its load information to client agents [27].) Implementing such an information port and load probes is easy. The difficultly lies in testing whether server load probes improve server selection for a representative set of Web servers, which currently provide no load data to their clients. Studies separating network and server effects are an important area of future research.

Chapter 4

Taxonomy and design analysis

4.1 Introduction

In this chapter, we discuss design guidelines and introduce a taxonomy for cooperative Web caches. The taxonomy is useful for comparing caching projects because it separates basic design choices from implementation details. Further, it helps focus on factors that may not be measured in simulation or performance studies. We analyze how alternative approaches in the taxonomy match characteristics of the Web. Although our analysis is primarily qualitative, it serves as a guideline for basic design decisions.

4.2 User and global performance considerations

A network cache should improve both individual user performance and global network performance. The cache design is unacceptable if it improves an individual's response time at the expense of increased network congestion and causes longer delays for other users. Unlike caches for memory and disks, cache sites on wide-area networks are not inherently faster than servers because cache sites do not necessarily have faster hardware or larger bandwidths. If caches and servers have similar resources and connection bandwidths, remote network caching actually increases average response time *unless* it offsets discovery overhead and cache miss penalties by reducing delivery time. We can reduce delivery time if we know which caches have copies and select a fast site. Unfortunately, it is difficult to predict response times because they depend upon current server load and network congestion, as well as upon static factors such as server capacity and network topology. Because Web pages typically contain several independent objects, we can reduce overall page

retrieval time by concurrently retrieving objects from different cache sites. This concept is similar to parallel HTTP, in which browsers send concurrent requests to the same server. However, parallel HTTP perturbs the TCP/IP congestion control mechanism by generating multiple client-server connections that use the same network route [7]. Spreading the concurrent requests across different sites reduces the traffic per route, which reduces burstiness because queuing delays on different routes are somewhat independent. In summary, the design goals for distributed Web caching should include 1) low discovery cost, 2) object dissemination that adapts to popularity shifts, 3) a method for selecting fast caches, and 4) concurrent delivery of page or object components from different caches.

4.3 Workload characterization

Cache designs perform well only if they match the workload, so first we examine the pattern of HTTP requests. Table 4.1 summarizes statistical characteristics of HTTP server workloads reported by a number of researchers, and Table 4.2 reports the HTTP workloads we observed on two Web servers at the University of Texas at San Antonio: UTSA-CS and UTSA-VIP. Characteristics of the UTSA traces appear similar to traces reported in the literature. Studies uniformly show skewed popularity distributions: most Web requests are for a small fraction of the objects. Arlitt and Williamson [6] report object size is well-modeled by Pareto distributions and the interrequest intervals for individual objects follow exponential distributions. Seltzer's results [56] show object popularity varies by region, and Web invariants apply most strongly to the more popular servers.

4.3.1 Web page statistics

Two statistics not reported in these earlier studies are the fraction of embedded image requests and the number of embedded images per page. These data are important for evaluating client prefetching or concurrent retrieval algorithms. In the UTSA traces, we found embedded images account for an average of 41% of the transfers and 33% of the returned bytes, with an average of 3.2 images per page.

Characteristic	*Reference*	*Notes*
Server load is growing exponentially	Seltzer [56]	Requests are growing exponentially. Documents are growing exponentially.
Most requests are from remote clients	Arlitt [6]	Remote clients: $\geq$ 70% of requests Remote clients: $\geq$ 60% of bytes.
Arrival time distribution is heavy-tailed	Mogul [46]	Appears log-normal.
Most transfers are small	Arlitt [6] Almeida [3]	Mean size $<$ 21 KB Image 13 KB Text 4 KB, Audio 179 KB Video 2300 KB
Object size distribution is Pareto	Arlitt [6] Crovella [18]	$0.40 < \alpha < 0.63$ for transferred objects. $\alpha = 1.06$ for transferred objects.
Most transfers are images or HTML	Arlitt [6] Gwertzman [32] Almeida [3]	Image 36-78% HTML 20-50% Dyn $<$7% Image 65% HTML 22% Dyn 9% Image 75% HTML 19% Dyn 0% Audio 5% Video 0.4%
Object popularity is skewed	Arlitt [6] Bestavros [10]	10% of objects satisfy 90% of transfers. 5% of bytes satisfy 85% of byte traffic.
Most transfers are duplicates	Arlitt [6]	$>$ 97% sent more than once.
Objects have long lifetimes	Bestavros [10]	JPEG 100 days GIF 85 days HTML 50 days
Popularity varies by region	Seltzer [56]	Clients often access nearby servers.
Popularity can change rapidly	Seltzer [56]	Objects and server popularity can change rapidly, producing *flash crowds*.

Table 4.1: *Summary of reported HTTP Web server characteristics.*

Characteristic	*UTSA-CS*			*UTSA-VIP*		
Dates	Apr 97 - Sept 97			May 96 - Aug 96, Dec 96 - Aug 97		
Number of requests	561,292			547,272		
Traffic (bytes transferred)	3.8 GB			2.9 GB		
Remote clients	82.4%			77.9%		
Mean object size	<12 KB			<11 KB		
Object type	Size	Transfers	Bytes	Size	Transfers	Bytes
HTML	4 KB	44.2%	15.2%	4 KB	41.9%	15.2%
Image	15 KB	47.6%	61.8%	6 KB	53.7%	30.4%
Audio	200 KB	0.4%	7.6%	81 KB	0.1%	1.1%
Video	na	0.0%	0.0%	452 KB	0.9%	40.2%
Application	135 KB	1.0%	12.2%	386 KB	0.3%	10.2%
Dynamic	1 KB	3.6%	0.3%	1 KB	0.1%	0.0%
Other	11 KB	3.2%	2.9%	10 KB	3.0%	2.9%
Object popularity	10% of objects = 69% of transfers			10% of objects = 79% of transfers		
Duplicate transfers	>99% sent more than once			>99% sent more than once		
Embedded images	37% of transfers		43% of bytes	46% of transfers		24% of bytes
Request types:						
Session, single file	50%			41%		
Session, multi-file	13%			13%		
Component	37%			46%		
Embedded images per page	2.9			3.5		

Table 4.2: *UTSA Web server characteristics.*

4.3.2 HTTP session statistics

We categorize HTTP requests as either *session requests* or *component requests.* Users initiate a session request when they instruct the browser to retrieve a Web page or an independent object. For Web pages, the browser first retrieves the HTML file (the session request), then issues component requests for the embedded objects. Requests for independent objects such as multimedia, postscript, and text-only HTML files generate only a session request; no component requests follow. In the UTSA traces, we observed an average of 59% session requests and 41% component requests. Within the session requests, 13% correspond to multi-object Web pages and 46% to single, independent Web objects.

Distinguishing between session and component requests is important because human-initiated session requests are uncorrelated, while machine-generated component requests are correlated with other requests within the session. Consequently, we expect different arrival time behaviors. For FTP, TELNET, SMTP and NNTP, Paxson and Floyd [52] find Poisson distributions are valid for user session arrivals, but not for machine-generated requests. Machine-generated requests exhibit burstier behavior that is better modeled by self-similar processes. In particular, Paxson and Floyd show FTP session arrivals follow a Poisson distribution, while FTPDATA connections within a session are clustered into bursts for which Poisson modeling fails.

4.4 Taxonomy

Distributed network caches provide three services: *discovery, dissemination* and *delivery* of cache objects. Discovery refers to how clients locate a cached object. Dissemination is the process of selecting and storing objects in the caches; that is, deciding which objects are cached, where they are cached, and when they are cached. Delivery defines how objects make their way from the server or cache site to the client. Our taxonomy is given in Table 4.3 and explained in the sections below.

Function	*Categories*	*Details*
Discovery	**Fixed cache**	Client always sends request to the same cache.
	Group query	Client queries a group of cache sites.
	Manual	Clients maintain individual member addresses.
	Auto	Clients maintain a group address.
	Directory lookup	Client looks up object in a metadata directory.
	Centralized	One centralized directory for each server.
	Distributed	Metadata directory is distributed across nodes; (distributed structure may be hierarchical, mesh, etc.).
Dissemination	**Client-initiated**	Object cached as a result of client request (*pull-caching*).
	Server-initiated	Server decides what, where and when to cache (*push-caching*).
Delivery	**Direct**	Object always returned directly to the client.
	Indirect	Object may pass through several sites before reaching client.

Table 4.3: *A taxonomy for cooperative Web caches.*

4.4.1 Discovery

In *fixed cache discovery*, a client sends all cache requests to a single, pre-configured location. Servers are stateless; it is the client who maintains knowledge of the cache location. Stateless server caching has two advantages: no caching information is lost in the event of server failure, and the design requires no changes to existing Web servers. Fixed cache discovery is used in proxy server caching, where browsers are configured to send all requests to their site's proxy server.

In *group query discovery*, clients locate cached copies by querying members of a cache group. This permits servers to be stateless. Clients are responsible for knowing where to send their queries, requiring some method for maintaining group membership information and addresses. Two methods have been proposed: *manual configuration* and *automatic configuration.* With manual configuration, clients must be informed when the group membership changes. Typically the clients' systems administrators must manually modify their configuration files. This is the approach in the Harvest [16] and Squid [58] caching software, which run on many national cache systems throughout the world [48]. Zhang and coauthors point out manual configuration is cumbersome, error-prone and does not scale. Instead, they propose using IP multicast to automatically configure cache groups [70]. Queries are sent to a multicast address, so clients need not be

reconfigured every time a member joins or leaves the group. A drawback to group query is that competing factors constrain group size. If the cache group is too large, the client floods the network with query packets. IP multicasting will not relieve this flooding until true multicast routers are in widespread use. Even then, replies to the query may cause congestion and overloading. On the other hand, Web caching cannot achieve high hit rates unless cache groups are large because there is not enough overlap in client requests within smaller groups. Cache hierarchies attack the size problem by organizing groups into a cache tree. Although hierarchies improve aggregate hit rate, the response time suffers because each subsequent cache level miss spawns another remote network transfer of the object. Thus distributed cache hierarchies cannot achieve both high hit rates and low overall response time.

With *directory discovery*, the client locates cached copies by looking in a metadata directory. Along with copy locations, directories may store timestamps or version numbers for cache consistency, access control data for security, and site performance figures for use in resource selection. Retrieving metadata involves the same discovery, dissemination and delivery issues as retrieving the data objects. However, metadata is smaller than the corresponding data, which lowers propagation and prefetching costs.

Centralized directories store metadata for a server's objects together on one network node, with the obvious advantage that clients immediately know which directory to contact. Unfortunately, centralization does not scale and ultimately leads to congestion and bottlenecks. Server-initiated dissemination approaches typically use stateful servers and centralized directories. *Distributed directories* spread metadata for a server's objects across the network.

4.4.2 Dissemination

Client-initiated dissemination ("pull-caching") is a client-driven technology in which clients determine what, when and where to cache. Servers may be stateless because they do not need to know client request patterns; the client-driven nature automatically matches object distribution to request patterns. The problem of cache consistency arises if servers are stateless, but this not associated *per se* with client-initiated dissemination. The major advantage of client-initiated dissemination is that it automatically adapts to rapidly changing request patterns. This is critical when hot spots or flash crowds occur.

Server-initiated dissemination ("push-caching") is a server-driven technology in which servers choose what, when and where to cache [10][31]. Cache sites may, however, have the authority to refuse or remove objects. Replication is a server-initiated dissemination method that restricts the authority of the cache sites to refuse or remove objects. On the Internet, replication is commonly associated with "mirror sites" that duplicate an entire Web site. In general, server-initiated dissemination operates on a finer scale by storing individual objects at remote cache sites. Server-initiated dissemination uses stateful servers because servers need historical data to make dissemination decisions. This server-centric approach provides strong consistency and assures objects with long-term demand are not prematurely replaced by short-term, "hot" items. Conversely, server dissemination does not cope well with rapid or localized changes in request patterns [31]. A second disadvantage lies in resource and security concerns that arise because cache sites store unsolicited objects instead of only storing objects requested by local users. This raises serious concerns on the Internet, where sharing occurs across different companies and governments. In short, server-initiated dissemination works best for objects with long-term or static request distributions, or for objects where consistency is more important than response time.

4.4.3 Delivery

Direct delivery always returns the object directly from the "hit site" to the client, where hit site refers to the cache or server at which the object is found. This method assures the lowest latency for delivery.

In *indirect delivery,* the object may travel through intervening cache layers on its path from hit site to client. Each layer adds a store-and-forward network transfer, which generates longer and more variable response times, and increases the number of packets on the network. Performance penalties are greatest for large multimedia objects. Typically, indirect delivery follows the reverse request path. Intervening cache layers may save the object as it passes through; combining delivery with dissemination.

The HTTP protocol has no provision for returning an object other than via the request connection. This has repercussions for multi-level cache protocols that may send the request through several cache levels. Consider the following scenario: A requests an HTTP object from B, B misses and requests the object from C, and C hits. Under the HTTP protocol, C cannot return the object directly to A because C is not

Discovery	*Dissemination*	*Delivery*	*Projects*	*Ref.*
Fixed cache	Client-initiated	Direct	Proxy server caching	[5]
Group query, Manual	Client-initiated	Indirect	Harvest/Squid hierarchical caches	[16]
			NLANR, other national caches	[48]
Group query, Automatic	Client-initiated	Indirect	Adaptive Web caching	[70]
			Cooperating WWW cache servers	[44]
Directory, Centralized	Server-initiated	Direct	Geographic push-caching	[31]
			Demand-based dissemination	[9]
Directory, Distributed	Server-initiated	Direct	Metadata hierarchy	[62]
Directory, Distributed	Client-initiated	Direct	Server-directed proxy sharing (SDP) (Appendix A)	[22]

Table 4.4: *Classification of cooperative Web caching projects.*

connected to A. Even if C knew about A, it would have to initiate the connection as a client rather than a server. Consequently, multi-level Web caches must either use indirect delivery or use a protocol other than HTTP.

4.5 Web caching projects

Combining choices for discovery, dissemination and delivery yields twenty taxonomy classes. Table 4.4 uses the taxonomy to classify several Web caching projects.[1] The projects are discussed below, classified according to taxonomy class and identified using common terminology.

4.5.1 Hierarchical Web caches

Manual group query projects using Harvest and Squid caches organize proxy servers into hierarchical groups. When a cache enters or leaves the system, its siblings and children must be manually reconfigured. The National Laboratory for Applied Network Research (NLANR) manages a hierarchical Internet cache system built upon Squid caches. NLANR supplies the top level cache group, proxy servers constitute the intermediate levels, and users make up the bottom level.

[1] With proxy servers, a protocol is classified as direct delivery if objects are returned directly to the proxy server.

4.5.2 Multicast groups

The automatic group query project of Zhang, Floyd, and Jacobson [70] partitions users into multicast groups for discovery and dissemination. IP multicast allows for automatic group configuration: caches join and leave groups without having to reconfigure other group members. Within a cache group, the request is multicast to all members. If no member has the object, the request is forwarded via a cache in the overlapping group closest to the server. Delivery follows the reverse route, with the object multicast to every group along the way. Acceptable performance will likely require hardware IP multicast to avoid flooding the network with queries. Because group members need to determine which overlapping group is closest to the server, this project requires some integration between caches and routers. Malpani, Lorch and Berger [44] propose a different multicast discovery design that uses a single group. If the group misses, the protocol resorts to contacting the server. Here the problem is scalability: a single group design does not scale to the Internet, but smaller groups suffer from the same lack of request overlap that limits proxy server hit rates.

4.5.3 Push-caching

Push-caching projects of Gwertzman, et al. [31] and Bestavros, et al. [9] use stateful servers that decide what, where, and when to cache. Clients send requests directly to the Web server, which redirects the client to a "nearby" cache. Servers decide which cache is nearest the client by comparing geographic or network topology (hop count) information. However, Crovella and Carter [18] show geographic distance and hop count both poorly predict the latency of Internet traffic. Consequently, these projects face difficulties in responding to flash crowds and in selecting cache sites.

4.5.4 Metadata directories

In the metadata approach, information about cached objects is stored locally on the proxies and propagated separately from data objects. Metadata can be propagated in different ways. Tewari, et al., [62] propose a hierarchical push structure, while Fan, et al., [26] suggest compressing the metadata into a *cache digest* for exchange with other proxies. In [22], we proposed a lazy prefetching technique that piggybacks a small amount of metadata onto object transfers. Appendix A describes this technique in more detail.

Element	*Recommendation*	*Rationale*	*Reference*
Discovery	**Distributed directory**	Sparse user sharing (low Zipf α)	Section 2.5.5
Dissemination	**Client-initiated**	Sparse user sharing "Flash crowds"	Section 2.5.5 [32][39]
Delivery	**Direct**	Lower response time Less network traffic	[62]
Organization	**Flat mesh**	Multi-level HTTP caches must use indirect delivery.	[28]
Propagation of metadata	**Night: cache digests Day: updates**	Infrequent cache replacement Diurnal congestion pattern	Section 2.5.1 Section 3.6.5

Table 4.5: *Design analysis for cooperative Web caches.*

4.6 Design recommendations

Our taxonomy analysis argues the best distributed caching approaches for Internet HTTP traffic are 1) directory-based discovery using a distributed directory, 2) client-initiated dissemination, and 3) direct delivery. The rationale are summarized in Table 4.5 and detailed below.

4.6.1 Directory-based discovery

Directory-based discovery best fits Web request patterns because, as established in Section 2.5.5, Web documents are sparsely shared between proxies. For example, the NLANR BO1 cache services 90 proxies, yet Figure 2.5 shows that only 10% of the requested documents are shared. Moreover, only 2% of the documents are shared by more than 2 proxies, and 0.6% are shared by more than 3 proxies, These figures ignore the cache warmup effect which increases hit rate by a factor of 4. However, even a four-fold increase in these sharing rates would be considered spare.

Spare sharing is incompatible with query-based discovery because query systems exhibit low hit rates when sharing is spare and there are few proxies in the group. However, increasing group size leads to unacceptable overhead because overhead increases as $O(RN_{Proxies})$, where R is the number of requests and $N_{Proxies}$ the group size. Moreover, overhead traffic occurs at the same time as data traffic, resulting in more packet drops. Increasing traffic during congested periods can dramatically reduce cache performance

and interfere with network cross-traffic.

By contrast, the overhead of directory-based systems is not dependent upon number of requests, nor is it temporally linked to data transfers. Directory-based systems can propagate metadata at night when the network is quiescent. Consequently, directory-based discovery is better suited to the Web conditions of sparse sharing and cyclic load.

4.6.2 Client-initiated dissemination

Client-initiated dissemination is preferred to a server push approach for two reasons. First, sparse sharing makes it difficult for servers to predict where objects will be requested and should be cached. Secondly, the Web exhibits rapid popularity shifts and flash crowds. as evidenced by the massive request loads during the 1996 Atlanta Olympic games [7] and the 1997 Mars Pathfinder mission [36]. Because client-initiated dissemination caches objects based upon current requests pattern rather than upon server predictions, it adapts automatically to sudden or unexpected popularity shifts.

4.6.3 Direct delivery

Direct delivery transfers the object once instead of multiple times, therefore it results in lower latencies, fewer TCP/IP connections, and less network traffic than indirect delivery.

4.6.4 Flat cache organization

Cache misses for multi-level caches are expensive because each miss adds an HTTP connection and object transfer, which are costly operations on the Internet. Tewari, et al., [62] quantify this effect, showing that accessing an additional level of the hierarchy approximately doubles the average delivery time and dramatically increases the worst-case delays. This argues that proxy caches should be organized into a flat mesh for document delivery.

4.6.5 Periodic metadata propagation

The advantage of metadata propagation is that metadata can be sent separately from object delivery, thereby exploiting observed traffic patterns to reduce cooperation overhead. The diurnal nature of Internet traffic observed in Section 3.6.5 and the infrequency of cache replacement observed in the cache warmup investigation in Section 2.5.1 suggest that a substantial fraction of metadata could be transferred during quiet nighttime periods, supplemented by small updates during the more congested daytime hours. Cache digests [26] are a possible method for exchanging large amounts of metadata, while the metadata hierarchy of Tewari, et al., [62] and our lazy prefetching technique offer potential update mechanisms.

4.7 A proposed design: Server-directed proxy sharing

We have designed a cooperative Web caching protocol based upon the guidelines listed in the last section. Our design organizes proxy servers into a flat mesh, and uses metadata directories to share information between proxies. To exploit the structure of Web pages, the Web server sends metadata for linked objects with the initial HTML file, thereby allowing clients to obtain embedded images and other related objects from faster cache sites. Because of Web server participation, this design is known as *server-directed proxy sharing* (SDP). An implementation approach for SDP was tested by building transparent proxies that wrap around existing HTTP proxies and HTTP servers. SDP is described in detail in Appendix A.

4.8 Implications for viability analysis

Our design analysis concludes the best approach to cooperative cache design uses a flat mesh organization to deliver Web objects and metadata directories to separate the discovery and delivery functions. This separation also simplifies cache models because the cooperation overhead is independent of other factors. The viability analysis in the next chapter assumes a flat mesh with separate data and metadata delivery mechanisms, allowing us to use a flat delivery cost and vary cooperation overhead independently of the number of requests and object size.

Chapter 5

Viability Analysis

In this chapter we return to the fundamental question posed in the introduction: is cooperative Web caching viable? That is, can cooperative caching improve average user response time and, if so, how does the speedup depend upon cache overhead? To answer this question, we formulate an expression for the average speedup in response time in terms of hit rates, cache speed differential, cache overhead, and the efficiency of metadata propagation. Chapter 1 introduced these issues, and the intervening chapters provide experimental data for estimating their values. The analysis in this chapter uses the experimentally determined hit rates and cache speed differential to estimate an upper bound for speedup and to explore the trade-off between cache overhead and propagation efficiency. Results of the analysis enable us to determine the extent of cooperation speedup and the conditions for cooperation viability.

5.1 L3 cache hit rate

5.1.1 Real and ideal hit rates

We begin the viability analysis by estimating the maximum L3 hit rate for cooperative Web caches. The maximum L3 hit rate assumes cooperating caches have received all necessary metadata (i.e., perfect metadata propagation), therefore it depends only upon proxy sharing behavior and document cachability. Because cachability rules and workloads change over time, we estimate an ideal upper bound by assuming all documents are cachable. The term *real hit rate* refers to real document cachability, while an *ideal hit rate* assumes all documents are cachable. Thus the maximum real L3 hit rate assumes perfect metadata and uses

measured document cachability, while the maximum ideal L3 hit rate assumes perfect metadata and ideal cachability.

Our estimate of maximum L3 cache hit rates are derived from measurements of the NLANR caches reported in Chapter 2. Two corrections are required to convert the NLANR hit rates to L3 cooperative hit rates. First, ideal hit rates must be corrected for repeat requests. Chapter 2 points out that if all documents were cachable, the repeat requests currently seen by L3 caches would disappear because they would handled by browser and proxy caches. Therefore computation of ideal cache hit rate requires that repeat requests for uncachable documents be removed from the request stream.

Second, because our data is from hierarchical cache traces, we must correct for the omission of sibling hits. Hierarchical caches such as Squid [58] record hits only for requests received by the parent cache, which ignores sibling exchanges between the children. These sibling requests are recorded in the children's traces, not in the parent's trace. In this chapter, we estimate L2 sibling hit rates by examining L2 traces, and compute overall L3 sharing from the sibling and parent hit rates. The sibling hit rate correction applies to both real and ideal hit rates.

5.1.2 Correction for repeat requests

This section derives an expression for ideal hit rate that corrects for repeat requests.[1] Sibling hit corrections will be added in the next section. Let:

N = number of L3 cache requests

N_{CH} = number of L3 cache hits

N_{UH} = number of L3 cache hits for uncachable objects if they could be cached

N_{UR} = number of L3 repeat requests for uncachable objects

The real and corrected ideal hit rates, H_{Real} and H_{Ideal}, are defined as:

$$H_{Real} = \frac{N_{CH}}{N}$$

[1] where requests are successful GETs and cache hits are defined as in Section 2.3.1

$$H_{Ideal} = \frac{N_{CH} + N_{UH}}{N - N_{UR}}.$$

Converting from request counts to request fractions yields:

$$\begin{aligned} H_{Real} &= C_{Hit} \\ H_{Ideal} &= \frac{(\frac{N_{CH}}{N}) + (\frac{N_{UH}}{N})}{1 - (\frac{N_{UR}}{N})} \\ &= \frac{C_{Hit} + U_{Hit}}{1 - U_{Repeat}} \\ &= \frac{H_{Real} + U_{Hit}}{1 - U_{Repeat}}. \end{aligned} \tag{5.1}$$

C_{Hit} is the fraction of requests that are for cachable documents and are cache hits. U_{Hit} is the fraction of requests that are for uncachable documents and would be cache hits if these documents were cached. U_{Repeat} is the fraction of requests that are for uncachable documents and are repeat requests from the same client.

U_{Hit} can be determined directly from a cache trace if the trace period allows for adequate cache warming. However, the NLANR traces in our study require 4-7 days of cache warming but only preserve client IDs within a single day's trace (see Chapter 3). To compute U_{Hit}, we assume that after removing repeat requests, cachable and uncachable documents have the same sharing rate:

$$\frac{U_{Hit}}{U - U_{Repeat}} = \frac{C_{Hit}}{C} = \frac{H_{Real}}{C}. \tag{5.2}$$

where C is the total fraction of requests for cachable documents and U is the total fraction for uncachable documents. It follows that $C + U = 1$.

Applying the assumption in Eq. 5.2 to Eq. 5.1 produces an expression for ideal cache hit rate in terms of measured values provided in Tables 2.2 and 2.3:

$$H_{Ideal} = \frac{H_{Real} + H_{Real} \times (U - U_{Repeat})/C}{1 - U_{Repeat}}. \tag{5.3}$$

	Definition	*BO1*	*UC*
L2 Proxy clients		≈ 90	≈ 60
Request fractions			
C	Cachable objects	0.72	0.76
U	Uncachable objects	0.28	0.24
U_{Repeat}	Repeat requests for uncachable objects	0.20	0.18
Real hit rates			
H_{Real}	Measured L3 parent hit rate	0.30	0.24
$H_{3,Real}$	with correction for sibling hits	**0.34**	**0.28**
Ideal hit rates			
$H_{Ideal,All}$	Computed L3 hit rate assuming all documents are cachable	0.51	0.49
H_{Ideal}	with correction for repeat requests	0.42	0.32
$H_{3,Ideal}$	with corrections for repeats requests and sibling hits	**0.45**	**0.36**

Table 5.1: *Hit rate estimates for cooperative Web caching. Final estimates are $H_{3,Real}$ (current cachability constraints) and $H_{3,Ideal}$ (all documents cachable).*

Table 5.1 summarizes the computations and results, including corrections for sibling hit rates discussed in the next section. For BO1 and UC, the measured hit rates, H_{Real} from Table 2.2 are 0.30 and 0.24. Their uncorrected ideal hit rates, $H_{Ideal,All}$, are 0.51 and 0.49, respectively. Correcting for repeat requests using Eq. 5.3 lowers their ideal hit rates to 0.42 (BO1) and 0.32 (UC). Based upon these measurements, repeat requests appear to be responsible for approximately one-half to two-thirds of the difference between real and uncorrected ideal hit rates. *These results show current L3 hit rates are much closer to the upper bound than previously believed* [68].

5.1.3 Correction for sibling hits in hierarchical caches

Real and ideal hit rates for hierarchical caches measure the performance of the parent cache, but do not reflect the total hit rate because they do not include sibling cache hits. In hierarchical caches, the child sends queries to its siblings and an echo request to its parent [16]. Sibling caches respond only if they have a cached copy of the object. If a sibling response arrives before the parent's response, the child cache requests the object from the sibling. In such cases, the parent never sees the request, and it does not appear as either a hit or a miss in the parent's trace. As a result, the parent's trace does not reflect all the sharing between

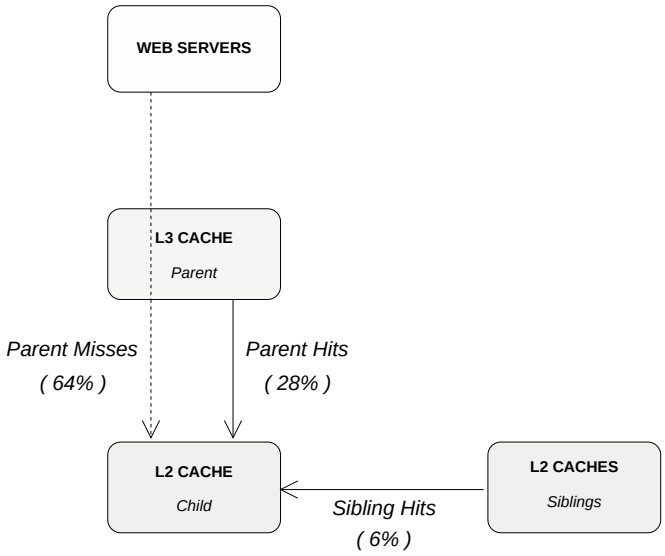

Figure 5.1: *Sibling hit contribution to L3 hit rates. Values are for NLANR BO1 cache.*

its children. A more accurate estimate of sharing must include both parent and sibling requests, as shown in Figure 5.1. In general, the maximum hit rate for proxy cooperation, H_3 can be computed from hierarchical L2 and L3 cache traces.

$$H_3 = \frac{\textit{L3 Parent hits} + \textit{L2 Sibling hits}}{\textit{L3 Parent hits} + \textit{L3 Parent misses} + \textit{L2 Sibling hits}}.$$

Sibling misses are considered negligible because siblings only respond to queries for objects in their cache.

Although L2 sibling hits are not available in the parent's trace, they can be obtained from the *children's* traces. Appendix B develops the following expression for computing the L3 hit rate from hierarchical traces:

$$H_3 = \frac{H_{3,Parent} + S_3/P_3}{1 + S_3/P_3} \tag{5.4}$$

where H_3 = total hit rate for the L3 level

$H_{3,Parent}$ = hit rate at the L3 parent cache

P_3 = fraction of L2 requests sent to the parent cache

S_3 = fraction of L2 requests sent to sibling caches.

Using reported data for two L2 caches [42][64], we measured median L2 sibling hit rates from 0.06 to 0.08 and ratios of sibling to parent requests that ranged from 0.06 to 0.16. The analysis that follows uses the more conservative 0.06 estimate. By comparison, L3 sibling hit rates for NLANR caches appear to hover around 0.03 to 0.05, with a sibling/parent request ratio of approximately 0.04 to 0.08.

5.1.4 Final estimate of maximum L3 hit rate

Substituting values from Table 5.1 and using 0.06 for the sibling/parent ratio in Eq. 5.4 leads to our final estimate of maximum cooperative Web cache hit rates for real and ideal cachability. Our final estimate is 0.28 to 0.34 for maximum real L3 hit rate, and 0.36 to 0.45 for maximum ideal L3 hit rate. Results are summarized in Table 5.1 and used in the remainder of the chapter.

5.2 Average speedup for a flat mesh

This section develops estimates for the speedup in average user response time offered by cooperative Web caches in a flat mesh architecture. We begin by computing the upper bound from the L3 hit rate estimates, then apply the measured speed differentials from Chapter 3 to compute an attainable speedup. In developing the speedup expression, we consider only browser cache misses and remote requests. That is, we ignore user-initiated reloads and downloads from Web servers on the local LAN. We assume local computation time and local communication between the LAN proxy and the browser are negligible compared to remote network transfers.

Speedup, S, is defined as the ratio of task time without an improvement to task time with the improvement:

$$S = \frac{T_{without}}{T_{with}} \tag{5.5}$$

Let H_2 be the L2 hit rate at the local proxy cache, T_S be the expected time for the proxy to retrieve the document from a remote Web server, and T_C be the expected time to retrieve the document from a cooperative Web cache. Without L3 caching, all proxy misses are sent to the origin Web servers and user response time,

$T_{without}$, is

$$T_{without} = (1 - H_2)T_S. \tag{5.6}$$

With L3 caching, the fraction of proxy requests sent to cooperating caches is bounded by H_3. This bound reflects perfect metadata; that is, proxies know the location of all cached objects. The remaining $1 - H_3$ proxy requests are sent directly to the origin Web servers. Network delays may change with cooperation because other proxies are downloading Web pages from different sites, which changes network traffic patterns. We denote the expected server and cache response times in the presence of cooperation cross-traffic as T'_S and T'_C, respectively. Using these terms, user response time with proxy cooperation, T_{with}, is expressed as

$$\begin{aligned} T_{with} &= T_O + (1 - H_2)[(1 - H_3)T'_S + H_3T'_C] \\ &= T_O + (1 - H_2)[T'_S - H_3(T'_S - T'_C)] \end{aligned} \tag{5.7}$$

where T_O is the request overhead imposed by cache management. Piggybacked metadata adds additional overhead associated with cache hits, but this effect is not considered here. As a first order approximation, we also ignore the changes in contention traffic by assuming $T'_S = T_S$ and $T'_C = T_C$. Under these assumptions, the average speedup in user response time for cooperative caching with perfect metadata is

$$\begin{aligned} \overline{S} &= \frac{T_{without}}{T_{with}} \\ &= \frac{(1 - H_2)T_S}{T_O + (1 - H_2)[T_S - H_3(T_S - T_C)]} \\ &= \frac{1}{1 + \left(\frac{1}{1-H_2}\right)\left(\frac{T_O}{T_S}\right) - H_3\left(\frac{T_S - T_C}{T_S}\right)}. \end{aligned} \tag{5.8}$$

5.2.1 Upper bound for speedup

The upper bound on average speedup occurs when caches are infinitely faster than Web servers ($T_S >> T_C$) and there is no cache management overhead ($T_O = 0$). For this case, speedup expression 5.8 reduces to

$$\overline{S}_{max} = \frac{1}{1 - H_3}. \tag{5.9}$$

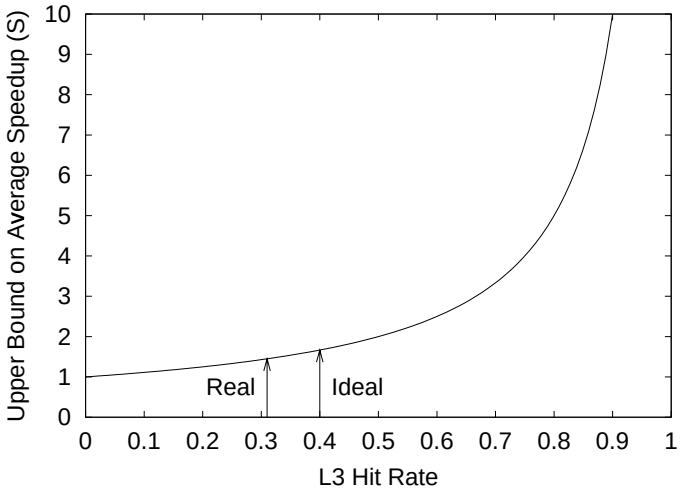

Figure 5.2: *Upper bound on average speedup in user response time for various L3 cache hit rates.*

As illustrated in Figure 5.2, average user response time improves slowly for L3 hit rates under 0.5. At a hit rate of 0.5 the maximum improvement in average user response is a speedup of 2; that is, caching is only twice as fast as no caching. Beyond hit rates of 0.5, speedup improves dramatically, asymptoting as hit rates approach 100%. Section 5.1 shows that under current cachability constraints, the L3 hit rate is 0.28 for UC and 0.34 for BO1, with a geometric mean of 0.31. For a hit rate of 0.31, the average speedup in response time cannot exceed 1.4. Stated differently, L3 caching reduces average users response time by a maximum of 31%. Removing all cachability constraints improves the mean L3 hit rate to 0.40, which raises the upper bound on speedup to 1.7 and the maximum reduction in average response time to 40%.

5.2.2 Attainable speedup

The site selection results described in Chapter 3 allows us to estimate attainable speedup for cooperative caches.[2] For a set of 10 sites, daytime downloads are typically 3 – 5 times faster using dynamic probes compared to random selection, with a geometric mean of 3.32. Because random selection models a single Web server and probe selection corresponds to a set of remote caches, the experimental ratio provides an estimate of the attainable cache speed differential for a set of remote caches.

[2]Results from Chapter 3 show that the daytime bandwidths are approximately independent of file size and that connect and latency times are small compared to read time. Therefore the ratio T_S/T_C is approximately independent of file size.

	L3 Hit Rate	Average Speedup		Response Time Decrease	
		Probe	*Upper Bound*	*Probe*	*Upper Bound*
Real Cachability					
UC	0.28	1.2	1.4	20%	28%
BO1	0.34	1.3	1.5	24%	34%
Geom. Mean	**0.31**	**1.3**	**1.4**	**22%**	**31%**
Ideal Cachability					
UC	0.36	1.3	1.6	25%	36%
BO1	0.45	1.5	1.8	31%	45%
Geom. Mean	**0.40**	**1.4**	**1.7**	**28%**	**40%**

Table 5.2: *Average speedups and reductions in user response time for L3 cooperative caching with no overhead and perfect propagation. Upper bounds assume caches are infinitely faster than Web servers.*

Combining the experimental server/cache response ratio of 3.32 with the real L3 cache hit rate of 0.31 and neglecting cache overhead, the average speedups for cooperative caching computed from Eq. 5.8 is 1.3 as compared to an upper bound of of 1.4. Assuming ideal cachability, the geometric mean for L3 hit rate is 0.40, for which the estimated attainable speedup is 1.4 with an upper bound of 1.7. The closeness of attainable speedup estimates to their upper bounds indicates that further improvements in client-side selection methods are of limited use to cooperative caches. Table 5.2 summarizes these speedup results.

5.3 Cache overhead and propagation efficiency

The preceding analyses assume 1) negligible cache overhead, and 2) perfect metadata. The first assumption holds if overhead communication does not affect user response time. For example, cache digests [26] or the compressed URL forwarding tables [45] can be transferred during quiescent nighttime periods. However, cache contents can change between these transfers, leading to incomplete metadata at the proxy caches and lower L3 hit rates. On the other extreme, caches maintain perfect metadata by propagating information concurrent with the requests, thereby maximizing hit rate at the expense of high overhead. In-between are designs which propagate large cache digests at night, and send small update messages during the day. The intermediate case incurs some overhead and has a less-than-optimal hit rate, but may result in the best overall performance.

5.3.1 Propagation efficiency

We model propagation efficiency by adding a parameter $\mathcal{E}$ to represent the fraction of metadata for potential cache hits that was propagated to the proxies. Values of $\mathcal{E}$ vary from 0 to 1, where $\mathcal{E} = 0$ indicates no useful information was propagated and $\mathcal{E} = 1$ is perfect propagation. Multiplying the maximum L3 hit rate by $\mathcal{E}$ yields the *effective hit rate,* $\mathcal{E}H_3$, that is realized by a specific cache design with a given workload. Introducing this effective hit rate into the expression for average speedup in Eq. 5.8 produces an expression for speedup in terms of cache overhead and propagation efficiency:

$$\overline{S} \;=\; \frac{1}{1 \;+\; \left(\frac{1}{1-H_2}\right)\left(\frac{T_O}{T_S}\right) \;-\; \mathcal{E}H_3\left(\frac{T_S-T_C}{T_S}\right)}. \tag{5.10}$$

5.3.2 Cache overhead

The overhead term, T_O, refers to cache overhead that affects user response time. Local lookup time is assumed negligible compared to network time, therefore contributions to T_O stem from overhead communication associated with the request and from overhead communication of other requests that affect response time because of added network congestion or server load. For example, Squid ICP queries [65] would be included, as would piggybacked metadata associated with queries [22]. Overhead associated with replies is not accounted for in this analysis (see Section 5.2). Rewriting Eq. 5.10 to express the overhead as a ratio of T_O/T_C yields

$$\overline{S} \;=\; \frac{1}{1 \;+\; \left(\frac{1}{1-H_2}\right)\left(\frac{T_C}{T_S}\right)\left(\frac{T_O}{T_C}\right) \;-\; \mathcal{E}H_3\left(\frac{T_S-T_C}{T_S}\right)}. \tag{5.11}$$

5.3.3 Speedup equation with parameter estimates

The speedup expression in Eq. 5.11 allows us to determine if cooperative caching is viable for a given set of parameters. Parameters for hit rates and cache speed ratio are independent of the cooperation mechanism, while $\mathcal{E}$ and T_O/T_C depend upon the cache design. Table 5.3 summarizes our estimates for H_2, H_3, and T_S/T_C, where H_2 is taken from reported local proxy hit rates [10][21][41][2][53][68], and H_3 and T_S/T_C are taken from this work. For these values, the average speedup for a cooperative Web cache design with

Term	*Estimate*	*Source*
H_2	0.40	Approximate median of reported values
$H_{3,Real}$	0.31	Geometric mean of real L3 hit rates from Table 5.1
$H_{3,Ideal}$	0.40	Geometric mean of ideal L3 hit rates from Table 5.1
T_S/T_C	3.32	Geometric mean for A&M, UH, and UTSA from Table 3.5

Table 5.3: *Summary of parameters for computing cooperation speedup (Eq. 5.10).*

efficiency $\mathcal{E}$ and overhead T_O/T_C is

$$\begin{aligned} \overline{S} &= \frac{1}{1+\left(\frac{1}{1-0.40}\right)\left(\frac{1}{3.32}\right)\left(\frac{T_O}{T_C}\right)-\mathcal{E}(0.31)(1-0.30))} \\ &= \frac{1}{1+0.502\left(\frac{T_O}{T_C}\right)-0.217\mathcal{E}}. \end{aligned} \tag{5.12}$$

5.4 Viability condition

We consider cooperative Web caching viable if there is no increase in average response time. This definition is equivalent to requiring average speedup be at least 1:

$$\overline{S} = \frac{1}{1+\left(\frac{1}{1-H_2}\right)\left(\frac{T_C}{T_S}\right)\left(\frac{T_O}{T_C}\right)-\mathcal{E}H_3\left(\frac{T_S-T_C}{T_S}\right)} \geq 1. \tag{5.13}$$

Simplifying Eq. 5.13 produces the viability condition

$$\frac{T_O}{T_C} \leq \mathcal{E}\ H_3\ (1-H_2)\ \left(\frac{T_S}{T_C}-1\right). \tag{5.14}$$

The viability condition expresses the limit on cache overhead for a given propagation efficiency, hit rate, and speed differential. Using parameter estimates from Table 5.3, cooperation for real cachability constraints is viable when

$$\frac{T_O}{T_C} \leq \mathcal{E}\ (0.31)\ (1-0.40)\ (3.32-1) = 0.43\ \mathcal{E}. \tag{5.15}$$

For ideal cachability, the viability condition is

$$\frac{T_O}{T_C} \leq \mathcal{E}\ (0.40)\ (1-0.40)\ (3.32-1) = 0.56\ \mathcal{E}. \tag{5.16}$$

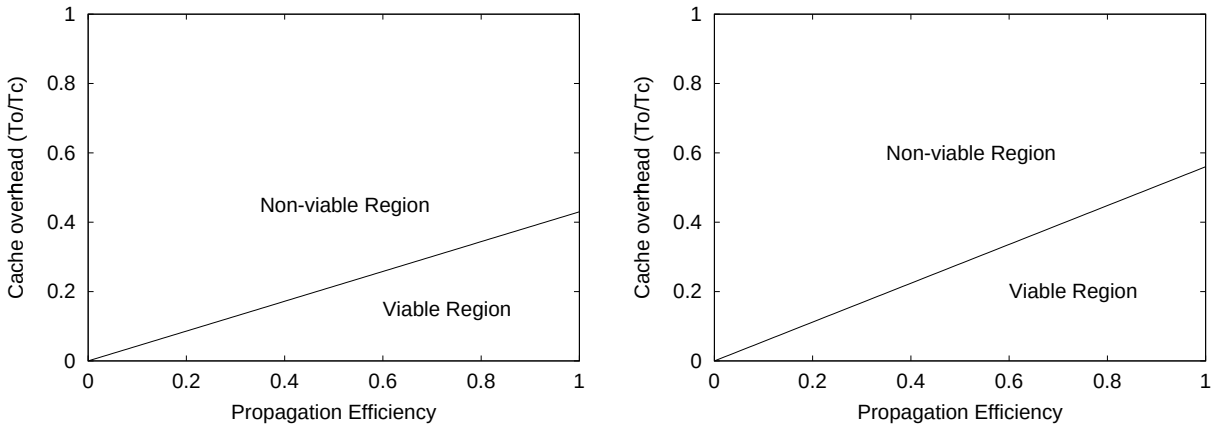

Figure 5.3: *Viability region for cooperative Web caches. Real cachability is shown on the left and ideal cachability on the right. Propagation efficiency $\mathcal{E}$ is defined in the text. Cache overhead T_O/T_C is the portion of cache overhead that occurs concurrently with object transfers and affects user response time.*

Figure 5.3 plots the viability conditions for real and ideal cachability, highlighting regions where combinations of efficiency and overhead are viable, and the non-viable regions where cooperative caching degrades rather than improves average response time.

5.4.1 Relative importance of overhead and efficiency

The plots show that viability depends more heavily upon overhead than upon propagation efficiency.[3] The viable region includes all values of $\mathcal{E}$, but limits overhead to a smaller range of values. Moreover, if a cache design with parameters $\mathcal{E}$ and T_O/T_C is not viable, then reducing overhead moves that design to the viable region faster than improving efficiency. For example, consider the point in the real cachability plot where $\mathcal{E} = 0.5$ and $T_O/T_C = 0.5$. The shortest distance from point (0.5, 0.5) to the viability line lies along the perpendicular to the viability line. This nearest point is at (0.60, 0.26), requiring a move of +0.10 along the efficiency axis and -0.24 along the overhead axis. In other words, the shortest distance from a non-viable point to the viable region involves more improvement in overhead than in efficiency. Consequently, cache designs should focus more on reducing overhead than upon ensuring better metadata.

[3]This occurs because the slope of the viability boundary is less than 1. It should also be noted that $\mathcal{E}$ is a nonlinear function of the amount of metadata propagated because propagation of information on popular documents has more of an effect than propagation of information on unpopular documents.

5.4.2 Maximum overhead

Maximum overhead for viable proxy cooperation is allowed when $\mathcal{E} = 1$ and is expressed as:

$$\left(\frac{T_O}{T_C}\right)_{Max} = H_3\,(1 - H_2)\,\left(\frac{T_S}{T_C} - 1\right). \tag{5.17}$$

As shown in Figure 5.3 and Eq. 5.15, the maximum overhead cannot exceed 43% of the cache response time for real cachability, and 56% of the cache response time for ideal cachability.

5.5 Isospeedup

5.5.1 Definition of isospeedup

Let us define *isospeedup* as combinations of efficiency and overhead that produce the same speedup. The isospeedup expression is derived by rearranging Eq. 5.11 and simplifying:

$$\frac{T_O}{T_C} = \mathcal{E}\ H_3\ (1 - H_2)(\tfrac{T_S}{T_C} - 1) \ - \ (1 - \tfrac{1}{S})(1 - H_2)(\tfrac{T_S}{T_C}). \tag{5.18}$$

5.5.2 Isospeedup plots

Figure 5.4 shows isospeedup plots for real and ideal cachability, using parameters from Table 5.3. Isospeedup curves for different values of $\overline{S}$ have the same slope, but different intercepts. Conversely, the L3 hit rate affects only the slope, therefore real and ideal cachability plots show different slopes but the same intercepts.

5.5.3 Effect of L2 hit rate

Hit rates for LAN proxy caches, H_2, range from around 30% to 50%. Because L3 caches only receive L2 misses, the L2 hit rate affects L3 viability and speedup. Examining the speedup expression in Eq. 5.18, we observe that the LAN cache hit rate affects *both* the slope and intercept of the isospeedup line. Figure 5.5 shows isospeedup plots for L2 hit rates of 30%, 50%, and 60% for both real and ideal cachability. Interestingly, increasing the L2 hit rate shrinks the region of viability for cooperative caches, but allows for higher speedup.

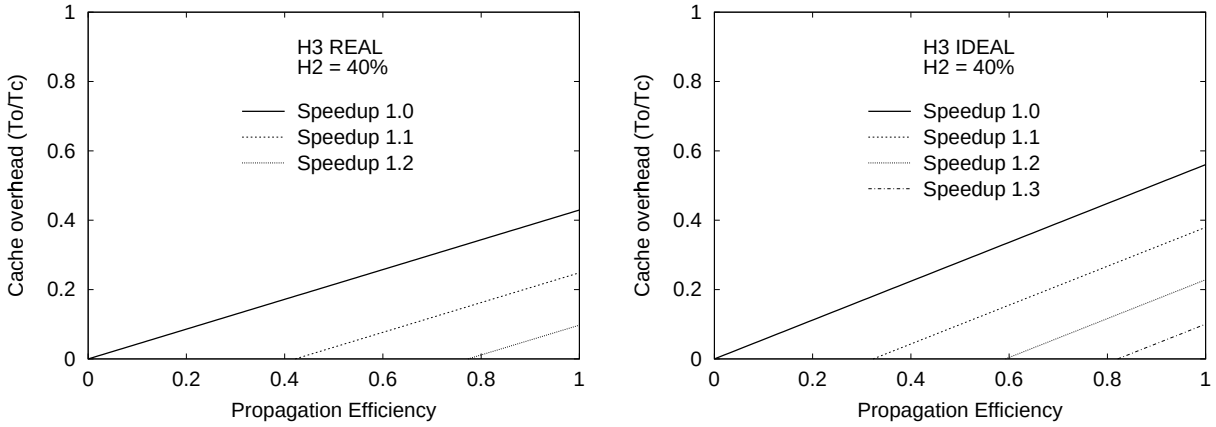

Figure 5.4: *Isospeedup curves for cooperative Web caches. Real cachability is shown on the left and ideal cachability on the right. Hit rates and cache speed differential are from Table 5.3, including the L2 proxy hit rate of 40%.*

5.6 Reduction of long delays

If we were to answer the viability question solely on the basis of average speedup or average delay reduction, it could be argued that cooperative Web caching does not offer significant performance improvement. However, an average value only partially characterizes a distribution. Distribution with long tails and large skews can have the same mean as symmetric, low variance distributions. Although cooperative caching may not have a dramatic effect on average response time, users would consider cooperation to be a valuable performance enhancement if it eliminates pathological delays or reduces the variability in response time.

Chapter 3 establishes that a set of remote Web caches can indeed reduce response time variability and number of long delays (see the probability distributions in Figure 3.3 and CDFs in Figure 3.6). This reduction is difficult to quantify because no single number characterizes the tail of a distribution. That is, what constitutes a long delay? To compare tail behavior without using arbitrary definitions, we examine 1) maximum delays for servers and caches, and 2) inverse percentiles for the response time CDFs.

5.6.1 Maximum delay

Table 5.4 reports median and maximum response times for clients in the site selection study. While remote cache accesses are usually only a few seconds longer than those of the Web server, the maximum delays are much longer for servers than for caches. The most noticeable example is A&M, where the longest server

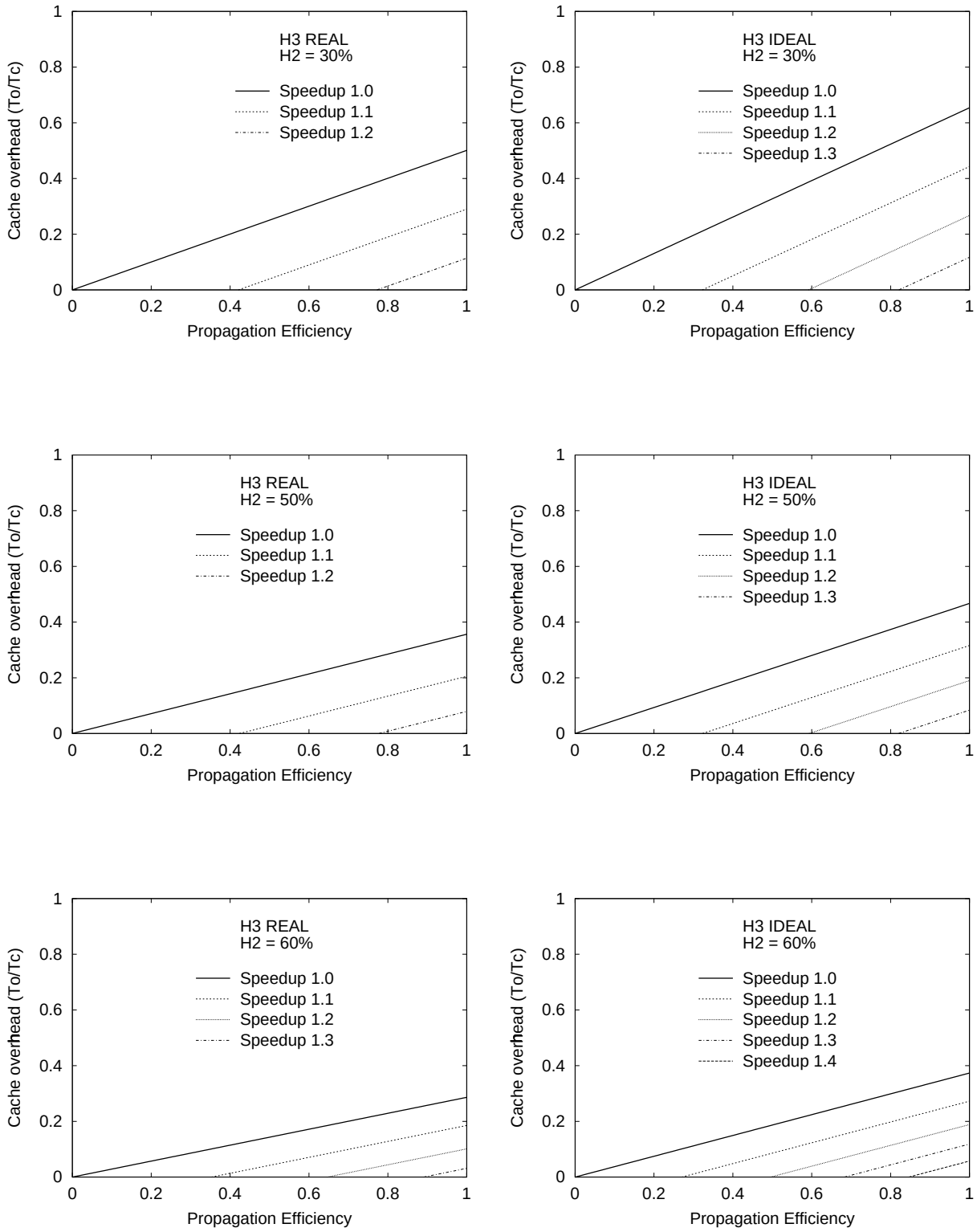

Figure 5.5: *Isospeedup curves for cooperative Web caches using L2 proxy hit rates of 30% (top), 50% (middle), and 60% (bottom). Real cachability is shown on the left and ideal cachability on the right. L3 hit rates and cache speed differential are from Table 5.3.*

	Median Response Time		Maximum Response Time	
	Server	*Cache*	*Server*	*Cache*
A&M	6 sec	1 sec	18 min	2 min
UH	10 sec	4 sec	11 min	6 min
UTSA	30 sec	11 sec	35 min	16 min

Table 5.4: *Median and maximum response times for Web servers (Random) and remote caches (Probe).*

		Response Time Percentile					
		25th	*50th*	*75th*	*90th*	*95th*	*99th*
A&M	Response time	>1 sec	>5 sec	>18 sec	>55 sec	>108 sec	>197 sec
	Random	0.75	0.50	0.25	0.10	0.05	0.01
	Probe	0.33	0.17	0.07	0.01	0	0
	Random – Probe	0.42	0.33	0.18	0.09	0.05	0.01
UH	Response time	>3 sec	>10 sec	>24 sec	>31 sec	>71 sec	>461 sec
	Random	0.75	0.50	0.25	0.10	0.05	0.01
	Probe	0.54	0.29	0.15	0.05	0.02	0
	Random – Probe	0.21	0.21	0.10	0.05	0.03	0.01
UTSA	Response time	>13 sec	>30 sec	>66 sec	>140 sec	>190 sec	>1317 sec
	Random	0.75	0.50	0.25	0.10	0.05	0.01
	Probe	0.44	0.20	0.07	0.04	0.02	0
	Random – Probe	0.31	0.30	0.18	0.06	0.03	0.01

Table 5.5: *Inverse percentiles ($1 - CDF$) of response time for servers (Random) and remote caches (Probe).*

delay is 18 minutes compared to 2 minutes for caches. Similarly, remote caches reduce the longest wait by 5 minutes at UH and 19 minutes at UTSA.

5.6.2 Distribution percentiles

Table 5.5 compares distribution tails for server and cache response times by examining their cumulative distribution functions (CDF) at responses times corresponding to the 25th, 50th, 75th, 90th, 95th and 99th percentile of the server distribution. To examine the tail, we convert CDF fractions into inverse percentiles, $1 - \mathrm{CDF}(T)$. Whereas $\mathrm{CDF}(T)$ is the fraction of responses $\leq T$, the inverse percentile is the fraction of responses $> T$. The difference between inverse percentiles at a time T is the fraction of delays $> T$ that occur for the server but not the cache.

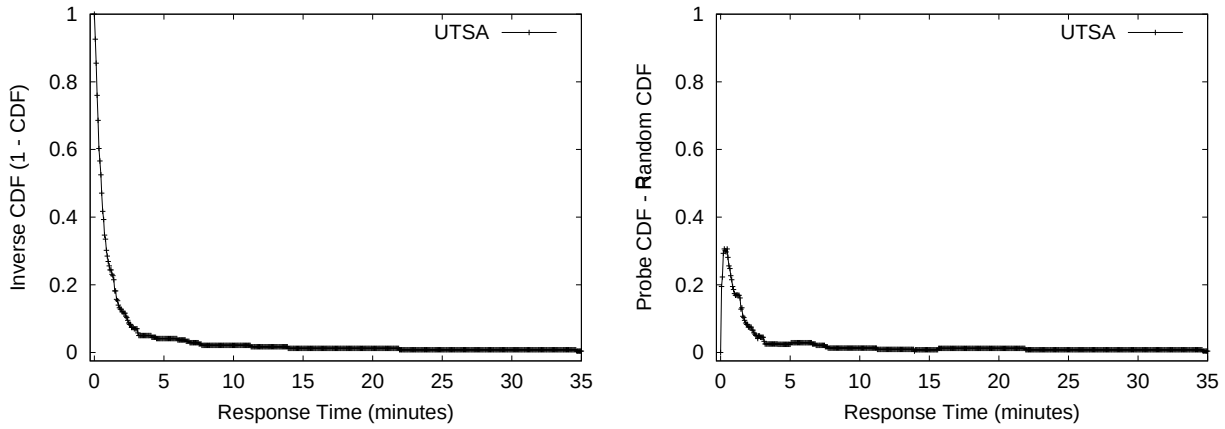

Figure 5.6: *Fraction of server responses $> T$ (left), and fraction eliminated when objects are retrieved from remote caches (right) for the UTSA client.*

At UTSA, the median response time for random Web servers is 30 seconds. However, 10% of the responses exceed 140 seconds and the slowest 1% are between 22 and 35 *minutes.* Using remote caches, only 4% of the responses exceed 140 seconds, and no response takes over 15 minutes. In comparison, cache retrievals result in 60% fewer delays over 140 seconds, 80% fewer delays over 190 seconds, and 100% fewer delays over 16 minutes. Figure 5.6 captures this improvement by plotting the fraction of delays $> T$ and plotting the difference in delays $> T$. These plots illustrate the degree to which remote caches reduce response time variability and number of long delays.

5.6.3 Web pages

If we consider an entire Web page rather than single Web objects, we discover that *the probability for a long page delay is higher than the probability of delay for a single object.* When an HTML file contains multiple embedded objects such as images, Web browsers download 4 objects at a time. The browser issues the first 4 requests immediately, then, if there are more than 4 embedded objects, issues additional requests as soon as earlier replies finish. If all objects have the same download time t, then, ignoring a small startup offset, 4 objects can be retrieved in time t. In this case, the download time for a Web page with n embedded objects is reduced from nt to approximately $nt/4$ (see footnote[4]). However, if one object has a much longer delay,

[4]Exact expression is $(\lfloor n/4 \rfloor + n \bmod 4)\ t$

t_{max}, then download time for the entire Web page is approximately t_{max}. Thus a long delay practically eliminates the benefit of overlapping downloads.

As there are 3.2 embedded images per page (Table 4.2), every third Web page contains, on the average, an object in the slowest 10% of responses. Likewise, every 31st page contains an object in the slowest 1% of the distribution. If all objects have the same download times, then at UTSA where median server delay is 30 seconds, the entire Web page would require 30 seconds (assuming 50 KB objects). For the real server distribution, the expected delay at UTSA increases to 2.3 minutes for every third Web page. Less frequent but more frustrating is the 22 minute delay for every 31st Web page. Now assume the Web page is obtained from a set of remote caches. For UTSA, the cache delay for every third Web page is under 1 minute. For delays exceeding 2.3 minutes, the probability drops from $\frac{1}{3}$ of the Web pages to $\frac{1}{8}$ of the Web pages. To compute this probability, we read from Table 5.5 that 4% of objects are delayed more than 2.3 minutes. The page probability is $3.2 \times 0.04 = 0.13$, or one such delay every 8th Web page.

From this example, we observe that because Web browsers retrieve page objects in parallel, long delays have a greater impact on Web pages than upon single objects. Consequently, the performance improvement offered by Web caches is larger when viewed from the perspective of Web pages.

5.7 Viability summary and discussion

• Is cooperative Web caching viable?

Yes, providing the cache design parameters stay within overhead and efficiency bounds of the viability condition in Section 5.4:

$$\frac{T_O}{T_C} \;\leq\; \mathcal{E}\; H_3\; (1 - H_2)\; \left(\frac{T_S}{T_C} - 1\right).$$

Using estimates derived in this dissertation for hit rate and cache speed differential, cooperative Web caching is viable when the ratio of overhead to efficiency is less than 0.43:

$$\frac{T_O}{T_C} \;\leq\; 0.43\; \mathcal{E}$$

• What is the expected average speedup in user response time?

Actual speedup depends upon the cooperation design. We estimate the average speedup in user response time would be under 1.3, reducing the average user response time by at most 23%. Allowing all documents to be cached would extend the maximum speedup to 1.4 and reduce the average wait by 29%.

- **Does cooperative Web caching offer other performance benefits?**

Yes, a major advantage of cooperative Web caching is its ability to reduce the variability in response time and the number of long delays.

- **How does cooperation overhead affect other network traffic?**

Cooperative caching can reduce loads on congested routers and servers by 20% – 25% (Table 5.2) Because dynamic probing identifies and avoids congested network routes, it lessens congestion at busy routers and improves conditions for network cross-traffic. Further, if Web servers and caches add load ports as suggested in Chapter 3, then dynamic probes also provide a mechanism to balance load across servers and caches. Traffic and load balancing would improve unrelated traffic by relieving congested routers and busy servers, thereby speeding up the remaining traffic. On the negative side, cooperation adds overhead traffic. However, Web objects remain in L3 caches for several days (Section 2.5.1) and change infrequently (Table 4.1). Consequently, a substantial fraction of metadata can be distributed during quiescent nighttime periods. Taking advantage of the diurnal nature of Internet load substantially reduces the effect of cooperation overhead on response time and network cross-traffic.

- **Is cooperative Web caching practical?**

Yes, although cooperation offers little speedup in user response time, its ability to reduce pathological delays and redistribute network and server load are valuable. Using the design guidelines in Chapter 4, cooperation does not require changes in either Web browsers or Web servers. In addition, cooperation does not require a special set of caches, nor does it entail added security risks. (Section A.6 discusses these issues in greater detail.) Overall, our analysis shows that cooperative Web caching is viable and practical, but that most of its benefits are dispersed across the network traffic rather than targeted at an individual user. Consequently, the benefits of cooperations grow with the number of participants: the more traffic that avoids congestion, the faster everyone's response time.

Chapter 6

Conclusions

6.1 Overview and perspective

6.1.1 Misleading notions

Before beginning this work, we believed cooperative Web caching would provide a valuable part of the Internet's infrastructure. We based this belief on the commonly held notion that there is abundant potential sharing across Internet users. This notion took hold in the research community because numerous studies reported a small fraction of Web documents receive a large fraction of the requests. Observed request distributions were successfully modeled by Zipf distributions with large α's, leading to acceptance of the Zipf α as a measure of potential cache success. Studies assumed that because a document is requested many times, it is requested by many users and therefore offers an opportunity for cache hits. The total number of duplicate requests was viewed as the upper bound on cache hit rate: if all documents could be cached, then all duplicate requests would be cache hits.

Accepting this established view, we initially attempted to analyze cooperation viability using an analytical model based upon the Zipf distribution. We noticed other researchers beginning to apply Zipf models to Web caching, although none had (or has yet) succeeded in predicting measured hit rates. When we attempted to reconcile our model's predictions with actual cache traces, we discovered the reason: request popularity does not reflect user popularity. This is the fundamental flaw in the perceived link between observed Zipf distributions and cache hit rates. A few studies remark on the large number of repeat requests, but do not perceive their relationship to Zipf parameters and hit rates models. Our primary insight was to recognize

that repeat requests will almost never be proxy cache hits and therefore should not be used to characterize potential sharing. If the document were cachable, then repeated requests for that document would be handled by the browser or lower level proxy and would never be seen by upper level caches. This serendipitous discovery and related insight led to one of our most important results, that being the determination of dramatically lower Zipf parameters and lower estimates of ideal cache hit rates.

6.1.2 Dominating effect of the network

Lower L3 hit rates would seem to bode ill for the viability and success of cooperative caching. However, in the site selection experiment we made another important observation: most delay in downloading Web documents is caused by congested network routers rather than overloaded Web servers. A primary benefit of cooperative caching lies in its ability to identify and use less congested network routes. Our insight here is the observation that selecting less congested routes has a larger effect on variability and number of long delays than it does on the expected wait time, and that congestion avoidance improves overall network performance.

6.1.3 The real benefit of cooperation

Should cooperative caching be part of the Internet infrastructure? We believe the answer is a qualified yes, but not for the reasons most often put forth. Our analysis predicts cooperative Web caching is viable for some combinations of overhead and efficiency, but offers only a modest direct improvement in average user response time. However, within the viability bounds, we believe cooperative caching has two major benefits, both of which are rooted in the ability of dynamic probes to avoid congestion. The direct benefit is the reduction in the distribution tail for response time, resulting in less variability and fewer long delays. The indirect benefit is the effect congestion avoidance has on other network traffic. Shifting traffic to less congested routes reduces the load on overburdened routers and frees up space in their queues. As a result, the remaining traffic experiences fewer dropped packets and faster communication. Here cooperative Web caching benefits users not because of their own use, but because of use by others.

One argument for cooperative Web caching postulates that increasing the number of clients increases

Contribution	*Section*
Exposure of the effect of repeat request	Sec. 1.5.1 Chapt. 2
Correction of Zipf parameters and ideal hit rates	Sec. 1.5.1 Chapt. 2
Techniques for correcting cache traces	Sec. 1.5.2 Chapt. 2
Evaluation of site selection methods	Sec. 1.5.3 Chapt. 3
Taxonomy for cooperative Web caching	Sec. 1.5.4 Chapt. 4
Viability analysis for cooperative Web caching	Sec. 1.5.5 Chapt. 5
SDP design for cooperative Web caching	Sec. 1.5.6 Appendix A
Standard terms and metrics	Sec. 1.5.7 Chapts. 2 and 3
`httpget` *and* `tcping`	Sec. 1.5.8, Appendix C

Table 6.1: *Dissertation contributions.*

cache sharing and therefore cache hit rates. Research is beginning to suggest this effect is limited. In our view, cache hit rate is not the correct focus. As suggested above, we believe increasing the number of users can improve performance because more users imply more congestion avoidance, and more congestion avoidance implies faster communication for all.

6.2 List of contributions

The sharing analysis, site selection experiment, and design study produced a number of results outside the central issues of cooperation viability and benefit. Major results and their implications are discussed at the end of each Chapter. The contributions of this dissertation are explained in Section 1.5. Contributions appear in the Introduction to provide focus for the in-depth description of individual studies that follow. The contributions are summarized in Table 6.1 and the reader is referred to Section 1.5 for details.

6.3 Viability answers

In the Introduction, we pose four fundamental questions addressing the viability of cooperative Web caching. The conclusions in Chapter 5 discuss these questions in detail and explain the analysis behind our answers. Here we revisit these questions and summarize the answers.

1. Is there enough sharing between Web proxy servers to make cooperation viable?

Yes, but barely. We estimate the maximum hit rate for cooperative caches is approximately 0.3 for real cachability constraints, with an upper bound of 0.4 for ideal cachability. These hit rates allow Web caching to be viable under limited combinations of cache overhead and efficiency, but require care in the cache designs to stay within the viability limits.

2. Can a set of remote caches offer faster response times than Web servers?

Yes, definitely. Using dynamic probes and a set of 10 remote candidate sites, retrieving 50 KB objects from caches is typically 3.3 times faster than downloading it from the Web server. The speed differential increases with the probability that a cache resides on the regional network; however, we did not explore how the speed differential depends upon number of candidate sites (i.e. number of cached copies).

3. What are the basic features of cooperative cache designs that best match user sharing and Internet traffic patterns?

The single most important feature in a cache design is direct delivery between the cache and client. This argues for a flat mesh organization over hierarchical or multi-level designs. Maintaining metadata in local databases is a viable method for essentially eliminating overhead associated with discovery, especially as metadata can be propagated during quiescent network periods. Options abound for propagating metadata. For example, caches may maintain a second hierarchical organization for metadata, or can use multicast. The key is to separate metadata propagation from object discovery and delivery.

4. Will cooperative caching speed up response time? If so, what is the relationship between speedup and cache overhead, and what is the maximum overhead for which cooperation is still viable?

Cooperative Web caching can speedup average response time, but the improvement is not dramatic. Our analysis shows that the 0.3 hit rate for real cachability limits speedup in average user response time to 1.4 or, equivalently, to a 31% reduction in response time. Realistically, speedups will be substantially lower due to cache overhead, incomplete metadata, and a finite speed differential between remote caches and Web servers. However, cooperation does substantially reduce both the variability in response times and the number of long delays. The maximum overhead for our estimates of hit rate and cache speed differential is 43% of the cache response time. This result is based on a definition of overhead that includes increased network congestion

caused by other cache requests. In future work we plan to investigate the effect of various components of cache overhead by modeling overhead as the ratio of metadata bytes to object bytes. While different models would be required for specific forms of cache organizations and metadata propagation methods, these models would provide a concrete physical interpretation of cache overhead and would allow comparison between different cache designs.

6.4 Critical factors

6.4.1 Network congestion is more important than server load

Two critical factors in the performance of cooperative Web caching are network effects and overhead that occurs during object discovery or delivery. Our site selection experiment shows that, under normal circumstances, the difference between network routes outweighs any difference in server load. These results suggest research should concentrate on factors that reduce communication rather than factors that reduce server load.

6.4.2 Overhead is more important than efficiency

The viability analysis examines how the effect of lower hit rates from incomplete metadata compares to the effect of added communication cost for cache overhead. We find cache performance is more affected by the overhead than by the effective hit rate. This argues for cache designs that can propagate cache information independently of object discovery and delivery.

6.5 Trends affecting Web caching

6.5.1 Web advertisement

In Chapter 2 we explained how Web advertisers are misusing HTTP/1.1 cache control directives to prevent caching of static images. These mechanisms are harmful because they generate many repeat requests that add to network congestion and are often needlessly forwarded through the cache levels. Further, these repeat requests skew the popularity distributions, leading to misinterpretation of user behavior. Web advertisers

use these mechanisms because their revenues depend upon the number of advertisement hits. As Web advertising is expected to increase, this problem is of growing concern. Several solutions could be explored. For example, Web browsers could monitor for repeat requests and either send such requests directly to the origin server, or cache the document in spite of the advertiser's HTTP cache control directives. A second approach is to modify the revenue model to allow Web caches to count hits for advertisers, or to statistically estimate hits based upon the true Zipf distribution and number of cache clients.

6.5.2 Multimedia

A second growing trend is the increase in audio and video traffic on the Internet. Real time transfers of multimedia use UDP, which does not contain congestion control mechanisms. As multimedia increases, the UDP traffic will increasingly contend with TCP/IP traffic. During heavy network loads, TCP traffic backs off and frees resources. If routers do not constrain UDP traffic, then UDP applications will grab the newly freed resources. This scenario makes the congestion avoidance aspect of cooperative Web caching even more attractive, as it may provide a means to route around heavy UDP traffic.

6.6 Directions for Web caching research

6.6.1 Standardization of terms and metrics

One of the highest priority items in the Web caching community should be the standardization of terms and metrics. Differing definitions make it very difficult to compare results across studies. Standardization would reduce the spread in reported hit rates and would allow for analysis of factors that differ between studies, such as number of proxies or number of users.

6.6.2 Trace sharing

A second priority should be the collection and sharing of high quality traces. The Internet Traffic Archive [51] does maintain and distribute some traces, but more are needed. Many traces reported in the literature could aid other research if they were made available. Further, there should be a collective, well-supported effort to

collect simultaneous traces at several (perhaps 10 or more) large proxy caches. Individual researchers have insufficient access to collect such traces, and this data would be of great value to the research community as a whole.

6.6.3 Integrate flat cache organization and regional networks

We showed cache speedup improves substantially when more caches share a regional or high-speed network. Research should focus on integrating a preference for regional caches into a flat cache mesh design. For example, metadata could be propagated more often to regional neighbors, and less often to far-off caches.

6.6.4 Focus on reducing overhead, not increasing hit rate

Because overhead has more effect on cache success than does hit rate, methods that reduce overhead will be more fruitful than methods that increase hit rate. This suggests cache designs should favor lower overhead over improved hit rate, with a balance point suggested by the isospeedup plots in Chapter 5.

6.6.5 Make Web browsers smarter

As mentioned above, Web browsers and local LAN proxies could reduce Internet traffic by monitoring for repeat requests and taking steps to eliminate them. Currently content providers are responsible for most cache control directives, and some are abusing the intent of the HTTP/1.1 protocol design. In such cases, Web browsers should give users the authority to decide whether they wish to repeatedly fetch the same gray GIF images or store them in the cache.

6.7 Final note

The Web caching community needs to realize a major benefit of cooperative Web caching stems from its ability to improve overall Internet performance by identifying and avoiding congested routes. Response times will decrease as more clients use cooperative caches because there will be more available queue space at routers and fewer packet drops, not because additional users increase sharing opportunities and cache hit rates. Research should focus on evaluating this aspect of cooperation rather than solely examining hit rate

metrics. An improvement dispersed across traffic flows is difficult to measure, especially when it interacts with the number of flows that are using the improvement and the number that are not. This is a complex and potentially fruitful area of Internet research because it offers an opportunity to introduce congestion avoidance into the application layer instead of relying entirely upon the network layer.

Appendix A

Server-directed proxy sharing (SDP)

A.1 SDP components and features

SDP implements Internet-wide cache sharing through cooperating proxy server caches. Proxies find cached copies by looking in local metadata directories and propagate update metadata by piggybacking it onto data transfers. In concert with directory-based discovery, SDP uses client-initiated dissemination and direct delivery. Figure A.1 shows SDP's components and Table A.1 highlights its major features.

SDP employs four information structures: *Proxy Tables*, *Proxy Lists*, *Cache Site Directories* and *Popular Lists*. Each server maintains a Proxy Table containing information about copies of its objects, including cache IP addresses. From the Proxy Table, the server compiles Proxy Lists of individual object metadata. Each proxy server maintains a Popular List and a Cache Site Directory. A proxy's Popular List points to the most popular objects in its local cache, while its Cache Site Directory contains metadata for objects at other cache sites. Metadata is used for both discovery and site selection. Criteria for selecting sites can

Features of Server-Directed Proxy Sharing
Cooperating proxy caches share Web objects.
Proxy caches are organized into a flat mesh.
Proxies locate cached copies by searching local metadata directories.
Comprehensive metadata is propagated nightly.
Update metadata is propagated by lazy prefetching.
Cache discovery is orthogonal to cache selection.
Proxy servers cache only objects requested by local clients.

Table A.1: Features of the SDP cache design.

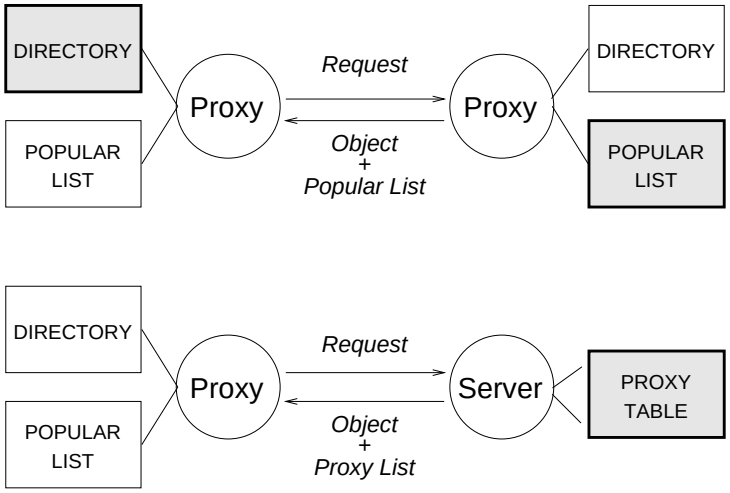

Figure A.1: *Proxy to proxy request (top), and proxy to server request (bottom).*

change, so metadata records have variable length and use tags to indicate information type. For example, metadata can include copy timestamp, security data and site performance information such as average load, network bandwidth, or latency history. Metadata is propagated by nighttime exchange of cache digests [26] supplemented by daytime update, as detailed in Section A.3.

A.2 SDP protocol

Users send requests to their local proxy server. If the proxy has the object cached, it returns a copy; otherwise it retrieves the object as explained below and in Figure A.2.

1. Client proxy looks up object in its local Cache Site Directory.

If the local Cache Site Directory lists cache site(s) for the object, the client proxy selects among those sites. If directory lookup fails, the proxy contacts the original server. Directory information need not be comprehensive; its function is to provide fast hints about copy locations with a small miss penalty.

2. Client proxy selects a site.

One benefit of SDP is that it separates discovery from site selection, allowing sites to be chosen based upon various criteria. If the criteria is response time, SDP uses dynamic network probes (see Chapter 3) and includes the server in the candidate list because it may offer the fastest response time.

```
proc SDP_GET (object)
    if (object in local data cache) then return object
    else if (object is listed in Directory) then
        select best cache site
        request object from selected site
        if (ok) then
            store Popular List in Directory
            return object
        end if
    end if

    /* Object is not in Directory or request failed */
            request object from primary server
    if (Proxy List was attached to reply) then
        store Proxy List in Directory
    if (Proxy List and no object) then SDP_GET(object)
    return object
end
```

Figure A.2: *SDP client proxy*

3. Server returns object and/or Proxy List or cache returns object and Popular List.

The server may return the object, a Proxy List for the object, or the object plus a Proxy List for related objects. A cache site may returns the object, an error message, or the object plus metadata updates from its Popular List. If the cache site returns an error message, the client proxy obtains the object by sending the Web server an ordinary HTTP request.

A.3 Metadata propagation

The hourly response time plots in Section 3.6.5 show Web response time is an order of magnitude faster at night than during the day. To take advantage of this diurnal cycle, the metadata propagation in SDP is based on an exchange of comprehensive metadata between proxy caches during the quiescent nighttime hours. The question then becomes whether metadata remains valid during the next day. The answer is found in results from the cache warmup experiment (Section 2.5.1). For the NLANR BO1 cache, the measured hit rate is 30%, of which only 8% is accounted for by documents requested that same day. The fraction of hits for documents cached in prior days but not the same day is 22/30, or 0.73. For the NLANR UC

cache, this fraction is 0.67. Thus the range 0.67 – 0.73 is an estimate of the lower bound on the fraction of nightly propagated metadata that would remain valid during the next 24 hours. The estimate is a lower bound because documents requested during previous days may also have been requested on the current day. As the most popular documents are likely requested each day, the fraction of valid metadata is almost certainly greater than 0.67 – 0.73. This fraction represents the efficiency of metadata propagation as defined in Section 5.3.1.

Cache digests proposed by Fan et al. [26] provide one method for compressing and communicating the entire contents of a cache. Our results indicate that propagating cache digests at night would provide a cache efficiency of approximately 70% without adding overhead to the cache response time. We propose supplementing nighttime exchanges of cache digests with daytime updates propagated via lazy piggybacking that operates as follows:

- when a server returns an object, it appends a Proxy List for related objects such as embedded images,
- when a proxy returns an object copy, it appends an update list containing metadata for popular documents that were cached after the latest nighttime exchange of cache metadata.

Only metadata for documents not in the nighttime cache digests need be piggybacked. Moreover, the daily updates can be limited to documents that exhibit a sudden increase in popularity to insure metadata for suddenly hot documents spreads rapidly while maintaining a low propagation overhead.

A.4 Cache consistency issues

Standard proxy server caching does not guarantee cache consistency: a proxy cache may return stale objects to its local users. Object timestamps, expiration headers, and proxy server directives are used to limit stale returns. SDP can provide the same safeguards by including the timestamps and expiration dates in the metadata, thereby ensuring cache consistency is no worse for shared objects than for objects returned from the local proxy cache.

```
proc SDP_GET_EMBEDDED (Web page)
    select set of best cache sites
    for each (embedded object)
        send request to next site
        mark site as "Unavailable"
    end

    /* Receive objects as they come in and return them */
    while (more objects)
        if (object received) then
            mark page site "Available"
            store Popular List in Directory
            return object
        else if (proxy timeout) then
            request object from next available site
        else if (proxy error) then
            request object from primary server
        else if (server timeout or error) then
            return error
    end
end
```

Figure A.3: *Concurrent image retrieval.*

A.5 Concurrent retrieval of embedded objects

If the requested object is a Web page, the client proxy can retrieve embedded images concurrently from different cache sites (see Figure A.3). The client proxy selects the best site, and sends it the first image request. Without waiting for a response, the client selects the next best site and continues until all images have been requested or all sites have outstanding requests. To guard against excessive delay, the client sets reply timers and sends the request to another site if a timer expires. The request to the slower sites is cancelled after the image is received. This technique can also be used for other strongly related objects or for portions of large objects. Rodriguez, et al., found substantial performance advantage for a similar approach that uses parallel access to mirror sites [54].

A.6 Summary

The SDP protocol is designed to match known characteristics of the Web, the most important being limited request duplications within user groups, skewed popularity distributions, rapid popularity shifts, fluctuating network congestion, and the structure of Web pages. By organizing cache sites as a flat mesh and propagating metadata updates with nighttime exchanges of comprehensive metadata supplemented by daytime update via lazy prefetching, SDP combines low delivery cost with low discovery cost. A second important feature of SDP is the separation of discovery from cache site selection, allowing site selection to be based upon multiple and configurable criteria. Lastly, we believe the Internet presents special problems for network caching because it has no central controlling authority. Our design takes into account this autonomous administrative nature. In push-caching, the local caches store unsolicited objects that may not be needed by local users, raising security and resource-sharing issues. Hierarchical cache designs require dedicated caches supported by government funding or a "pay-for-services" plan; neither is a particularly attractive option. In contrast, SDP requires only that local proxy servers share information they previously cached for their local users. Thus SDP matches both technical and political characteristics of the Web.

Appendix B

Hit rate expression for hierarchical caches

This Appendix develops an expression to compute total hit rate for the L3 level using data from hierarchical caches. The trace for a hierarchical cache contains information about received requests, but does not contain information about data exchanged between the cache's children. Sibling information must be determined by examining the children's traces. Let:

H_3	=	total hit rate for the L3 level
$H_{3,Parent}$	=	hit rate at the L3 parent cache
P_3	=	fraction of L2 requests sent to the parent cache
S_3	=	fraction of L2 requests sent to sibling caches.

In Figure B.1, the left diagram illustrates the proportioning of the L2 request flow between the parent cache and the sibling caches, while the right diagram illustrates the fraction of each flow that are L3 hits: $P_3 H_{3,Parent}$ is the fraction of requests that are parent cache hits and S_3 is the fraction of requests that are sibling cache hits.[1] The total hit rate for the L3 level is

$$H_3 = \frac{P_3 H_{3,Parent} + S_3}{P_3 + S_3} = \frac{H_{3,Parent} + S_3/P_3}{1 + S_3/P_3}. \quad \text{(B.1)}$$

The parent hit rate, $H_{3,Parent}$, is determined from the parent cache trace, while the ratio of sibling requests to parent requests, S_3/P_3, is determined from the children's traces.

[1] Based on the ICP protocol, we assume all sibling requests are sibling hits.

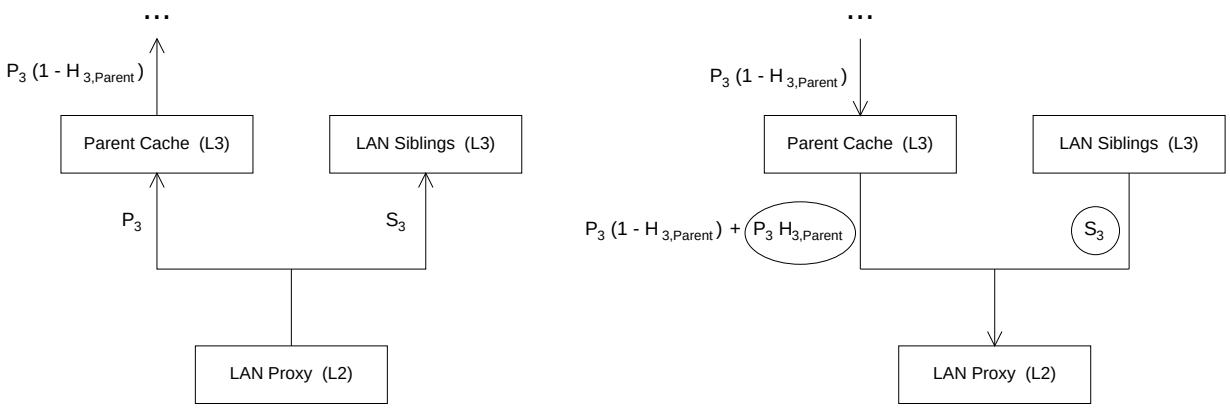

Figure B.1: *L3 request fractions (left) and L3 reply fractions (right). Circled terms are L3 level hits.*

Appendix C

Source code listings

C.1 Availability and restrictions

Source code and documentation for the utilities `httpget` and `tcping` are publicly available at the Web site `www.cs.utsa.edu/~sdykes/`. The code may be freely used and redistributed, with the restriction that code changes be documented in the header before redistribution of a modified version.

C.2 Makefile

```
#---------------------------------------------------------------------
#       Makefile
#---------------------------------------------------------------------
CC      = gcc
CFLAGS  = -O
INC     =
OBJS    = wrapper.o

#-----------------------------------------------------
#       LIBRARIES FOR SOLARIS AND LINUX:
#       use  LIBS = $(L_SOLARIS) or  LIBS = $(L_LINUX)
#-----------------------------------------------------
L_SOLARIS = -lsocket -lnsl
L_LINUX   =
LIBS      = $(L_SOLARIS)
#LIBS     = $(L_LINUX)

all:
                make httpget
                make tcping
                make clean

httpget:        httpget.o $(OBJS)
                $(CC) $(CFLAGS) -o httpget httpget.o $(OBJS) $(LIBS)

tcping:         tcping.o $(OBJS)
                $(CC) $(CFLAGS) -o tcping tcping.o $(OBJS) $(LIBS)

clean:
                rm *.o
```

C.3 tools.h

```
/*------------------------------
*       tools.h
*
*       written by S.Dykes
*       last update:   03-10-2000
*------------------------------*/

#ifndef _TOOLS_H
#define _TOOLS_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <limits.h>
#include <fcntl.h>
#include <math.h>
#include <errno.h>

#include <sys/types.h>
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/utsname.h>

#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define min(a, b)       ((a) > (b) ? (b) : (a))
#define max(a, b)       ((a) < (b) ? (b) : (a))

#define MAXLINE         256
#define MAX_DATELEN     26      /* length of date string        */
#define IP4_ADDRESSLEN  16      /* length of IPv4 address string*/
#define IP4_HEADERLEN   20      /* length of std IPv4 header    */
#define TCP_HEADERLEN   20      /* length of std TCP header     */
#define ICMP_HEADERLEN  8       /* length of ICMP header        */
#define UNUSED_PORT     62957
#define HTTP_OK         200

enum            {VERBOSE=1, LOG, PARTIAL};

#define DIFF_TIME(x,y)  \
        (x.tv_sec + x.tv_usec*1.E-06 - y.tv_sec - y.tv_usec*1.E-06)

void    my_herror       (const char*);
void    errx            (const char*);
void    errx2           (const char*, const char*);
int     Read            (int, char*, int);
FILE*   Fopen           (char*, char*);
int     Connect         (int, struct sockaddr*, int);
int     NB_Connect      (int, struct sockaddr*, int);
int     Fcntl           (int, int, int);
int     Select          (int, fd_set*, fd_set*, fd_set*, struct timeval*);
int     RST_Select      (int, fd_set*, fd_set*, fd_set*, struct timeval*);
int     Socket          (int, int, int);
```

```
int     Gethostname     (char*, int);
struct  hostent *Gethostbyname  (const char*);

#endif  /* _TOOLS_H */
```

C.4 httpget.c

```
/*--------------------------------------------------------------------------
*
*       httpget  [-r] [-l|-p|-v] [-t timeout] [-o outfile] [-h headerfile] URL
*
*       Measure time for object download from an HTML server. Uses HTTP 1.1
*       protocol.
*
*       Outputs:
*               Date
*               Client name
*               Server canonical name
*               Server IP address
*               URL
*               Bytes returned (includes HTTP header bytes)
*               DNS lookup time
*               Connect time
*               GET latency   = after connecting, time to first select
*               GET remainder = after 1st select, time to complete the GET
*               GET total     = latency + remainder
*               GET bandwidth = bytes / (GET total)
*
*       Arguments:
*               URL = object to be downloaded.
*
*       Options:
*               -h      File containing additional HTTP1.1 requests headers,
*                       one header per line.  If the terminating \r character is
*                       not included in the input file, it is added.
*
*               -l      Print results in log format. The log format is ascii,
*                       but less readable than standard format.
*
*               -p      Print only date and times in log format. This is useful
*                       for repeated measures using the same client, server,
*                       and url because it generates a much smaller log file.
*
*               -v      Output as-you-go info to stderr,
*                       in addition to standard format to stdout.
*
*               -r      Send request header "Pragma: no-cache".
*
*               -t timeout      Timeout period for GET latency. Timeout is a
*                       decimal value, with units of seconds.  Default is TCP/IP
*                       timeout. (This option does not change the connect
*                       timeout which on most systems is 75 seconds.)
*
*               -o outfile      Save object to outfile.  ! This affects timing
*                       data because timing will include file write time. !
*
*       Compile:
*               gcc -o httpget httpget.c wrapper.c -lsocket -lnsl
*
*
*       Written by Sandra G. Dykes
*       Last update:  3-10-2000
*--------------------------------------------------------------------------*/
#include        "tools.h"
```

```
#define BUFLEN          10*1024         /* each read can handle up to 10KB   */
#define MAX_PROTOCOLLEN 50
#define MAX_REQUESTLEN  MAX_PROTOCOLLEN + MAXHOSTNAMELEN + MAXPATHLEN + 100

void    parse_url       (char*, char*, char*, int*, char*);

/*****************************************************
*
*       main()
*
*****************************************************/
int main (int argc, char *argv[])
{
        char    usage[]=
"Usage: httpget [-r] [-l|-p|-v] [-t timeout] [-o outfile] [-h headerfile] URL";

        char    line[]="-------------------------------------------------";
        char    ch, *p, *q;
        int     n, reload=0, form=0, port, ss, ns, n_reads, status;
        long    bytes, first_bytes, rest_bytes, total_bytes;
        double  dns_sec, connect_sec, first_sec, rest_sec, get_sec,
                total_sec, sec, bw, norm_10K;
        char    *outfile, *url,
                request [MAX_REQUESTLEN],
                protocol[MAX_PROTOCOLLEN],
                path[MAXPATHLEN],
                date[MAX_DATELEN],
                server_IP  [IP4_ADDRESSLEN],
                server_name[MAXHOSTNAMELEN],
                client_name[MAXHOSTNAMELEN],
                buf[BUFLEN],
                response_hdr[BUFLEN];

        struct  hostent         *host_info;
        struct  in_addr         in;
        struct  sockaddr_in     addr;
        struct  timeval         tout, *timeout, time0, time1, time2;
        time_t                  timep;
        fd_set                  Rset;
        FILE                    *fp, *fp_hdr;

        /*------------------------------*/
        /*      COMMAND LINE OPTIONS    */
        /*------------------------------*/
        opterr  = 0;
        fp      = NULL;
        timeout = NULL;
        while ( (ch = getopt(argc, argv, "h:lvpro:t:")) != EOF)
        {
                switch(ch)
                {
                case 'l':  if (form) errx(usage);  form= LOG;      break;
                case 'p':  if (form) errx(usage);  form= PARTIAL;  break;
                case 'v':  if (form) errx(usage);  form= VERBOSE;  break;
                case 'r':  reload=1;                               break;
                case 'h':  fp_hdr = Fopen(optarg, "r");            break;
                case 'o':  fp     = Fopen(optarg, "w");            break;
```

```
        case 't':  sec = atof(optarg);
                   tout.tv_sec  = (long) sec;
                   tout.tv_usec = (long) (1.E+06 * (sec-tout.tv_sec));
                   timeout = &tout;
                   break;

        default:   errx (usage);
        }
}
argc -= optind;
argv += optind;
if (argc<1)  errx (usage);

/*------------------------------*/
/*      PARSE URL               */
/*------------------------------*/
url = *argv;
parse_url (url, protocol, server_name, &port, path);
if (form==VERBOSE)
{
        printf ("\nURL:     \t\t%s\n", url);
        printf ("Protocol:\t\t%s\n", protocol);
        printf ("Host:    \t\t%s\n", server_name);
        printf ("Port:    \t\t%d\n", port);
        printf ("Object:  \t\t%s\n", path);
        printf ("Options: \t\t");
        if (reload)             printf ("reload=YES  ");
        else                    printf ("reload=NO  ");
        if (fp)                 printf ("save=YES  ");
        else                    printf ("save=NO  ");
        if (fp_hdr)             printf ("headers=YES  ");
        else                    printf ("headers=NO  ");
        if      (form==VERBOSE) printf ("output=VERBOSE  ");
        else if (form==LOG)     printf ("output=LOG  ");
        else if (form==PARTIAL) printf ("output=PARTIAL  ");
        else                    printf ("output=STANDARD  ");
        if (timeout)            printf ("timeout= %g sec\n",sec);
        else                    printf ("timeout=TCP DEFAULT\n");
}

/*------------------------------*/
/*      PREPARE REQUEST         */
/*------------------------------*/
sprintf (request, "GET %s HTTP/1.1\r\nHost: %s\r\n", path, server_name);
strcat  (request, "User-Agent: Mozilla/4.0 Httpget\r\n");
if (reload)  strcat (request, "Pragma: no-cache\r\n");
strcat  (request, "Accept: */*\r\n");
strcat  (request, "Connection: close\r\n");
if (fp_hdr)
{
        /*------------------------------*/
        /*      Add any extra headers   */
        /*------------------------------*/
        while (fgets (buf, BUFLEN, fp_hdr))
        {
                n = strlen(buf);
                if (buf[n-1] == '\n') buf[n-1] = '\0';
                if (buf[n-2] == '\r') buf[n-2] = '\0';
                strcat (request, buf);
```

```
                strcat (request, "\r\n");
        }
}
strcat  (request, "\r\n");
printf  ("%s\nREQUEST\n\n%s%s\n", line, request, line);

/*------------------------------*/
/*      DNS LOOKUP FOR SERVER   */
/*------------------------------*/
gettimeofday (&time0, NULL);
host_info = Gethostbyname (server_name);
if (!host_info) exit(0);
gettimeofday (&time1, NULL);
memcpy  (&in, host_info->h_addr_list[0], sizeof(in));
strcpy  (server_name, host_info->h_name);
strcpy  (server_IP,   inet_ntoa(in));
dns_sec = DIFF_TIME(time1, time0);

/*------------------------------*/
/*      WRITE OPTIONS           */
/*------------------------------*/
Gethostname (client_name, MAXHOSTNAMELEN);
timep = time0.tv_sec;
strcpy  (date, ctime(&timep));
date[strlen(date)-1]='\0';
if (form!=LOG && form!=PARTIAL)
{
        printf ("\nDate:   \t\t%s\n",           date);
        printf ("Client: \t\t%s\n",             client_name);
        printf ("Server canonical name:\t%s\n", server_name);
        printf ("Server IP Address:    \t%s\n", server_IP);
        printf ("Document:     \t\t%s\n\n",     path);
        printf ("DNS lookup:   \t\t%.3f sec\n", dns_sec);
        printf ("Connect:      \t\t");
}
/*------------------------------*/
/*      SERVER SOCKET ADDRESS   */
/*------------------------------*/
bzero (&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port   = htons(port);
memcpy (&addr.sin_addr, host_info->h_addr_list[0],
        sizeof(struct in_addr));

/*------------------------------*/
/*      OPEN SOCKET AND CONNECT */
/*------------------------------*/
ss = Socket (AF_INET, SOCK_STREAM, 0);
gettimeofday (&time0, NULL);
Connect (ss, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
gettimeofday (&time1, NULL);
connect_sec = DIFF_TIME(time1, time0);
if (form!=LOG && form!=PARTIAL) printf("%.3f sec\n", connect_sec);

/*-------------------------------------*/
/*      SEND REQUEST, WAIT FOR REPLY.  */
/*-------------------------------------*/
FD_ZERO (&Rset);
FD_SET  (ss, &Rset);
```

```
gettimeofday (&time0, NULL);
write        (ss, request, strlen(request));
ns = Select  (ss+1, &Rset, NULL, NULL, timeout);
gettimeofday (&time1, NULL);

/*------------------------------*/
/*      READ REPLY              */
/*------------------------------*/
if (!ns)
{
        if      (form==PARTIAL) printf ("%s -1\n", date);
        else if (form==LOG)     printf ("HTTPGET %s %s %s %s -1\n",
                                date, client_name, server_IP, url);
        else printf ("Timeout waiting for object (%g sec)\n\n",sec);
        close(ss);
        if (fp) fclose(fp);
        exit(-1);
}
total_bytes = n_reads = 0;
while (1)
{
        if (!n_reads)
        {
                /*-----------------------------------------------*/
                /*      First packet - contains response header */
                /*                                               */
                /*      Latency = time to receive first packet  */
                /*-----------------------------------------------*/
                bytes = Read (ss, response_hdr, BUFLEN);
                first_bytes = bytes;
                if (!bytes) break;
                if (fp)  fwrite (response_hdr, 1, bytes, fp);
        }
        else
        {
                /*------------------------------*/
                /*      Remaining packets       */
                /*------------------------------*/
                bytes = Read (ss, buf, BUFLEN);
                if (!bytes) break;
                if (fp)  fwrite (buf, 1, bytes, fp);
        }
        total_bytes += bytes;
        n_reads++;
}
gettimeofday (&time2, NULL);
first_sec  = DIFF_TIME(time1, time0);
rest_sec   = DIFF_TIME(time2, time1);
get_sec    = first_sec + rest_sec;
total_sec  = dns_sec + connect_sec + get_sec;
rest_bytes = total_bytes - first_bytes;

/*------------------------------*/
/*      CLOSE SOCKET & OUTFILE  */
/*------------------------------*/
close(ss);
if (fp) fclose(fp);
/*------------------------------*/
/*      WRITE TIMING RESULTS    */
```

```
        /*-------------------------------*/
        if (get_sec)    bw= 0.001*total_bytes/get_sec;
        else            bw= 0;
        if (rest_bytes) norm_10K = connect_sec + first_sec +
                                   (rest_sec * 10000/rest_bytes);
        else            norm_10K = 0;
/*
        fprintf (stderr, "connect_sec= %.3f sec\n",   connect_sec);
        fprintf (stderr, "first_sec  = %.3f sec\n",   first_sec);
        fprintf (stderr, "rest_sec   = %.3f sec\n",   rest_sec);
        fprintf (stderr, "total_bytes= %ld  bytes\n", total_bytes);
        fprintf (stderr, "total_time = %.3f sec\n",   connect_sec +
                                                      first_sec + rest_sec);
        fprintf (stderr, "\n");
        fprintf (stderr, "rest_bytes = %ld  bytes\n", rest_bytes);
        fprintf (stderr, "norm_10K   = %.3f sec\n",   norm_10K);
        fprintf (stderr, "\n");
*/
        if (form==PARTIAL)
        {
                printf ("%s %ld\t%.3f %.3f %.3f %.3f %ld\n",
                        date, total_bytes, dns_sec, connect_sec,
                        first_sec, rest_sec, first_bytes);
        }
        else if (form==LOG)
        {
        printf ("HTTPGET %s %s %s %s %ld\t%.3f %.3f %.3f %.3f %ld %.3f   %g\n",
                   date, client_name, server_IP, url, total_bytes,
                   dns_sec, connect_sec, first_sec, rest_sec,
                   first_bytes, bw, norm_10K);
        }
        else
        {
           printf ("GET latency:  \t\t%.3f sec  for  %ld bytes",
                   first_sec, first_bytes);
           if (first_sec) printf (", \tBW= %.3f KB/s",
                                  .001*first_bytes/first_sec);
           printf ("\n");

           rest_bytes = total_bytes - first_bytes;
           printf ("GET remainder:\t\t%.3f sec  for  %ld bytes",
                   rest_sec, rest_bytes);
           if (rest_sec) printf (", \tBW= %.3f KB/s",
                                 .001*rest_bytes/rest_sec);
           printf ("\n");

           printf ("GET total:    \t\t%.3f sec  for  %ld bytes",
                   get_sec, total_bytes);
           if (form==VERBOSE) printf (" (%d read calls)", n_reads);
           if (get_sec)       printf (", \tBW= %.3f KB/s",
                                      .001*total_bytes/get_sec);
           printf ("\n");

           printf ("\n\nTotal time:   \t\t%.3f sec\n\n", total_sec);
        }
        /*-------------------------------*/
        /*      PRINT RESPONSE HEADER    */
        /*-------------------------------*/
        for (n=0, p=response_hdr+4;  n<first_bytes;  p++)
```

```
                if (*(p-3)=='\r' &&
                    *(p-2)=='\n' &&
                    *(p-1)=='\r' &&
                    *p=='\n')    break;
        *(p-1) = '\0';
        printf ("\n%s\nRESPONSE HEADER\n\n%s\n%s\n", line, response_hdr, line);
}

/*************************************************************
*
*       parse_url()
*
*-------------------------------------------------------------*/
void parse_url (char *url, char *protocol, char *host, int *port, char *path)
{
        int     n;
        char    *t, *slash, *colon;
        char    delim[] = "://";

        /*----------------------*/
        /*      Protocol        */
        /*----------------------*/
        t = url;
        if (!(colon = strstr(t, delim)))  strcpy (protocol,"http");
        else
        {
                n = (int)(colon-t);
                strncpy (protocol, t, n);
                protocol[n+1] ='\0';
                t = colon + strlen(delim);
        }
        /*----------------------*/
        /*      Host            */
        /*----------------------*/
        if (!(slash = strchr(t, '/')))
        {
                strcpy(path, "/");
                strcpy(host, t);
        }
        else
        {
                strcpy (path, slash);
                *slash = '\0';
                strcpy (host, t);
                *slash = '/';
        }
        /*----------------------*/
        /*      Port            */
        /*----------------------*/
        if (!(colon= strchr (host,':'))) *port = 80;
        else                             *port = atoi(colon + 1);
}
```

C.5 tcping.c

```
/*---------------------------------------------------------------------
*
*       tcping [-l|-p|-v] [-c count] [-i interval] [-t timeout] [-P port] host
*
*       Estimates network latency by measuring the round trip time for
*       TCP header packets.  The client sends a TCP SYN packet to an
*       arbitrary (hopefully unused) port on the server.  If the port is
*       unused, the server responds with a TCP RST packet.  The tcping
*       application detects the RST reply because it generates a
*       EONNREFUSED socket error.
*
*       tcping is similar to the standard ping program in that it measures
*       round trip time (RTT) to remote network locations.  However, ping
*       uses ICMP echo messages so must run as root or setuid, while
*       tcping runs with user permissions.  Consequently, tcping can be
*       adapted into a user-level library function and called without the
*       overhead of using system().  It can also be adapted to send concurrent
*       SYN probes to multiple hosts without using threads or forks.
*       Unfortunately, the SYN/RST technique does not allow packet size to
*       be varied; packets are always TCP header packets without data.
*
*       Because no TCP connection is establish, the SYN/RST probes
*       avoid TCP window effects and do not result in TCP TIME_WAIT states.
*
*       Arguments:
*               host = server name or IP address
*
*       Options:
*               -l      Print results in a log form. The log form is ascii,
*                       but less readable than standard format.
*
*               -p      Print only date and times in a log form. This is useful
*                       for repeated measures using the same client, server,
*                       and url because it genertes a much smaller log file.
*
*               -v      Output as-you-go info to stderr,
*                       in addition to standard form to stdout.
*
*               -c count        Number of packets to send.
*
*               -i interval     Time in seconds between packet sends.
*
*               -t timeout      Timeout period (sec) to wait for RST reply.
*
*               -P port         Destination port, hopefully unused.
*
*       Output:
*               Date
*               Client name
*               Server canonical name
*               Server IP address
*               { RTT } = round trip times,
*                       where RTT = time(RST receive) - time(SYN sent)
*
*               Summary statistics:
*                       nbr packets sent, received, and loss percent
*                       min/avg/max RTT
```

```
*
*       Compile:
*               gcc -o tcping tcping.c wrapper.c -lsocket -lnsl
*
*
*       Written by Sandra G. Dykes  (sdykes@cs.utsa.edu)
*       Last update:    3-10-2000.
*---------------------------------------------------------------------*/
#include               "tools.h"

/*************************************************
*
*       main()
*
*************************************************/
int main (int argc, char *argv[])
{
        char    usage[]=
"Usage: tcping [-l|-p|-v] [-c count] [-i interval] [-t timeout] [-P port] host";

        int     i, port, count, interval, timeout, form, bytes,
                ss, nr, ns, flags;
        double  dns_sec, delay_sec, RTT, rsum, rmin, rmax, loss;
        char    ch,
                client_name[MAXHOSTNAMELEN],
                server_name[MAXHOSTNAMELEN],
                server_IP[IP4_ADDRESSLEN],
                date[MAX_DATELEN];
        fd_set  Rset;
        struct  hostent         *host_info;
        struct  in_addr         in;
        struct  sockaddr_in     addr;
        struct  timeval         now, delay, tout, Tsend, Tr;

        /*------------------------------*/
        /*      COMMAND LINE OPTIONS    */
        /*------------------------------*/
        form     = 0;
        bytes    = IP4_HEADERLEN + TCP_HEADERLEN;
        port     = UNUSED_PORT;
        count    = 10;
        interval = 0;
        timeout  = 0;

        opterr = 0;
        while ((ch = getopt(argc, argv, "lvpc:i:t:P:")) != EOF)
        {
                switch(ch)
                {
                case 'l':  if (form) errx(usage);  form= LOG;      break;
                case 'p':  if (form) errx(usage);  form= PARTIAL;  break;
                case 'v':  if (form) errx(usage);  form= VERBOSE;  break;

                case 'c':  count = atoi(optarg);
                           if (count<1)
                                   errx ("Bad argument: count must be >0");
                           break;

                case 'i':  interval = atoi(optarg);
```

```
                    if (interval<1)
                            errx ("Bad argument: interval must be >0");
                    break;

            case 't':  timeout = atoi(optarg);
                    if (timeout<1)
                            errx ("Bad argument: timeout must be >0");
                    break;

            case 'P':  port = atoi(optarg);
                    if (port<1)
                            errx ("Bad argument: port must be >0");
                    break;

            default:   errx (usage);
            }
    }
    if (!interval && !timeout)  interval = timeout = 1;
    else if (!interval) interval= timeout;
    else if (!timeout)  timeout = interval;
    else if (interval < timeout)
    {
       fprintf (stderr,
       "Bad argument: interval (%d sec) must be > timeout (%d sec)\n\n",
                    interval, timeout);
       exit(-1);
    }
    argc -= optind;
    argv += optind;
    if (argc<1)  errx (usage);

    /*------------------------------*/
    /*      DNS LOOKUP FOR SERVER   */
    /*------------------------------*/
    gettimeofday (&Tsend, NULL);
    host_info = Gethostbyname (*argv);
    gettimeofday (&Tr, NULL);
    memcpy  (&in, host_info->h_addr_list[0], sizeof(in));
    strcpy  (server_name, host_info->h_name);
    strcpy  (server_IP,   inet_ntoa(in));
    dns_sec = DIFF_TIME(Tr, Tsend);

    /*------------------------------*/
    /*      SERVER SOCKET ADDRESS   */
    /*------------------------------*/
    bzero (&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port   = htons(port);
    memcpy (&addr.sin_addr, host_info->h_addr_list[0],
            sizeof(struct in_addr));

    /*------------------------------*/
    /*      PRINT OPTIONS           */
    /*------------------------------*/
    Gethostname  (client_name, MAXHOSTNAMELEN);
    gettimeofday (&now, NULL);
    strcpy       (date, ctime((time_t *) &(now.tv_sec)));
    date [strlen(date)-1]='\0';
    if (form!=LOG && form!=PARTIAL)
```

```
{
        printf ("\nDate:      \t\t%s\n",        date);
        printf ("Client:    \t\t%s\n",         client_name);
        printf ("Server canonical name:\t%s\n", server_name);
        printf ("Server IP Address:    \t%s\n", server_IP);
        printf ("Port:      \t\t%d\n",         port);
        printf ("\nIP packet size:       \t%d bytes\n",  bytes);
        printf ("Count:     \t\t%d\n",         count);
        printf ("Interval:  \t\t%d sec\n",     interval);
        printf ("Timeout:   \t\t%d sec\n",     timeout);
        printf ("DNS lookup:\t\t%.3f sec\n\n",  dns_sec);
}
/*---------------------------------------------*/
/*      COUNT LOOP:  SEND SYN/ RECEIVE RST     */
/*---------------------------------------------*/
rmin =  DBL_MAX;
rmax = -DBL_MAX;
rsum = nr = 0;
for (i=0; i<count; i++)
{
        /*-------------------------------------*/
        /*      OPEN NON-BLOCKING SOCKET       */
        /*-------------------------------------*/
        ss    = Socket (AF_INET, SOCK_STREAM, 0);
        flags = Fcntl  (ss, F_GETFL, 0);
        Fcntl (ss, F_SETFL, flags|O_NONBLOCK);

        /*-------------------------------------*/
        /*      RESET TIMEOUT & FD_SET         */
        /*-------------------------------------*/
        tout.tv_sec  = timeout;
        tout.tv_usec = 0;
        FD_ZERO(&Rset);
        FD_SET(ss, &Rset);

        /*-------------------------------------*/
        /*      DELAY UNTIL NEXT INTERVAL      */
        /*-------------------------------------*/
        gettimeofday(&now, NULL);
        if (!i) delay_sec= 0;
        else    delay_sec= interval -
                        ((now.tv_sec+1.E-06*now.tv_usec) -
                        (Tsend.tv_sec+1.E-06*Tsend.tv_usec));
        if (delay_sec >0)
        {
                if (form==VERBOSE) fprintf (stderr,
                        "delaying %g sec ...\t",delay_sec);
                delay.tv_sec  = (int)(delay_sec);
                delay.tv_usec = (int)(1.E+06 *
                                (delay_sec - delay.tv_sec));
                Select (ss+1, NULL, NULL, NULL, &delay);
        }
        /*-------------------------------------*/
        /*      SEND SYN, WAIT FOR RST         */
        /*-------------------------------------*/
        gettimeofday (&Tsend, NULL);
        NB_Connect   (ss, (struct sockaddr *)&addr,
                      sizeof(struct sockaddr_in));
        ns = RST_Select (ss+1, &Rset, NULL, NULL, &tout);
```

```
                gettimeofday (&Tr, NULL);
                close (ss);
                /*--------------------------------------*/
                /*      WRITE RTT                       */
                /*--------------------------------------*/
                RTT   = DIFF_TIME(Tr, Tsend);
                if (!ns)
                {
                        if (form!=PARTIAL && form!=LOG)
                                fprintf(stderr,
                                "RT time =  ***   no reply after %g sec\n",RTT);
                }
                else
                {
                        nr++;
                        RTT   = DIFF_TIME(Tr, Tsend);
                        rsum += RTT;
                        rmin  = min(RTT, rmin);
                        rmax  = max(RTT, rmax);
                        if (form!=LOG && form!=PARTIAL)
                                fprintf (stderr, "RT time =  %.1f ms\n",
                                        RTT*1000);
                }
        }
        /*--------------------------------------*/
        /*      WRITE STATISTICS                */
        /*--------------------------------------*/
        loss = (int)(.5 + 100.*(count-nr)/count);
        if (form==PARTIAL)
        {
                printf ("%s %d %d %d %.3f",
                        date, bytes, count, nr, dns_sec);
                if (nr >1) printf("\t%.3f %.3f %.3f\n",
                        rmin, rsum/nr, rmax);
                else if (nr==1) printf("\t%.3f\n", rmin);
                else    printf("\n");
        }
        else if (form==LOG)
        {
                printf ("TCPING  %s %s %s %s %d %d %d", date, client_name,
                        server_IP, server_name, bytes, count, nr, dns_sec);
                if      (nr >1) printf("\t%.3f %.3f %.3f\n",
                                rmin, rsum/nr, rmax);
                else if (nr==1) printf("\t%.3f\n", rmin);
                else    printf("\n");
        }
        else
        {
                printf ("\n%d SYN packets transmitted, ", count);
                printf (  "%d RST packets received, ",    nr);
                printf (  "%.0f%% packet loss\n",         loss);
                if (nr)
                {
                        printf ("Round-trip time:   ");
                        printf ("min/avg/max = %.0f/%.0f/%.0f ms\n\n",
                                 1000*rmin, 1000*rsum/nr, 1000*rmax);
                }
        }
}
```

C.6 wrapper.c

```
/*------------------------------------------------------------------------
*
* NETWORK WRAPPER FUNCTIONS for error handling, as per Stevens.
*
*------------------------------------------------------------------------/
#include       "tools.h"

/*----------------------*/
/*      errx(), errx2() */
/*----------------------*/
void    errx  (const char *msg)
{
        if(msg) fprintf(stderr,"\n%s\n\n",msg);
        exit(0);
}
void    errx2 (const char *f, const char *m)
{
        fprintf(stderr,"\n");
        fprintf(stderr,f,m);
        fprintf(stderr,"\n\n");
        exit(0);
}

/*----------------------*/
/*      Gethostname     */
/*----------------------*/
int     Gethostname (char *name, int len)
{
        int     n;
        if ( (n=gethostname(name, len)) <0)
        {
                perror("\ngethostname");
                exit(0);
        }
        return (n);
}
/*----------------------*/
/*      Gethostbyname   */
/*----------------------*/
struct hostent  *Gethostbyname (const char *name)
{
        struct hostent *h;
        if (!(h=gethostbyname(name)))    my_herror (name);
        return (h);
}
/*------------------------------*/
/*      Connect, blocking       */
/*------------------------------*/
int     Connect (int s, struct sockaddr *addr, int addrlen)
{
        int n;
        if ( (n=connect (s,addr,addrlen)) <0) { perror("\nconnect"); exit(0); }
        return(n);
}
/*------------------------------*/
/*      Connect, non-blocking   */
/*------------------------------*/
```

```
int     NB_Connect (int s, struct sockaddr *addr, int addrlen)
{
        int n;
        if ( (n=connect (s,addr,addrlen)) <0)
                if (errno != EINPROGRESS && errno != ECONNREFUSED)
                {
                        perror("\nNon-blocking connect");
                        exit(0);
                }
        return(n);
}
/*----------------------*/
/*      Fnctl           */
/*----------------------*/
int     Fcntl (int s, int command, int arg)
{
        int n;
        if ( (n= fcntl (s, command, arg)) <0)
        {
                if      (command==F_SETFL) perror("\nfcntl (F_SETFL)");
                else if (command==F_GETFL) perror("\nfcntl (F_GETFL)");
                else                       perror("\nfcntl");
                exit(0);
        }
        return(n);
}
/*----------------------*/
/*      Select          */
/*----------------------*/
int     Select (int nf, fd_set *Rs, fd_set *Ws, fd_set *Es,
                struct timeval *tout)
{
        int n;
        if ( (n=select (nf,Rs,Ws,Es,tout)) <0)
        {
                perror("\nselect");
                exit(0);
        }
        return(n);
}
/*----------------------*/
/*      RST_Select      */
/*----------------------*/
int     RST_Select (int nf, fd_set *Rs, fd_set *Ws, fd_set *Es,
                    struct timeval *tout)
{
        int n;
        if ( (n=select (nf,Rs,Ws,Es,tout)) <0)
                if (errno!=ECONNREFUSED) { perror("\nselect"); exit(0); }
        return(n);
}
/*----------------------*/
/*      Socket          */
/*----------------------*/
int     Socket (int family, int type, int protocol)
{
        int n;
        if ( (n=socket (family,type,protocol)) <0)
        {
```

```
                perror("\nsocket");
                exit(0);
        }
        return(n);
}
/*----------------------*/
/*      Fopen           */
/*----------------------*/
FILE*   Fopen (char* path, char* mode)
{
        FILE *fp;
        if (! (fp=fopen (path,mode)))  { perror("\nfopen"); exit(0); }
        return (fp);
}
/*----------------------*/
/*      Read            */
/*----------------------*/
int     Read (int s, char* buf, int len)
{
        int n;
        if ( (n=read (s,buf,len)) <0)  { perror("\nread"); exit(0); }
        return(n);
}
/*----------------------*/
/*      Uname           */
/*----------------------*/
void    Uname (struct utsname *buf)
{
        if ( uname(buf)<0 )  { perror("\nuname"); exit(0); }
}

/*------------------------------------------------------------------------
*
*       my_herror()
*
*       Prints DNS lookup errors in human readable form, similar to
*       perror() printing for system call errors.
*       Functions gethostbyname() and gethostbyaddr() return DNS erros
*       in global variable h_errno.  Error definations and h_errno
*       declaration are in netdb.h.
*
*       Note, Linux and Solaris netdb.h includes a prototype for herror(),
*       but the function is not in libraries -lsocket or -lnsl.
*
*------------------------------------------------------------------------*/
void    my_herror (const char *msg_str)
{
        if (msg_str) fprintf (stderr, "*DNS error for %s: ", msg_str);
        switch (h_errno)
        {
        case HOST_NOT_FOUND:
                printf("Authoritive Answer Host not found\n");
                break;

        case TRY_AGAIN:
                printf("Non-Authoritive Host not found, or SERVERFAIL\n");
                break;

        case NO_RECOVERY:
```

```
                printf("Non recoverable errors, FORMERR, REFUSED, NOTIMP\n");
                break;

        case NO_DATA:i
                printf("Valid name, no data record of requested type\n");
                break;

        default:
                printf("Undefined\n");
                break;
        }
}
```

Bibliography

[1] G. Abdulla, E. A. Fox, M. Abrams, and S. Williams. Shared user behavior on the World Wide Web. In *Proc. of WebNet97*, Toronto, Canada, Oct. 1997.

[2] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching proxies: Limitations and potentials. In *Proc. of the 4th Int'l. World-Wide Web Conf.*, pages 119–133, Dec. 1995.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proc. of the IEEE Conf. on Parallel and Distributed Systems (PDIS'96)*, Dec. 1996.

[4] D. Andresen, T. Yang, V. Holmedahl, and O. H. Ibarra. SWEB: Towards a scalable World Wide Web server on multicomputers. In *Proc. of the 10th Int'l. Parallel Processing Symp. (IPPS'96)*, Honolulu, HI, Apr. 1996.

[5] Apache HTTP Server Project. Available online at http://www.apache.org/.

[6] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *Proc. of ACM SIGMETRICS*, May 1996.

[7] H. Balakrishnam, V. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP behavior of a busy Internet server: Analysis and improvements. In *Proc. of IEEE INFOCOM 98*, San Francisco, CA, Mar. 1998.

[8] H. Balakrishnam, S. Seshan, M. Stemm, and R. H. Katz. Analyzing stability in wide-area network performance. In *Proc. of ACM SIGMETRICS Conf. on Meas. & Modeling of Computer Systems*, Seattle, WA, June 1997.

[9] A. Bestavros. Demand-based document dissemination to reduce traffic and balance load in distributed information systems. In *Proc. of the Seventh IEEE Symp. on Parallel and Distributed Processing (SPDP'95)*, San Antonio, TX, Oct. 1995.

[10] A. Bestavros. WWW traffic reduction and load balancing through server-based caching. *IEEE Concurrency*, pages 56–67, Jan./Mar. 1997.

[11] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, V. Shah, and Z. Fei. Application-layer Anycasting. In *Proc. of IEEE Infocom'97*, Kobe, Japan, Apr. 1997.

[12] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. of IEEE Infocom'99*, Mar. 1999.

[13] T. Brisco. RFC1794: DNS support for load balancing. The Internet Engineering Taskforce, Apr. 1995.

[14] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. of the 1997 USENIX Symp. on Internet Technology and Systems (USITS'97)*, pages 193–206, Dec. 1997.

[15] R. L. Carter and M. E. Crovella. Server selection using dynamic path characterization in wide-area networks. In *Proc. of IEEE Infocom'97,* Kobe, Japan, Apr. 1997.

[16] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical Internet object cache. In *Proc. of the 1996 USENIX Annual Technical Conf.*, San Diego, CA, Jan. 1996.

[17] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic evidence and possible causes. In *Proc. of the 1996 ACM SIGMETRICS Int'l. Conf. on Meas. and Modeling of Computer Systems,* Philadelphia, PA, May 1996.

[18] M. E. Crovella and R. L. Carter. Dynamic server selection in the internet. In *Proc. of the Third IEEE Workshop on the Architecture and Implementation of High Perf. Comm. Subsystems (HPCS'95)*, pages 158–162, New York, NY, Aug. 1995.

[19] C. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of WWW client-based traces. Technical Report TR-95-010, Computer Science Dept., Boston University, Boston, MA, 1995.

[20] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable and highly available web server. In *Proc. of the 41st IEEE Computer Society Int'l. Conf.*, Feb. 1996.

[21] B. M. Duska, D. Marwood, and M. J. Feeley. The measured access characteristic of World-Wide-Web client proxy caches. In *Proc. of the 1997 USENIX Symp. on Internet Technology and Systems (USITS97),* Dec. 1997.

[22] S. G. Dykes, C. L. Jeffery, and S. Das. Taxonomy and design analysis for distributed web caching. In *Proc. of the IEEE Hawaii Int'l. Conf. on System Sciences (HICSS'99),* Maui, HI, Jan. 1999.

[23] S. G. Dykes, C. L. Jeffery, and K. A. Robbins. An empirical evaluation of client-side server selection algorithms. In *Proc. of IEEE Infocom*, pages 1361–1370, Mar. 2000.

[24] S. G. Dykes and K. A. Robbins. A viability analysis of cooperative proxy caching. Available online at http://www.cs.utsa.edu/~sdykes/.

[25] S. G. Dykes, K. A. Robbins, and C. L. Jeffery. Uncacheable documents and cold starts in Web proxy cache simulations: How two wrongs appear right. Available online at http://www.cs.utsa.edu/~sdykes/.

[26] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *ACM SIGCOMM '98,* Vancover, Canada, 1998.

[27] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proc. of IEEE Infocom'98,* Mar. 1998.

[28] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP1.1. The Internet Engineering Taskforce, June 1999.

[29] S. Glassman. A caching relay for the World Wide Web. In *Proc. of the First Int'l. World Wide Web Conf.*, pages 69–76, May 1994.

[30] J. Gwertzman and M. Seltzer. An analysis of geographical push-caching. Available online at http://www.eecs.harvard.edu/~vino/web/.

[31] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proc. of the 1995 Workshop on Hot Operating Systems (HotOS-V)*, pages 51–55, 1995.

[32] J. Gwertzman and M. Seltzer. World-Wide Web cache consistency. In *Proc. of the 1996 Usenix Technical Conf.*, San Diego, CA, Jan. 1996.

[33] V. Jacobson. traceroute. Available online at ftp://ftp.ee.lbl.gov/traceroute.tar.Z.

[34] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., New York, NY, U.S.A., 1991.

[35] D. Johnson and S. Deering. RFC2526: Reserved IPv6 subnet Anycast addresses. The Internet Engineering Taskforce, Mar. 1999.

[36] NASA JPL. Mars pathfinder. Available online at http://mpfwww.jpl.nasa.gov/MPF.

[37] M. Karaul, Y. A. Korilis, and A. Orda. A market-based architecture for management of geographically dispersed, replicated web servers. In *Proc. of the First Int'l. Conf. on Information and Computation Economics (ICE'98)*, pages 158–165, Charleston, SC, 1998.

[38] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server; the NCSA prototype. *Computer Networks and ISDN Systems*, 27:155–164, 1994.

[39] K. Kong and D. Ghosal. Mitigating server-side congestion in the internet through pseudoserving. *IEEE/ACM Trans. on Networking*, 7(4):530–544, Aug. 1999.

[40] Korea National Cache. Available online at http://cache.kaist.ac.kr.

[41] P. Krishnan and B. Sugla. Utility of co-operating Web proxy caches. *Computer Networks and ISDN Systems*, 30(1-7):105–203, Apr. 1998.

[42] J. Lee, H. Hwang, Y. Chin, H. Kim, and K. Chon. Report on the costs and benefits of cache hierarchy in Korea. In *3rd International WWW Caching Workshop*, University of Manchester, June 1998.

[43] W. Li. References on Zipf's law. Available online at http://linkage.rockefeller.edu/wli/zipf/.

[44] R. Malpani, J. Lorch, and D. Berger. Making World Wide Web caching servers cooperate. In *Proc. of the 4th WWW Conf.*, Boston, MA, Dec. 1995.

[45] B. S. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang. Url forwarding and compression in adaptive web caching. In *Proc. of IEEE Infocom 2000*, volume 2, pages 670–678, Tel Aviv, Isael, Mar. 2000.

[46] J. Mogul. Network behavior of a busy web server and its clients. Technical Report Research Report 95/5, Digital Equipment Corporation, Oct. 1995.

[47] A. Myers, P. Dinda, and H. Zhang. Performance characteristics of mirror servers on the Internet. In *Proc. of IEEE Infocom'99*, Mar. 1999.

[48] National Laboratory for Applied Network Research (NLANR). Ircache project. Available online at http://ircache.nlanr.net/.

[49] N. Nishikawa, T. Hosokawa, Y. Mori, K. Yoshida, and H. Tsuji. Memory-based architecture for distributed WWW caching proxy. In *Proc. of the 7th WWW Conf.*, Apr. 1998.

[50] C. Partridge, T. Mendez, and W. Milliken. RFC1546: Host Anycasting service. The Internet Engineering Taskforce, Nov. 1993.

[51] V. Paxson. Internet traffic archive. Available online at http://www.acm.org/sigcomm/ITA/.

[52] V. Paxson and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Trans. on Networking*, 3(3):226–244, June 1995.

[53] J. E. Pitkow. Summary of WWW characterizations. In *Proc. of the 7th WWW Conf.*, Brisband, Australia, Apr. 1998.

[54] P. Rodriguez, A. Kirpal, and E. W. Biersack. Parallel-access for mirror sites in the internet. In *Proc. of IEEE Infocom 2000*, volume 2, pages 864–873, Tel Aviv, Isael, Mar. 2000.

[55] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. *Performance Evaluation Review*, 26(3):44–50, Dec. 1998.

[56] M. Seltzer. Issues and challenges facing the World Wide Web. Available online at http://www.eecs.harvard.edu/~margo/.

[57] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared passive network performance discovery. In *Proc. of the First USENIX Symp. on Internet Technologies and Systems (USITS'97)*, Monterey, CA, Dec. 1997.

[58] Squid Internet Object Cache. Available online at http://squid.nlanr.net/.

[59] W. R. Stevens. *UNIX Network Programming, Volume 1*. Prentice Hall PTR, Upper Saddle River, NJ, U.S.A., second edition, 1998.

[60] A. S. Tannenbaum. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, NJ, U.S.A., third edition, 1996.

[61] TERENA. WWW cache coordination for Europe. Available online at http://www.terena.nl/task-forces/tf-cache/.

[62] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design considerations for distributed caching on the Internet. In *Proc. of the Int'l. Conf. on Distributed Computing Systems (ICDS'99)*, 1999.

[63] R. Tewari, H. M. Vin, A. Dan, and D. Sitaram. Resource-based caching for Web servers. In *Proc. of the SPIE/ACM Conf. on Multimedia Computing and Networking (MMCN)*, San Jose, CA, Jan. 1998.

[64] Uppsala University. Uppsala University web-cache statistics. Available online at http://netstat.uu.se/Stat/Squid/.

[65] D. Wessels and K. Claffy. RFC 2186: Internet cache protocol (ICP), version 2. The Internet Engineering Taskforce, Sep. 1997.

[66] D. Wessels and K. Claffy. RFC 2187: Application of internet cache protocol (ICP), version 2. The Internet Engineering Taskforce, Sep. 1997.

[67] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of Web-object sharing and caching. In *Proc. of the 2nd USENIX Symp. on Internet Technologies and Systems (USITS'99)*, Oct. 1999.

[68] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative Web proxy caching. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, Dec. 1999.

[69] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using smart clients to build scalable services. In *Proc. of the 1997 USENIX Annual Tech. Conf. (USENIX'97)*, pages 105–117, Anaheim, CA, 1997.

[70] L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web caching. In *Proc. of the 2nd Web Cache Workshop*, Boulder, CO, June 1997.

[71] G. K. Zipf. Relative frequency as a determinant of phonetic change. *Reprinted from the Harvard Studies in Classical Philiology*, 1929.

Vita

Sandra Kay Goles Dykes was born in Waco, Texas on January 3, 1955, the daughter of Thomas Stanley Goles and Mary Alice Hunter Goles. She attended Roosevelt High School in San Antonio, Texas, graduating in 1972. In 1976, she graduated cum laude from the University of Texas at Austin, receiving the Bachelor of Science with a major in chemistry. In the same year she married James R. Dykes, Jr, of Dallas, Texas, a professor at the University of Texas at San Antonio (UTSA). She entered the graduate program at UTSA and received the Master of Science degree in chemistry in 1979. During the next several years, she taught chemistry courses at UTSA. From 1983 to 1987, she worked as a research scientist at H. Dell Foster Associates in San Antonio, Texas, on a project in computer vision. In 1994 she received a Master of Science degree in computer science from UTSA and entered the computer science Ph.D. program the following year. She is a member of IEEE and ACM, is a recipient of NSF and NASA/TSGC fellowships, and was named in *Who's Who in American Colleges and Universities.*

James and Sandra have two children, Travis Alexander, born in August, 1987, and Sean Thomas, born April, 1989. They currently live in Boerne, Texas.

Papers in Computer Science

Sandra G. Dykes, Kay A. Robbins, and Clinton L. Jeffery, "A Viability Analysis of Cooperative Proxy Caching", Available online at http://www.cs.utsa.edu/~sdykes/.

Sandra G. Dykes, Kay A. Robbins, and Clinton L. Jeffery, "Uncacheable Documents and Cold Starts in Web Proxy Cache Simulations: How Two Wrongs Appear Right", Available online at http://www.cs.utsa.edu/~sdykes/.

Sandra G. Dykes, Clinton L. Jeffery, and Kay A. Robbins, "An Empirical Evaluation of Client-side Server Selection Algorithms", *in Proceedings of IEEE Infocom 2000*, Vol. 3, March, 2000, pp. 1361-1371.

Sandra G. Dykes, Clinton L. Jeffery, and Samir Das, "Taxonomy and Design Analysis for Distributed Web Caching", *in Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS '99)*, January, 1999, p. 286.

Sandra G. Dykes, “Cooperative Web Caching Using Server-Directed Proxy Sharing”, *Technical Report CS-98-01,* University of Texas at San Antonio, Division of Computer Science, San Antonio, TX 78249-0664, April, 1998.

Clinton L. Jeffery, Sandra G. Dykes, Xiaodong Zhang, Guillermo H. Gonzalez, and Jason L. Peacock, “Nova Visualization for Optimization of Data-Parallel Programs”, *in Proceeding of Euro-Par’97*, July, 1997.

Xiaodong Zhang, Sandra G. Dykes and Hong Deng, “Distributed Edge Detection Methods and Their Communication Pattern Implications”, *Computational Science and Engineering*, Vol. 4, Number 1, January-March, 1997, pp. 72-82.

Sandra G. Dykes, Xiaodong Zhang, Yi Shen, Clinton Jeffery, and Devin W. Dean, “*GRAPH: A Tool for Visualizing Communication and Optimizing Layout in Data-Parallel Programs”, *in Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, Vol. II, pp. 121-129, August, 1995.

Sandra G. Dykes and Xiaodong Zhang, “Folding Spatial Image Filters on the CM-5”, *in Proceedings of the 9th International Parallel Processing Symposium (IPPS)*, pp. 451-456, April, 1995.

Sandra G. Dykes, “An Efficient Data-Parallel Algorithm for 2-D Convolutions”, *Master’s Thesis*, University of Texas at San Antonio, December, 1994.

Sandra G. Dykes and Bruce E. Rosen, “Parallel Very Fast Simulated Reannealing by Temperature Block Partitioning”, *in Proceedings of the 1994 IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 2, pp. 1914-1919, October, 1994.

Sandra G. Dykes, Xiaodong Zhang, Yan Zhou, and Haixu Yang, “Communication and Computation Patterns of Large Scale Image Convolutions on Parallel Architectures”, *in Proceedings of the 8th International Parallel Processing Symposium (IPPS)*, pp. 926-931, April, 1994.