Sharing and Caching Characteristics of Internet Content

Alastair Wolman

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2002

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Alastair Wolman

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

_____

Henry M. Levy

Reading Committee:

_____

Henry M. Levy

_____

Anna R. Karlin

_____

Steven D. Gribble

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted “the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.”

Signature_____

Date_____

University of Washington

Abstract

Sharing and Caching Characteristics of Internet Content

by Alastair Wolman

Chair of Supervisory Committee:

Professor Henry M. Levy
Computer Science and Engineering

To improve the performance of Internet content delivery, many techniques exploit sharing: repeated requests to the same object by multiple clients. One widely deployed technique is Web proxy caching, where requests to shared objects are served from a proxy cache instead of the origin server. In this dissertation, we present a network tracing system that enables the study of application-level Internet workloads, and we present three Internet caching studies performed using workloads collected by the tracing system.

The first study investigates Web document sharing patterns from an *organizational* point of view. We explore the extent of document sharing both within and across organizations. We find that when clients are members of the same organization, the amount of sharing increases measurably when compared with clients that are members of different organizations. However, this increase is not large enough to have a significant impact on cache performance.

The second study explores the performance of cooperative Web proxy caching, focusing on the effectiveness of cooperation over a wide range of client population sizes. Allowing proxy caches to cooperate effectively combines the client populations served by those proxies. This provides new opportunities for sharing, and therefore offers the potential to increase cache hit rates. Overall, we find that proxy cooperation provides significant performance benefits only within limited population bounds.

The final study is motivated by the increasing availability of multimedia Internet content, such as

streaming audio and video. We compare the workload characteristics of streaming-media content to traditional Web content, and we evaluate the effectiveness of proxy caching and multicast delivery for streaming-media content. We find that these multimedia workloads exhibit strong temporal locality, and we quantify the benefit it provides for caching and multicast delivery.

Finally, we present the design and implementation of our trace collection system. It uses passive network monitoring to observe all Web traffic generated by the University of Washington client population. Our system employs anonymization safeguards to protect users' privacy. It has been deployed at the University network border for three years, and has scaled to handle a factor of three load increase during that period.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere appreciation to everyone who made my time in graduate school rewarding and enjoyable. First and foremost, thanks to my wife Yvonne Kania Wolman, who showed tremendous support and patience and also helped me improve the quality of my writing.

I would like to thank Hank Levy for his guidance, support, and his restrained style of encouragement. I learned a tremendous amount from him, and I really enjoyed working with him. Thanks to Brian Bershad for many hours of entertaining discussions (mostly arguments, really) during our weekly meetings, and for his leadership on the Etch project. Thanks also to the many other terrific faculty at UW whom I had the pleasure of working with, especially Anna Karlin, Jean-Loup Baer, Ed Lazowska, and Steve Gribble.

A huge thanks to Geoff Voelker for being a great friend and a terrific collaborator. Thanks to all the other students who contributed to the UW Web workload analysis project including Maureen Chesire, Neal Cardwell, Nitin Sharma, Tashana Landray, Denise Pinnel, and Molly Brown. Thanks to the entire Etch team: Geoff, Dennis Lee, Ted Romer, Wayne Wong, Hank, Brian, and Brad Chen. Thanks to the many terrific students I got to know during my time at UW including Ruth Anderson, Brad Chamberlain, Sung Choi, Ben Dugan, Marc Fiucynzki, Jim Fix, Kevin Hinshaw, E Lewis, Kurt Partridge, Mike Perkowitz, and Stefan Savage.

Finally, thanks to Steve Corbato, Terry Gray, David Richardson, and the many other people at the UW Computing and Communications department who generously provided assistance in collecting our trace data.

Three of the chapters in this thesis have been previously published as conference papers. Chapter 3 is based on the paper “Oranization-based Analysis of Web-Object Sharing and Caching” published in the *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems* [Wolman et al. 99a]. Chapter 4 is based in part on the paper “On the scale and performance of cooperative Web proxy caching” published in the *Proceedings of the Seventeenth Symposium on Op-*

erating Systems Principles [Wolman et al. 99b]. Chapter 5 is based on "Measurement and Analysis of a Streaming-Media Workload" published in the *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems* [Chesire et al. 01].

Chapter 1

Introduction

In recent years, the World Wide Web has become the most popular application for networked computer systems. In July of 2001, approximately 60% of all households in the United States had access to the Internet [Nielsen 01]. A February 2000 study of an Internet traffic exchange point showed that HTTP accounted for 58% of all the traffic at that backbone location [McCreary et al. 00]. Furthermore, users are no longer just accessing static Web content over the Internet. Access to multimedia content such as streaming audio and video clips is also becoming commonplace.

Given the widespread usage of the Web, getting acceptable performance for interactive Web browsing is seen as important by content providers, ISP network managers, and end users. To that end, there has been a great deal of recent focus on improving the mechanisms used to deliver Internet content from servers to clients. One of the key mechanisms currently used to improve the delivery of static Internet content is *proxy caching*. For streaming multimedia content, both proxy caching and multicast delivery offer the potential to significantly reduce network load.

This dissertation presents the design and implementation of a network tracing system for measurement and analysis of Internet applications. We demonstrate the utility of this system by performing a set of three Internet traffic studies. The common theme across these studies is their focus on *sharing*. Sharing occurs when multiple clients make accesses to the same Internet content over time. Sharing plays an important role in many techniques used to improve the delivery of Internet content, such as proxy caching and multicast delivery. For example, the amount of sharing that occurs among a group of clients places an upper bound on the hit rate that a proxy cache serving those clients can achieve.

1.1 Caching for the World Wide Web

Caching is a prevalent technique used to improve performance throughout centralized and distributed computer systems. A cache is simply a high-speed memory used to store frequently accessed objects. For the World Wide Web, there are a number of different types of caches in use. A browser cache typically uses a combination of RAM and local disk to store recently used pages on behalf of one user. A proxy cache typically is a dedicated machine that caches pages on behalf of a *group* of users, and these users often share a connection to the Internet. The nodes of a content distribution network are another type of caching used on the Web today. Content distribution network caches (CDN caches) are similar to proxy caches in that they are dedicated machines that store frequently accessed Web pages. There are two key differences: CDN caches only store pages from those sites that pay for the content distribution service, and the mapping of users to CDN caches is dynamic rather than static.

There are three fundamental benefits of caching for the World Wide Web:

- When Web documents are downloaded from a cache rather than the origin server, there is the potential to significantly reduce the delay that the user experiences waiting for those documents to appear in the Web browser window. The latency benefit of caching depends on the cache hit ratio, the cache load, and the network connectivity between the client and the cache.

- When an organization purchases a link to connect to the Internet, a Web proxy cache can significantly reduce the amount of traffic on that link. This can help to reduce congestion delays, and it may allow the organization to reduce the cost of the link.

- When pages are fetched from a cache, the amount of traffic on the Internet links between the cache and the origin server is reduced. This cuts down on Internet backbone congestion and server load. One of the earliest motivations for deploying proxy caches was to eliminate Internet hot spots that arise when an unusual event occurs.

The actual benefits obtained from Web caching are very dependent on the location of caches and the browsing behavior of Web users. In order to understand the true benefits of caching, workload

analysis must be done to better understand cache hit ratios and the latency and bandwidth savings that result from cache hits.

1.2 Workload Characteristics

The Web is a complex distributed system, and one of the key design principles for improving the performance of any complex system is to optimize for the common case. This principle motivates the study of Web workload characteristics. Understanding workload characteristics helps the designers of Web software and protocols to choose which problems to solve, and also allows developers to evaluate and predict the performance of their solutions in the lab without fully deploying those solutions.

The task of both collecting and analyzing Web workloads is a difficult endeavor. The size and diversity of the Internet makes it difficult to collect workloads that are truly representative of the entire network. Furthermore, the core Internet infrastructure was not designed to support easy collection of detailed measurements. To make the problem even worse, the Web is a moving target. Rapid growth in Web usage, rapid changes to the software for all components that interact directly with the network (such as Web servers, browsers, and caches), and even changes to the protocol standards, all pose the challenge that characteristics observed in today's workloads may not hold true in a few years.

One of the key contributions of this dissertation is the design, implementation, and experience with deploying a scalable application-level network measurement system. This system uses passive network monitoring to observe the Web behavior of a large and diverse group of clients. It is installed at the Internet border of the University of Washington (UW) and it observes all Internet traffic that flows through the border. The traffic studies in this dissertation focus on UW as a large client population. Therefore, we filter the workload to examine only those requests that originate from clients inside UW directed towards external servers and those responses that originate outside UW directed towards internal clients.

The tracing system was in use at the UW Internet border for approximately three years, from the Summer of 1998 through the Spring of 2001. During that time period, the traffic load grew by a factor of three and the tracing system scaled to keep up with the load increases with negligible

packet loss. The tracing system makes extensive use of safeguards to protect the privacy of users at UW whose Web behavior is observed. The tracing system has been used to study the characteristics of two application-level Internet protocols, namely HTTP and RTSP. HTTP, the Hypertext Transfer Protocol, is used to transfer most static Web content, such as HTML pages and images. RTSP, the Real-Time Streaming Protocol, is the most commonly used standard for delivering streaming audio and video.

1.3 Workload Studies

We performed a set of three workload studies using the aforementioned tracing system. In each of these studies, we focus on those workload characteristics that have a significant impact on the effectiveness of techniques designed to improve the delivery of Internet content. For traditional static Web content, the performance optimization that we focus on is Web proxy caching. For streaming multimedia content, we focus on both proxy caching and multicast delivery.

Although there are many different workload characteristics that affect the performance of caching systems (e.g. the size of the documents transferred, the percentage of requests that are allowed to be cached, or the rate at which changes are made to Web documents), the amount of sharing among clients is a key workload characteristic because it places an upper bound on the hit rate that a proxy caching system can achieve. Sharing is important not only because of its direct applicability to proxy caching, but also because many other proposed performance optimizations depend on the amount of sharing in the workload. Examples include content distribution networks, multicast delivery for streaming media, network performance databases for server selection, and sharing of TCP congestion control information across hosts.

1.3.1 Organizational Sharing

The first workload study in this dissertation investigates Web document sharing and caching characteristics from an *organizational* point of view. An organizational analysis of sharing is important because Web proxy caching is typically performed on an organizational basis. For example, Web proxies are often placed in front of large and small companies, universities, departments, and so on. The primary goal of this study is to explore the extent of document sharing among clients

both within and across multiple organizations. We also study how often accesses are made to uncachable documents, and whether the sharing characteristics differ for cachable versus uncachable documents.

To perform these kinds of analyses, one needs simultaneously collected traces of multiple diverse organizations. Such traces are not currently available, and are difficult to obtain in practice. To solve this problem, we use organizations inside UW to model organizations connecting to the Internet. We identify 170 internal organizations at UW and we classify each client in our trace with an organization identifier. For example, the organizations at UW include the English Department, the School of Dentistry, and the undergraduate dormitories. We collect a trace of all Web requests and responses from UW that protects the privacy of users, yet preserves this organizational membership information.

1.3.2 Cooperative Web Proxy Caching

The second study explores the performance of cooperative Web proxy caching, focusing on the effectiveness of cooperation over a wide range of client population sizes. One of the key factors that limits the hit rate for a given Web proxy cache is the total size of the client population that it manages. One way to increase the client population size is to have multiple proxy caches that cooperate with each other. The increased population size provides new opportunities for sharing, and therefore offers the potential to increase proxy cache hit rates. This study uses trace-based analysis to investigate the potential advantages and drawbacks of inter-proxy cooperation. With our traces, we can quantitatively evaluate the performance-improvement potential of cooperation between 170 small-organization proxies within the UW environment. We also collect a simultaneous trace of the Microsoft campus proxy caches, which allows us to evaluate the performance improvement of cooperation between two large-organization proxies handling 23,000 and 60,000 clients, respectively.

1.3.3 Streaming Media Content

The motivation behind our final study is the increasing availability of multimedia Internet content, such as streaming audio and video. For example, almost all the popular news, sports, and entertainment Web sites offer streaming video clips. Also, audio clips are becoming commonplace in many

forms such as Internet radio stations, peer-to-peer networks for music trading, and Internet retailers that sell tapes and CDs. Compared with traditional Web workloads, these multimedia objects may require significantly more storage and transmission bandwidth. As a result, performance optimizations such as streaming media proxy caches and multicast delivery offer the potential to minimize the impact of delivering this content over the Internet. However, few studies of streaming-media workloads exist. Therefore, the extent to which these mechanisms will improve performance is unclear. This study presents and analyzes a client-based streaming-media workload generated by a large university. Where possible, we compare the characteristics of our streaming-media workload to traditional Web-object workloads. We also evaluate the effectiveness of proxy caching and multicast delivery for streaming-media content.

1.4 Contributions

The contributions of this dissertation are the following:

- We explore document sharing from the perspective of organizations. When clients are members of the same organization, we find there is a measurable increase in the amount of sharing compared with clients that are members of different organizations. However, this increase is not large enough to have a significant impact on cache performance. We also explore the impact of requests to uncachable documents – documents that the HTTP protocol rules state are not allowed to be served by a proxy cache. We find that 40% of all the requests in our workload are uncachable – a significantly higher percentage than previous studies reported.

- We explore the effectiveness of cooperative Web proxy caching for a range of client population sizes, and find that the largest benefit is achieved at relatively small population sizes (on the order of 2000 to 5000 clients). We use simultaneous traces of two large organizations, UW and Microsoft, to confirm that combining large organizations provides a marginal benefit. Combining populations of this size (each has more than 20,000 clients) makes sense only in very high-bandwidth and low latency environments.

- We analyze the RTSP client activity generated by a large university. We study the properties of our streaming media workload and compare it where possible to the properties of

well-studied HTTP workloads. We also explore the effectiveness of two performance optimizations – proxy caching and multicast delivery. We find that the median size of streaming media downloads is 400 times larger than the median HTTP request size, and the mean RTSP download size is 175 times larger than the mean HTTP size. We observe a number of properties that indicate multimedia workloads would currently benefit less from proxy caching than HTTP workloads. On the other hand, we find that streaming media workloads exhibit surprisingly strong temporal locality, which indicates that multicast delivery will be an effective technique to reduce network load.

- We design and implement a network tracing system that allows us to observe the Web behavior of a large client population with relatively low overhead. Our contributions in developing this system are described above (Section 1.2).

1.5 Overview

The rest of this dissertation is organized as follows:

- Chapter 2 begins with additional background information about the HTTP protocols and support for proxy caching. It then provides an overview of related research.

- Chapter 3 examines both sharing and caching characteristics of Web workloads from the perspective of organizations.

- Chapter 4 examines issues of scale for cooperation among Web proxy caches.

- Chapter 5 examines the characteristics of a new and increasingly popular kind of Internet content: streaming media.

- Chapter 6 provides an overview of the design and implementation of the tracing system we use to perform the workload analyses in Chapters 3, 4, and 5.

- Finally, Chapter 7 summarizes the contributions of this dissertation and concludes.

Chapter 2

Background and Related Work

This dissertation focuses on the characteristics of Web workloads that have a large impact on the efficiency of Web content delivery to clients. In this chapter, Section 2.1 provides background information on the protocols and software components used for Web content delivery, and Section 2.2 investigates related research in the areas of Web workload collection and characterization.

In the background section, we provide an overview of the basic interactions between Web clients, proxies, and servers. We begin with an overview of the HTTP transport protocol, including some of the important differences between HTTP versions 1.0 and 1.1. We then introduce the basics of Web caching and examine the reasons why some Web requests and responses are not cachable. Next, we examine the explicit support for caching defined by the HTTP protocol specifications. We conclude with an overview of the RTSP transport protocol used for delivering streaming media.

In the related research section of this chapter, our focus is on research that studies the performance of Web caching. There are three key factors that make workload characteristics related to Web caching the most widely studied aspects of Web workloads up to this point. First, the design of algorithms for both Web proxy caching and cooperative Web proxy caching has been an extremely active research area in recent years [Chankhuntod et al. 96, Zhang et al. 97, Touch 98, Fan et al. 98, Karger et al. 99, Tewari et al. 99]. Second, a great deal of interest in caching systems for the Web is due to the effectiveness of caching as a performance improvement technique in other areas of computer systems, such as filesystems and CPU architecture. Third, Web caching and content distribution systems are the most widely deployed techniques used today to improve Web performance.

In order for proxy caching to be effective, the workload must have the following properties: many of the requested documents were also requested at some point in the past; those documents did not change in between the repeated accesses; and the caches were allowed to store those documents in the first place. Therefore, we identified three workload properties that are most critical to the overall effectiveness of Web caching – the cachability of Web documents, the rate of changes to

Web documents, and the extent of Web document reuse. In Sections 2.2.1 through 2.2.3, we cover the related research on each of these three topics in detail.

We also investigate related research on three additional topics: cooperative Web proxy caching, characteristics of streaming media traffic, and systems for collecting Web workloads.

2.1 Background

In this section, we provide background information needed to understand the basic interactions between Web clients, proxies, and servers. We begin with an overview of the HTTP transport protocols. We then introduce the basics of Web caching, and next we describe cookies, a mechanism that allows Web servers to store persistent information at client browsers. We look at the many different reasons why HTTP requests and responses may be uncachable. Next, we describe the explicit support for caching in the HTTP protocol specifications. Finally, we present an overview of the RTSP protocol which is used to deliver streaming media content.

2.1.1 The HTTP Protocols - 1.0 and 1.1

HTTP, the Hypertext Transfer Protocol, is the protocol that Web browsers, proxies, and servers use to transfer most Web documents. The first officially standardized version of HTTP was called HTTP 1.0 [Berners-Lee et al. 96]. HTTP 1.0 is a simple request-response protocol. The most recent standard for HTTP is version 1.1 [Fielding et al. 99], which contains some significant enhancements to HTTP 1.0. In this section, we provide an overview of the basic operation of HTTP and summarize the key differences between the different versions.

HTTP is an application-level protocol, with protocol messages encoded in the human-readable ASCII format. The message format for HTTP is based on the Multipurpose Internet Mail Extensions (MIME) specification. HTTP must be layered on top of a reliable byte-stream transport protocol, and for current implementations this means TCP. Figure 2.1 shows an example of the headers for a simple HTTP 1.0 style request and the corresponding response. In this example, the method for the request is “GET”, and the URL of the image being fetched is “http://s1.abcnews.go.com/ad/sponsors/news-120.gif”. The Web server responds with a document whose content type is “image/gif” and whose size is 2869 bytes. For the request, there is no data that

```
GET /ad/sponsors/news-120.gif HTTP/1.0
User-Agent: Mozilla/4.7 [en] (WinNT; U)
Host: s1.abcnews.go.com
Referer: http://abcnews.go.com/sections/business/economy.html
Accept: image/gif, image/x-xbitmap, image/jpeg, image/png
Accept-Encoding: gzip
Accept-Language: en
```

```
HTTP/1.0 200 OK
Server: Microsoft-IIS/4.0
Date: Mon, 28 Feb 2000 20:55:01 GMT
Content-Type: image/gif
Last-Modified: Thu, 24 Feb 2000 23:02:31 GMT
Content-Length: 2869
```

Figure 2.1: An example HTTP request (top) and response (bottom).

follows the headers (though methods other than GET do sometimes have data), and for the response the image data is sent immediately after the headers over the same TCP connection.

One of the most significant differences between HTTP 1.0 and HTTP 1.1 is the use of persistent connections. In HTTP 1.0, the Web browser sets up a TCP connection to the Web server, and it sends a single request over that connection. The server sends its response over that same connection, and when the response is complete then the connection is closed. What appears to the user as a single Web page usually consists of many embedded documents, such as images, frames, style sheets, and Javascript code. Each embedded document is transferred over its own HTTP 1.0 TCP connection. With HTTP 1.1, the browser may request more than one document over the same TCP connection, as well as pipeline the requests rather than wait for each response before sending the next request. The only constraint is that requests and responses do not contain sequence numbers, so the server must send the responses in the same order that the client sent the requests. Either side of the connection may terminate the persistent connection by attaching the “Connection: Close” header to a request or response.

2.1.2 Caches for the Web

There are three widely deployed forms of caching in the Web today: (1) browser caches, (2) proxy caches, and (3) content distribution network (CDN) caches. All popular Web browsers today support both memory and disk-based caches. Most browsers allow users to control the amount of resources allocated to the cache as well as the policy used by the cache to determine when to revalidate cached documents. Browser caches handle Web document reuse by a single user.

A proxy cache is a network service for caching Web objects. Just as browser caches handle reuse for a single user, proxy caches handle reuse for a group of users because they can be simultaneously accessed and shared by many clients. The proxy cache acts as an intermediary between Web clients and servers. Cache hits at a proxy provide the following potential benefits: reduced latency for users browsing the Web, reduced load for the origin Web servers, and reduced network traffic for the portion of the network between the proxy cache and the origin server. Misses at a proxy cache can increase latency for users because of the additional cost of communicating with the proxy cache before contacting the origin server.

CDN caches are similar to Web proxy caches. They are typically dedicated machines that serve large groups of users, and they act as intermediaries between clients and servers. CDN caches also differ from proxy caches in a number of ways. CDN caches only store the pages of sites that pay for the content distribution service. Furthermore, the naming of documents at CDN caches is explicit rather than implicit. For proxy caches, there is nothing specific about a URL that indicates the page is being served by a proxy cache. For CDNs, the URL of each document served by a CDN contains a hostname that is part of the CDN administrative domain rather than using the hostname of the origin server. It is up to the CDN internally to decide how to determine what the origin server is and how to contact it if necessary. The Internet Domain Name System (DNS) is a distributed naming service used to translate hostnames into IP addresses. Since the DNS is used to locate CDN caches, a commonly deployed trick in these networks is to dynamically map users to CDN caches based on the current cache load or network conditions. By setting a short time-to-live value for the DNS entry that maps the CDN cache name to an IP address, the CDN system can update that DNS entry on a frequent basis to implement the dynamic mapping of users to CDN caches.

2.1.3 Cookies

Cookies are a general purpose state management mechanism for the Web. Cookies allow a server-side Web application to store persistent information at the browser and later retrieve that information. One use for cookies is to implement a shopping cart application. For this application, a cookie stores a unique identifier for each user's browsing session. This unique identifier provided by the cookie maps directly to the user's shopping cart. The user can then make multiple selections at different Web pages in the online store, each of which gets placed in the shopping cart. At check-out time, the cookie locates the shopping cart and then the application can determine the cost of all items in the cart. Cookies are not mentioned in the HTTP 1.0 specification, but they were first defined and implemented with Netscape V1.0 [Netscape 95]. According to the Netscape specification, requests with a "Cookie" header should never be cached, and responses with a "Set-Cookie" header should also not be cached. Cookies were later standardized in [Kristol et al. 97]. That specification requires the use of the HTTP 1.1 "Cache-control" headers to control the behavior of proxy caches.

2.1.4 Uncachable Documents

In any caching system, there is a need to keep the cached copies of data consistent with the original location of the data. When the data is modified, the caching system must ensure that all copies of the data are updated in a timely manner. A caching system is said to maintain strong consistency if there is no possibility that a client can ever see a stale copy of the data. The Web uses a weak consistency model where there is the possibility that a Web cache may serve a stale copy of a document to clients if that document has recently changed at the origin Web server.

The principal benefit of using a weak consistency model is better availability. If strong consistency is used then one must ensure that a cached copy of a document is the most recent version before allowing a client to read the cached document. If the origin server for that document is unavailable, then there is no way to perform that check. Many techniques have been developed to help increase availability while maintaining strong consistency (e.g. locking, leases, and replication), but fundamentally one can always provide higher availability with weak consistency.

Certain uses of the Web do not interact well with the weak consistency semantics of the Web. For example, suppose the content of a Web page for an online store is computed by querying a product

inventory database. Without caching, a database query would run each time a client accesses that URL. If a proxy cache stores that Web page, then the cache actually stores the results of a past query to the database. There is a strong possibility that these results may be out of date. Web documents such as the one in this example are commonly referred to as *dynamic* Web documents. Because there are certain Web applications where caching simply gets in the way and does not provide a performance benefit, both Web clients and Web servers provide mechanisms to bypass intermediate Web caches. For Web clients, an uncachable request is one that must be passed along to the origin server, even if an intermediate proxy cache has what it believes to be a valid copy of the requested document. For Web servers, an uncachable response is one that should not be stored at any intermediate proxy caches.

2.1.5 HTTP 1.0 Support for Caching

In HTTP 1.0, there are three features that support Web document caching. The “Pragma” header is used to attach implementation specific directives that apply to any recipient in the request/reply chain [Berners-Lee et al. 96]. The “no-cache” pragma explicitly defines that requests with this header should only be handled by the origin server, whether or not any intermediary cache has the requested document. As defined in the standard, this mechanism applies only to requests, so it does not provide a mechanism for content providers to specify that their documents are dynamic and should not be cached. In practice, however, most proxy caches support the “no-cache” pragma for response messages as well as request messages.

Another explicit element of support for caching in HTTP 1.0 is the conditional “GET” method. A GET request is conditional if the “If-Modified-Since” header is included with the request. The semantics of a conditional GET allow the server to only transfer the requested document if it was modified after the date specified in the “If-Modified-Since” header. The Web server attaches the document modification time to the response with the “Last-Modified” header. Conditional requests are sometimes referred to as cache validation requests because they can be used by caches to verify that the copy of a document stored in the cache is still valid.

The final feature is the “Expires” header. This field specifies the date and time after which the document should be considered stale, which implies that caches should either discard or revalidate

the document. The specification further defines an additional mechanism to prevent dynamic documents from being cached. If the server sets the time in the “Expires” header to be earlier than the time in the “Date” header then recipients of that response must not cache the document.

2.1.6 HTTP 1.1 Support for Caching

HTTP 1.1 introduces more substantial support for proxy caches and eliminates many of the ambiguous situations that arise when using a proxy cache with HTTP 1.0.

The most significant addition to the HTTP 1.1 specification is the “Cache-control” header. This header supports a number of different mechanisms; browsers or servers can add this header allowing either party to control the behavior of any intermediary caches. The “Cache-control” directives may be used to accomplish any of the following tasks: to impose or eliminate restrictions on what is cachable, to modify the default expiration mechanisms, to control cache revalidation strategies, and to control document transformations. The controls also distinguish between private caches (browser caches) and shared caches (proxy caches). Furthermore, the “Cache-control” directives may be applied to individual header fields as well as to whole requests or responses. For example, this feature is used to enable caching for documents that contain the “Set-Cookie” header, so that only the “Set-Cookie” header field is excluded from the cache.

The conditional GET technique of HTTP 1.0 is extended to support entity tags (using the “If-Match” or “If-None-Match” headers) in addition to dates (“If-Modified-Since” or the new “If-Unmodified-Since”). An entity tag is a compact unique identifier for an object and is specified with the “Etag” header. A typical implementation for entity tags is a document checksum. Furthermore, the “If-Range” header changes a GET request to only fetch a portion of the document rather than the entire document. This is useful when poor network connectivity leads to frequent connection failures.

Although we typically think that the content of a document is uniquely specified by its URL, this is in fact not the case. Both HTTP 1.0 and 1.1 include support for content negotiation with a set of “Accept” headers that allow the browser to specify preferred languages, character sets or encodings for the content. This behavior introduces difficulties for implementing caching that are not addressed by HTTP 1.0, because potentially all documents could be using content negotiation. HTTP 1.1

introduces a new header field called “Vary” which must be added to a response whenever the server uses additional request header fields to determine the content. “Vary” also informs the caches as to which header fields determine the content. This allows caches to serve content negotiated documents if those headers match in addition to the URL. Most current proxy cache implementations treat responses with a “Vary” header as uncachable.

The HTTP 1.1 specification also provides guidance on how proxy caches should implement weak cache consistency. It defines techniques to calculate the age of a document and it recommends a heuristic for calculating the freshness lifetime for documents that do not specify explicit expiration times. These heuristic freshness lifetimes cannot be used for documents that do not supply a “Last-modified” header. HTTP 1.1 also requires that a cache must attach a warning header to any document served from the cache where that document is more than 24 hours old and where the document’s freshness lifetime is calculated heuristically. Finally, any other stale documents served from a cache must also have a warning header attached.

2.1.7 RTSP: The Real-Time Streaming Protocol

In contrast with traditional Web pages, which are almost always delivered using HTTP, there is no single protocol that is used to deliver all streaming-media content. Instead, a number of commercial applications support a variety of standard and proprietary protocols. Given the diversity of protocols in use, in this dissertation we focus on the standardized and well documented RTSP protocol [Schulzrinne et al. 98]. RTSP is one of the most widely used streaming protocols. For example, RealNetworks’ RealPlayer and Apple’s QuickTime Player are two widely used streaming media client applications that support RTSP.

The RTSP protocol is used to setup and control the delivery of one or more continuous media streams. Media objects are identified by an RTSP URL, with the prefix “rtsp:”. Conceptually, RTSP distinguishes between control traffic and media-data traffic, and allows for control traffic and media-data traffic to be sent over different channels. The protocol also allows for interleaving of control traffic and media-data traffic on the same channel. For most RTSP implementations, the control channel is layered on top of TCP, but the TCP congestion control algorithm can potentially get in the way of low latency delivery of the media data. Therefore, most RTSP streaming-media servers

use UDP to deliver the continuous media data.

The RTSP protocol syntax uses the MIME header format in a manner very similar to HTTP, although the operations supported by RTSP are different. Another key difference between RTSP and HTTP is that RTSP is a stateful protocol, which means that RTSP servers must keep track of the current state of each session. RTSP uses sequence numbers to match requests and responses. In a typical sequence of interaction, an RTSP client sends a SETUP request to the server which causes the server to allocate resources for the stream. When the server receives a PLAY request, it begins to transmit the stream data. A PAUSE request halts the data transmission without freeing the server resources, and a TEARDOWN request terminates the session and frees the server resources. Although RTSP and HTTP are both request-response protocols, RTSP also differs from HTTP in that both clients and servers can initiate requests. For example, an RTSP server can query a client using the GET_PARAMETER request to discover packet reception characteristics for the current stream, such as packet loss rate or jitter.

2.2 Related Research

In this section, we summarize related research in the areas of Web workload collection and workload characterization for Web caching. In Sections 2.2.1 through 2.2.3, we look at the three workload characteristics that have the greatest impact on the effectiveness of Web caching systems. In Sections 2.2.4 through 2.2.6, we look at previous research in the areas of cooperative caching, streaming media workloads, and passive network monitoring.

2.2.1 Cachability of Web Documents

Document cachability refers to the question of under what conditions a cache may store a given Web document. In this section, we provide an overview of the research characterizing how often and why Web documents are uncachable.

One of the earliest projects to point out many of the interesting deployment issues for Web caching was the Harvest system [Chankhuntod et al. 96]. Their paper demonstrated that deploying caches for Web traffic during the 1994 and 1995 time-frame was difficult because there was no standard for specifying objects as uncachable. Harvest used string matching on the URL name to

detect uncachable CGI scripts. They also showed that HTTP content negotiation effectively makes the deployment of correctly behaving transparent caches impossible. The problem is that Web servers may use the request header contents to compute the HTTP response, so a correct Web cache must match all HTTP request headers in addition to the URL in order to get a cache hit. The frequent usage of optional fields in the HTTP request headers, such as the user-agent field, would lead to a cache hit rate of close to zero with this policy. The Harvest solution was to completely avoid the header comparisons, which lead to the possibility that a Harvest cache could return incorrect data from when content negotiation was used.

[Gribble et al. 97] collected network traces of Web traffic from the modem pool at the University of California Berkeley for a 45 day period during October and November of 1996. During this time period, they collected a trace containing 24 million requests generated by 8,000 unique clients. The authors looked at whether requested Web content was cachable by examining the usage of the "Cache-control" and "Pragma" headers, and found that 7% of requests were specified as uncachable and less than 1% of responses were uncachable. They also found that only 2% of responses were CGI responses. Based on those numbers, they concluded that Internet services could benefit strongly from caching. The authors did not report on the usage of cookies in their workload, as will be discussed later.

The [Manley et al. 97] study collected server logs from ten different Web sites. Most of the logs were collected in the late 1996 and early 1997 time-frame, though some go back as early as 1994. These ten server sites were quite diverse in terms of their purpose and usage. There were three academic sites, five business sites, and two institutional sites. One of many workload aspects covered in this study was the amount of CGI usage at each site, because the authors were concerned about the server overhead of CGI processing. Most deployed proxies at the time scanned the URL for CGI or the query character and considered those pages uncachable.

The authors found that, in contrast with popular belief at the time, the usage of CGI was small and did not appear to be increasing. Only three of the ten sites found that more than 2% of all their requests were CGI requests, and only the professional organization site (at 34%) saw more than 10%. The most widely used CGI script among the sites surveyed was a page counter. In addition to the relatively small overall usage, the ratio of CGI traffic to regular traffic did not grow during the measurement intervals.

In 1998, we begin to see a different trend emerge. The [Caceres et al. 98] study collected network traces from the AT&T WorldNet ISP for a 12 day period in August of 1997. The authors pointed out that the conclusion of the [Gribble et al. 97] study was flawed because they did not consider cookies as a reason that certain resources might be uncachable. They found that 30% of the requests were uncachable due to cookies, and this resulted in a 20% decrease in the overall cache hit rates for their simulated proxy cache. The Squid proxy cache, derived from the earlier Harvest project, is currently the most popular freely available proxy cache. Squid does allow caching of requests that contain cookies, even though that policy conflicts with the original Netscape cookie specification. The draft cookie standard [Kristol et al. 97] makes it clear that the HTTP "Cache-control" headers must be used to control caching behavior, and therefore requests that contain cookies can be cached unless a "Cache-control" header states otherwise. This Squid caching policy directly conflicts with the authors conclusion that cookies are a significant factor that prevents caching of Web resources. [Feldmann et al. 99] performed a followup to the [Caceres et al. 98] study, and showed that including the requests with cookies, the overall rates of uncachability were 43% for the ISP trace and 37% for a trace of the AT&T Labs research community from 1997.

In [Wills et al. 99b], the authors used synthetic Web requests to investigate the cachability of resources from popular sites. They used site popularity information from 100hot.com to construct five sets of sites to probe. The pages queried from each site were the entry pages and the set of pages directly linked from the entry page. For each page, they downloaded all the embedded documents. They looked at the percentages of uncachable documents due to "Cache-control" and "Pragma" headers, and found that four of the five sets had a rate of uncachability less than 5%. The second commercial set had 17% of its documents uncachable. They also showed the percentage of documents that were uncachable because they were pre-expired, but these results were not listed as an overall rate. They were only listed broken down by content type, and the percentages appeared to be very small (less than 2%).

Since this study generated requests from outside of a Web browser, the authors could not report the percentage of requests that contained a cookie because none of their requests contained a cookie. They did report the percentage of responses that contained a "Set-cookie" header, and showed that it ranged from 10% to 17% for all but the educational set. One problem with interpreting this rate is that they fetched a high proportion of entry pages, which is the one place where cookies would

most likely be assigned for the entire site. The authors also examined whether or not resources changed based on cookies in order to decide whether or not requests that contained cookies ought to be declared cachable. They limited their test set to those pages from the first commercial set that returned a "Set-cookie" header. They accessed the pages twice and recorded the two different assigned cookies. They later re-accessed the site using the two different cookies to see if the content changed based on the cookie value. In 94% of the cases, the resource returned was identical, and they determined by manual examination that when the resource changed it was caused by banner advertisement rotation. They concluded that requests with cookies could be cached, and in most cases the cached content could be reused. One problem with this methodology is that cookies are often used to create personalized content at Web sites. Customizing the appearance of a page might require specific user actions, yet their methodology would not generate this customization behavior. Therefore, the reported 94% figure may be a significantly inflated estimate of how often different cookie values return the same content.

In [Wills et al. 99a], the authors performed a similar study but they constructed the set of pages to query from a proxy log rather than from the 100hot.com list. This page set was constructed from highly accessed documents in the NLANR proxy logs of December 1998 and January 1999. The advantage of this approach was that they examined those pages within a site that real users were accessing, while the disadvantage was that it made the results more dependent on the specific users of that proxy. The overall conclusions for this paper were very similar to the previous paper. The overall rate of uncachability from "Cache-control" and "Pragma" headers ranged from 4% to 12%. The percentage of pre-expired pages was in the 3% range, and the "Set-cookie" percentages ranged from 9% to 31%.

In this dissertation, we investigate the issue of uncachable documents based on a one week trace from the University of Washington (UW) population during May of 1999. Our trace contains 83 million requests from 23,000 clients. As with the [Caceres et al. 98, Feldmann et al. 99] studies, we see overall rates of uncachability that are much higher than previous studies. One key difference between our study and the [Caceres et al. 98, Feldmann et al. 99] studies is that we base our cachability decision on the Squid proxy cache implementation, which means that we do not consider requests containing a cookie to be uncachable. If we had, our rates for uncachability would be even higher. We enumerate ten possible protocol reasons for Web documents to be uncachable. When

combining all ten reasons, we find that 40% of all requests in our one week trace are uncachable. We also analyze one week of proxy logs from the Microsoft campus during the same May 1999 time period, and we find an even higher overall rate of uncachability at 49%. We do not know what the Microsoft proxy cache policy was for dealing with cookies.

Discussion

The conclusion one draws about the cachability of Web documents appears to be very much dependent upon the time at which the workloads being studied are collected. We have seen a significant shift: early studies up to the 1996 time frame showed a very small percentage of uncachable documents; whereas more recent studies including our own showed a much higher percentage of uncachable documents. Further study will be needed to see if the rates level out or go down in the future. Because HTTP 1.1 includes specific support for proxy caching, many of the issues of ambiguity in these workload studies, such as how to handle cookies, will be resolved once HTTP 1.0 usage fades away. A number of research proposals have been made to alleviate the problem of high rates of access to uncachable documents. These proposals are designed to allow limited caching of dynamic documents[Douglis et al. 97c, Cao et al. 98, Smith et al. 99], but as of this writing none of these techniques have been substantially deployed.

2.2.2 The Rate of Change to Web Documents

In this section, we look at issues related to how quickly Web documents change. Given a set of Web documents that are potentially cachable, the rate of change to those documents will be a limiting factor on the potential cache hit rates. In other words, a fast rate of change leads to a limited time period during which document reuse will cause a cache hit. Much of the early work looking at Web document rate of change investigated the design of cache consistency techniques for the Web. Recently there has been more focus on the detailed characteristics of how frequently documents change, and to what extent they change.

Cache Consistency

The Web does not support a strong cache consistency model where users always see the most recent version of a document. Instead, the Web uses explicit document expirations and time-to-live (TTL) heuristics. As part of the Harvest project, [Worrell 94] investigated using a hierarchical invalidation model for cache consistency. In order to decide whether or not invalidation-based consistency was needed, the author studied Web workloads to look at the predictability of object lifetimes. The main finding of this work was that object lifetimes were not easy to predict, implying that making TTL heuristics accurate would be very difficult.

The author generated synthetic Web requests in an adaptive manner to look at object lifetimes. The URLs in the test set were generated from users' history files, Web indices, and Harvest caches. The test set contained approximately 5000 URLs for which the object lifetime could be estimated, and was collected in the fall of 1994. The generated requests were adaptive: when the "Last-modified" headers detected a document change, the probe interval was shortened to better estimate the rate, and when no change was detected the probe interval was lengthened. The results of the study showed a large range of object lifetimes. Even for a single object, the time between changes varied considerably. Over 42% of the objects exhibited some variance in document lifetimes, and documents that changed quickly had smaller variances while documents that changed slowly had larger variances. The mean lifetime was 44 days for all objects, 75 days for HTML objects, and 107 days for images. The authors concluded that predicting object lifetimes is a very difficult task, and this increases the chance that TTL heuristics will be inaccurate. The consequence of inaccurate heuristics is that caches may return stale data to users unless lifetimes are chosen very conservatively which would generate extra cache validation traffic. In conclusion, the authors showed that an invalidation-based consistency scheme would be appropriate for the Web because at the break-even point in terms of bandwidth consumption, the fixed TTL mechanism returns stale documents 20% of the time.

[Gwertzman et al. 96] performed another study of cache consistency approaches for the Web, extending the simulator built by [Worrell 94]. This study used three server logs collected from different servers at Harvard University over a one month period during 1995, and they generated last-modification times from the file system. In their traces, they observed that the most popular

files change less often than the unpopular files. Their study also used data collected from a Boston University server over a 186 day period to look at object lifetimes. This data showed mean object lifetimes of 50 days for HTML documents and 85 to 100 days for the different image types.

Using simulation, they compared a fixed TTL algorithm such as Harvest's with invalidation protocols and with an adaptive polling algorithm. They conclude that the adaptive polling algorithm performs the best. Their polling algorithm is based on the Alex FTP cache [Cate 92]. The polling algorithm uses an update threshold to decide when to poll the server. The update threshold is calculated as a fixed percentage of the document's age. For example, if the threshold is 10%, and a given document has an age of 10 days, then the cache will poll the server to revalidate the document after 1 day. This algorithm forms the basis for the heuristic described in the HTTP 1.1 specification, except that the heuristic only determines the TTL value and does not suggest that proxies should poll the server when the freshness lifetime expires.

[Cao et al. 97] compared the adaptive TTL algorithm with two variants of strong consistency: polling every time and invalidation. They used five server traces from commercial sites, government organizations, and universities to evaluate the algorithms. The traces did not contain "Last-modified" headers; instead, they generated synthetic modification patterns that followed the distributions observed in [Gwertzman et al. 96]. They concluded that invalidation can be implemented with network overhead comparable to the adaptive TTL algorithm, and they claimed that the behavior with respect to failures was acceptable. We suspect that the current failure characteristics of the Internet will prevent invalidation systems from being seriously considered for quite some time.

[Duska et al. 97] used a set of seven proxy traces to investigate how often proxy caches need to validate expired objects in their caches, and the percentage of time those revalidations do not detect a change. Their trace set included a large 1996 proxy trace from DEC, four university traces, the NLANR second-level proxy cache traces, and a national trace from Korea. They found that the percentage of revalidations of unchanged objects ranged from 2% to 7%, and this represents the inefficiency of the coherence protocol.

[Krishnamurthy et al. 97] described using piggyback cache validation to improve cache consistency for the Web. When a proxy cache needs to contact a server to fetch a page, the cache can piggyback a set of validation requests onto the page request. Their proposal suggested that the piggybacked validation requests should be generated for those pages in the cache where the TTL has

expired and the page originates from the server being contacted. The benefit is that this eliminates the latency penalty for separate conditional GET requests. The reduction in message traffic therefore allows the heuristic TTL values to be set more conservatively, which in turn reduces the chance that stale pages will be served by the cache. This study used a 1996 proxy log trace from DEC and a 1996 packet trace from AT&T Research to compare their proposal with the adaptive TTL algorithm. They found that piggyback validation reduced the number of messages to the server by 17%, and reduced their cost metric (which combines the effects of latency and bandwidth) by 6%.

[Padmanabhan et al. 00] also investigated the effectives of adaptive TTL estimation, using server logs collected from an active news Web site, the MSNBC Web site. They collected a series of traces from 1998 and the fall of 1999. They looked at the correlation between their prediction based on N previous samples of the modification interval and the actual modification interval. They also looked at the accuracy of the prediction in terms of the error percentage. The results showed that with enough history (at least 15 samples), the coefficient of correlation was good, but the accuracy was not. However, they did find that 90% of their predictions were within 400% of the real modification interval, so they concluded that rough estimation is possible using adaptive TTL.

Change Characteristics

In this section, we take a closer look at the specifics of how frequently documents change and to what extent they change. Understanding the change characteristics of cachable documents is important because of the direct impact on cache hit rates. Understanding the rate of change for uncachable documents may help in the design of systems for caching dynamic documents.

[Douglis et al. 97a, Douglis et al. 97b] performed the most extensive study to date on the rate of change characteristics of Web documents. As with the [Krishnamurthy et al. 97] study, they used traces from AT&T and DEC collected in late 1996. They investigated the relationship between the rate of change to documents and other document characteristics such as content type, access rate, and size. The authors found that contrary to previous server log studies [Gwertzman et al. 96], the more popular documents in their traces were also the documents that changed more frequently. They found that the content type of a document did effect the frequency of changes, and that HTML documents changed more frequently than images. They found that the size of a document did not

affect the rate of change. They also investigated another issue related to document change, namely how often they discovered identical content with different names. This effect could be caused by document name changes, by site mirroring, or by encoding session-ids inside the URL. In the AT&T trace, they found that 18% of the responses had identical content to at least one instance of a different URL.

Using the same data, [Mogul et al. 97] investigated the benefits of delta encoding and data compression for the Web. When a conditional GET request determines that a cached document has in fact changed, delta encoding allows the server to just send the changes to that document rather than the entire document. Their delta encoding experiments used the vdelta algorithm [Hunt et al. 96], and their compression experiments used both vdelta and gzip [Deutsch 96]. One key finding was that when the headers implied that the document had changed, frequently the content had not changed. For the DEC and AT&T workloads, 21% and 32% respectively of the delta-eligible references were completely unchanged. They also found that when documents did change, the extent of the changes was small and therefore delta-encoding was very effective at reducing the number of bytes transferred. For the DEC proxy traces which were filtered to eliminate binary content, they found that the vdelta approach reduced the bandwidth requirements by 83% for the delta-eligible responses and by 31% for all status OK responses. For the AT&T traces which included all content types, they found that vdelta reduced the bandwidth by 85% for delta-eligible responses, but only 9% for all status OK responses. The final result was that simply compressing all document transfers cut the bandwidth requirements by 39% (DEC) and 20% (AT&T). They also investigated the host overhead of calculating the deltas at the server and applying the deltas at the proxy. They drew the conclusion that delta-encoding was effective, but they did not consider the additional storage overhead required at the server to allow the delta calculations.

[Wills et al. 99b] and [Wills et al. 99a] generated synthetic Web requests to popular Web sites to examine many similar issues. They fetched pages in their study only once per day, which placed limitations on their ability to detect very fast rates of change. They found results similar to [Douglis et al. 97a] in that HTML documents changed more frequently than images. They also saw significant variation in HTML change rates based on the different test sets they looked at. For instance, for the commercial sets, 70% to 80% of the pages changed on each retrieval, whereas for the educational set more than 60% of the pages did not change at all. They found significant prob-

lems with use of the HTTP cache validation mechanisms such as the “Etag” and “Last-Modified” headers. For both of these validation mechanisms, there are cases when the validator informs the cache that the document has changed, yet the content comparison shows that the document has not changed. For the com2 test set, this happened 9% of the time with the “Last-modified” header, and 37% of the time with the “Etag” header. Another significant problem was missing validation headers. For the proxy generated cnt20 test set, 37% of the time the “Last-modified” header was unavailable and the document did not change. A missing “Last-modified” header prevents a a cache from using heuristic expiration, and because the documents did not change it is likely that these responses would have lead to cache hits had they included the“Last-modified” header. Finally, they looked at the relationship between container pages and embedded images. When the container page changed, most of their test sets showed that more than 50% of the embedded images remained in the new version of the container document.

[Warren et al. 99] used weekly site snapshots to look at the evolution of the content of a set of Web sites over time. Though much of their focus was on the link structure of HTML pages, they observed that the rate of change on a given Web site appeared to be dependent on the total number of documents served by that site, with smaller sites having fewer changes. They also observed that for individual pages, the higher density of outgoing links that a page had, the higher the rate of change.

In our own work [Wolman et al. 99b] not included in this dissertation, we looked at document rate of change based solely on the HTTP header information, since our UW trace did not calculate document checksums which would allow verification of the changes. Our main contribution with respect to rate of change was to show that uncachable documents changed at a significantly higher rate than cachable documents. We also confirmed previous results that showed popular documents changed faster than unpopular documents. The hit rates estimates from our analytic model of the behavior of large-scale caching systems were quite sensitive to the change characteristics of unpopular documents, so we think more extensive information is needed in this area.

[Padmanabhan et al. 00] used server logs from a busy commercial Web site, the MSNBC site, to look at document rate of change from the server perspective. Most of the rate of change results were based on one week during October of 1999. One limitation arose from the fact that the site was actually a cluster of 40 machines, and the updates to documents were made using the Microsoft Content Replication System (CRS). Therefore, the information about document changes was gath-

ered from CRS logs, rather than directly from the filesystem. There were two key limitations of this technique: one cannot distinguish between modification and creation events, and one cannot record the filesystem last modification times before a change, which means that one can only determine modification intervals for files that change at least twice during the trace period. This also limited the maximum change interval to the trace period.

The results showed that during the week there were 29,000 modification events of which their heuristics estimated that 23,000 were file modifications and 6,000 were file creations. The modifications were made to 2,400 unique files during the week, or an average of 10 modifications per file. Further examination showed that most of these modifications were made to only a small set of files. Only 1% of the modifications were to images, and those were mostly changes to image maps. They found that 90% of the modification intervals fell between one hour and one day. They also found that with the vdelta algorithm, the extent of the changes to HTML files was very small. For 77% of the modifications, the delta size was less than 1% of the original file size, and for 96% of the modifications, the delta size was less than 10% of the original file size. Many of the very small changes were updates to either the date and time or to links.

Discussion

When looking at Web cache consistency mechanisms, we saw that estimating document lifetimes accurately is difficult. The adaptive TTL approach specified by HTTP 1.1 trades off extra validation traffic to reduce the chance of stale documents. We also saw that the current approach to Web cache consistency leads to a significant number of unnecessary document transfers due to inaccurate or missing header information. Of the research proposals for improving cache consistency, piggyback validation seems the most promising due to the relative ease of deployment.

In terms of Web document change characteristics, we saw a number of trends. Early studies showed that popular documents changed less frequently that unpopular documents, but all the studies since 1997 show just the opposite. The most popular Web documents appear to change at a faster rate than the unpopular documents. Different content types exhibit different change characteristics, and in particular HTML documents appear to change at a faster rate than images. When documents do change, the extent of the changes is often small, and delta encoding appears to be a very effective

technique to reduce bandwidth requirements for transferring those changes. However, deployment of delta-encoding may be controversial due to the extra burdens it places on Web servers.

2.2.3 Document Sharing

In this section, we look at the locality properties of Web workloads. In particular, we examine the amount of document reuse among clients, and the amount of temporal and geographical locality in the workloads. We then examine the popularity distribution of documents, and the implications of that distribution for modeling the effectiveness of large-scale Web caching.

Reuse and Locality

We first investigate the experimental results that look at the relationship between cache hit rates and client population size. We then summarize studies that look at geographical and temporal locality.

One of the earliest studies to look at the effectiveness of Web caching was [Glassman 94]. Their study was based on 12 weeks of proxy logs for the internal client population at DEC. They found that on a daily basis, the cache hit rates varied from 30% to 50% of all requests. For the full time period, approximately 1/3 of all requests were for pages that had never been requested, 1/3 of all requests were hits, and 1/3 of all requests were for previously requested documents that had expired.

[Gribble et al. 97] traced Web traffic from the modem pool at Berkeley for a month and a half in late 1996. Their simulations showed a peak cache hit rate of approximately 55% for the entire time period. Furthermore, the authors investigated the relationship between cache hit rates and the size of the client population. They found that the asymptotic hit rate grew logarithmically with the client population size, at least over the range of populations observable within their trace.

[Kroeger et al. 97] used the three week DEC trace from 1996 to analyze cache hit rates, and found that an upper bound on the cache hit rate for a proxy cache with unlimited storage would be 52% over the entire three week period.

[Duska et al. 97] used a set of seven proxy traces to investigate the relationship between cache hit rates, request rates, and population sizes. With their cache simulator, the peak hit rate for the DEC trace was 42%, which conflicts with the 52% reported in [Kroeger et al. 97]. One possible explanation for this discrepancy is how the cache simulator handled validation hits. The Squid

proxy cache implementation considers a cache hit to be any document whose body is served from the cache, regardless of whether or not a cache validation message was sent to the server before the cache served the document. It is clear that the [Duska et al. 97] paper did not consider these to be cache hits, but it is possible that the [Kroeger et al. 97] study did. The results of the [Duska et al. 97] study also showed that hit rates grew with both request rates and increases in client population size. For their workloads it appeared that growth in request-rate was a better predictor for cache hit rates than growth in client population size.

In this dissertation, we use workloads from UW and Microsoft to investigate the relationship between client population size and ideal proxy cache hit rates. Ideal hit rates are defined as an upper bound that ignores document expirations and uncachable documents. We find that for both populations, the ideal hit rates grow logarithmically with the client population sizes. We use these results to evaluate the effectiveness of cooperative caching across large populations, and we find that combining the two large populations from our traces only leads to a small increase in ideal hit rates.

We now examine the work that has investigated geographical and organizational sharing in Web workloads. The first study to investigate these issues was [Gwertzman et al. 95]. Using server traces from Harvard and NCSA, they found that only a small percentage of documents at a given site were extremely popular. They plotted the geographical location of the client subnets that made frequent requests to their two servers. They calculated geographic location based on the zip code of the administrative entity responsible for the subnet. They performed a clustering analysis on this data and determined that there were 22 clusters of subnets spread across the United States for the NCSA server, while there were only 4 clusters, two on each coast, for the Harvard server. They concluded from this data that site mirroring would be inefficient and that their results motivate distance-sensitive caching.

[Duska et al. 97] also examined geographic locality, but in a very different manner. The seven different traces in their study each represented a distinct collection of users, so they analyzed the sharing across the different user populations. Unfortunately, their largest trace from DEC was excluded from this portion of the study because the URLs were anonymized. The authors found that only a very small portion (0.2%) of the objects were shared across all six traces, but those shared objects were extremely popular and accounted for 16% of all the requests. 18% of the objects were shared by at least two traces, and these objects accounted for 56% of all the requests. They

characterized objects as either narrowly shared or widely shared and looked at the percentage of accesses to each. The widely shared objects tended to be the only ones that were shared across traces. They concluded that sharing is bimodal because half of the shared accesses in a trace were made to narrowly shared objects and half to widely shared objects.

In this dissertation, we examine organization-based sharing, which is a form of geographic sharing. For every client access in our trace from UW, we record the client's University organization. This allows us to investigate whether members of the same organization are more likely to share objects than members of different organizations. We answer this question by comparing the amount of sharing within the University organizations to the amount of sharing in randomly constructed organizations of the same size. We find that the amount of sharing in the University organizations is more than in the random organizations, but that this effect is not a strong one because the percentage increases are quite small.

[Padmanabhan et al. 00] performed a similar experiment on their server traces, but with the key difference that their organizations were formed by using the DNS level-two domain name (e.g. cnn from the name "www.cnn.com"). Since their trace was collected at a Web server, their organizations covered a much larger overall population, but the opportunity for local sharing was less since they only analyzed accesses to a single site. On normal days, the amount of sharing within a domain was significantly greater when compared with the random assignment. This effect appears to be stronger in their traces than in our study. However, on the day of an unusual event of global interest, they found that the amount of sharing within domains matched the amount of sharing with the random assignment, implying that the organizational locality effect was overwhelmed by these extremely "hot" events.

There has also been some study of temporal locality in Web workloads. [Almeida et al. 96] used the concept of stack distance to measure the amount of temporal locality in their workloads. The stack distance measures the number of intervening references in between two references to the same object. Their trace had a significant amount of temporal locality, when comparing the stack distances measured in their trace with the stack distances measured when they randomly reordered their trace. Furthermore, they showed that a log-normal distribution provides a reasonable model for the distribution of stack distances.

[Padmanabhan et al. 00] looked at temporal locality by examining the stability of document

popularity over time. They examined the percentage overlap of the most popular documents during different days of their traces. They found that the overlap remained quite high (in the 60% range), not only between adjacent days, but also between days that were almost one week apart. They concluded that a week-old trace is almost as good as a trace from yesterday for predicting which documents will be popular today. However, they did find a significant drop-off in the overlap when using traces that were three months apart or a year apart to make that same prediction.

Document Popularity and Zipf's Law

The document popularity distribution is a function that ranks the access frequency of documents.

George Zipf, a professor of linguistics at Harvard, published a book in 1949 that showed a large number of examples from social and economic data where the relationship between rank and frequency followed a particular distribution [Zipf 49]. In particular, he observed that the frequency of some event, as a function of the rank i of that event, is proportional to $1/i^{\alpha}$, where α is a constant close to one. This became known as Zipf's law, and probability distributions that follow this formula are often referred to as Zipf-like even when α is not equal to one [Breslau et al. 99]. The most famous example of Zipf's law comes from examining the frequency of words in the English language. Zipf showed the distribution by analyzing word frequencies from the 260,000 words of James Joyce's book *Ulysses*. Zipf assumed that the frequent occurrence of this distribution reflected some universal property of the human mind, and came up with an explanation he called the principle of least effort. At the time, there was a great deal of controversy over this explanation, but today it is commonly accepted that for both English texts and random texts, the cause of Zipf's law is purely statistical [Li 92].

[Glassman 94] was the first to observe the Zipf distribution for Web traffic patterns. They analyzed the 12 weeks of proxy logs from the DEC client population. On a log-log scale, they plotted the number of accesses versus the page rank for all the requests in their logs, and compared the results with the original Zipf distribution where $\alpha = 1$. They found that their distribution was similar, though not identical to the Zipf distribution. They then described how the Zipf distribution can predict an upper bound for cache hit rates if you have an estimate for the total number of documents on the Web. They also showed the reverse: if you observe a given proxy cache hit rate, you can use

that to create a rough estimate for the total number of documents on the Web.

[Cunha et al. 95] instrumented a version of the Mosaic Web browser to collect traces from a set of 40 machines in the Boston University Computer Science department. The traces were collected from late 1994 to mid 1995, and they observed 575,000 requests to 47,000 unique URLs. The authors found that their trace data followed a Zipf distribution, and they calculated the α parameter to be 0.986. Note that this trace included documents that were handled by the browser cache, and this affects the value of α when compared to traces from proxy caches which do not contain the accesses filtered by the browser caches. [Almeida et al. 96] examined 2 weeks of server logs in October of 1995 from the Boston University Computer Science department, and found that these requests followed a Zipf-like distribution with an α parameter of 0.85.

[Breslau et al. 99] examined a wide set of traces in an attempt to conclusively answer whether or not Web requests followed a Zipf-like distribution. The traces analyzed include one week of the widely studied 1996 DEC trace, 18 days of the 1996 Berkeley modem pool trace, part of a three month trace in 1997 from the University of Pisa, one day of traces from the NLANR second-level squid proxy caches in late 1997, one week of traces from a second-level proxy cache at Questnet in Australia in early 1998, and a ten day trace in mid 1998 of the FuNet proxies serving academic communities in Finland. All six traces were found to have a Zipf-like distribution, with the α parameters ranging from 0.64 to 0.83. The authors developed a model for estimating cache hit rates based on independent requests that follow a Zipf-like distribution. With this model, they showed that hit rates grow logarithmically as a function of the number of requests, which was consistent with the experimental observations. In other words, the results from the previous section which showed that cache hit rates grow logarithmically with respect to client population size are a direct consequence of the Zipf-like popularity distribution. They also used their model to evaluate four different cache replacement policies. They found that a least-frequently-used replacement policy is best for byte hit rates, and a document size conscious policy that favors smaller documents is best for overall hit rates.

In this dissertation, we study two traces that are an order of magnitude larger than those studied in [Breslau et al. 99], and we find that the popularity distributions are both Zipf-like, with an α parameter of 0.8.

[Padmanabhan et al. 00] examined the popularity distribution in server traces from the busy

MSNBC Web site. They also found that their server logs followed a Zipf-like distribution. When calculated on a daily basis, α ranged from 1.397 up to 1.816 for all of the days traced. They noted that the highest observed value of α was on December 17th, 1998. On this day there was an unusual event of global interest, namely a United States air-strike on Iraq. Since a higher value of α corresponds to a greater percentage of requests to the most popular documents, this result is not surprising in retrospect. The values of α in their server traces were significantly higher than those values observed in recent proxy traces. They provided some simple mathematical analysis to show that it is not surprising that proxy logs show lower values of α than server logs. They showed that if we take s popular servers, each of which has the same α value and the same number of documents n, and we combine the access logs for those servers to create a proxy log, then $\alpha_{proxy} = \alpha_{server} * (log(n)/log(s*n))$. Because the ratio $log(n)/log(s*n)$ is less than 1, the proxy will always have a lower α value than the individual servers.

[Huberman et al. 98] created a model for how users surf a given Web site. Their model is based on the idea that users make a sequence of decisions while accessing pages at a site. The decision to proceed to the next page is made as long as the value of the current page exceeds a certain threshold. This model yields a probability distribution for the number of pages visited by a user at a given site. This model is then combined with a spreading activation algorithm [Shrager et al. 87] to predict the number of hits at individual pages. The authors used this technique on randomly constructed Web sites, using a variety of initial conditions, to show that the resulting probability distribution of the page accesses over the total set of pages followed a Zipf distribution. Therefore, their value-threshold model provided a direct explanation for the occurrence of the Zipf-like distributions in Web proxy and server logs.

Discussion

It is clear from experimental measurements of cache hit rates and from analytic modeling results that the Zipf-like popularity distribution is a limiting factor for the overall effectiveness of Web caching and content distribution. The Zipf-like behavior in proxy and server logs appears to be the most consistent pattern throughout all of the caching related Web workload characteristics we looked at. We also saw a number of studies that found noticeable effects in terms of both geographical and

temporal locality, but these effects do not appear to dominate the large scale caching behavior of Web workloads.

2.2.4 Cooperative Caching

There has been extensive work on cooperative Web caching as a technique to reduce access latency and bandwidth consumption. Cooperative Web caching proposals include hierarchical schemes such as Harvest and Squid [Chankhuntod et al. 96, Squid 01], hash-based schemes [Karger et al. 99, Valloppillil et al. 98], directory-based schemes [Fan et al. 98, Tewari et al. 99], and multicast-based schemes [Michel et al. 98, Touch 98]. Although each of these research efforts included a performance evaluation of the protocols proposed and a discussion of algorithm scalability, only [Krishnan et al. 98] presented empirical evaluations of cooperation for small populations, and none presented empirical or analytical evaluations of the effectiveness of their schemes for the large client populations found in a wide-area setting.

For infinite-sized caches, it has been shown empirically and analytically that the hit ratio for a Web proxy grows logarithmically with the client population of the proxy and the number of requests seen by the proxy [Breslau et al. 99, Cao et al. 97, Duska et al. 97, Gribble et al. 97].

Using client traces, [Krishnan et al. 98] studied the utility of cooperation among three Bell Labs proxies with a small user population. They concluded that cooperative Web caching can be useful, but that a cache manager was necessary to dynamically determine when to cooperate because of the extra server load imposed by cooperation.

This dissertation expands on these previous research efforts. We use trace-based analysis to quantify the potential advantages and drawbacks of inter-proxy cooperation for small- and medium-size organizations.

In our own work not included in this dissertation, we used analytic modeling to examine cooperative-caching performance in wide-area environments. We extended the analytic model from [Breslau et al. 99] to include document rate of change parameters and to look at the behavior of caches in steady-state, rather than the behavior from a finite request sequence. We used the model to evaluate the benefits of large scale cooperative caching in terms of hit rate, latency, and storage costs. The results of our analytic modeling confirmed our trace-based results. We found that most

of the benefits of cooperative caching can be obtained at relatively small population sizes (e.g. a medium-size city).

[Gadde et al. 00] used the Zipf-based caching model from [Wolman et al. 99b] to investigate the effectiveness of content distribution networks such as Akamai. They found that although interior caches may yield good local hit ratios, they contributed little to the effectiveness of the caching system as a whole, as long as the populations served by the leaf caches were reasonably large.

[Dykes et al. 02] used analytic modeling to investigate the benefits of cooperative proxy caching on user response time, in contrast with previous studies that focused primarily on hit rates. They found that cooperation offered at best a small improvement in user response times, but that the primary benefit of cooperation was to reduce the variation in response times and eliminate very long delays.

Finally, much of the work examining document reuse and locality described in Section 2.2.3 is relevant not only to traditional Web proxy caching, but also to understanding the effectiveness of cooperative Web proxy caching.

2.2.5 Streaming Media Workloads

While Web client workloads have been studied extensively [Almeida et al. 96, Duska et al. 97, Gribble et al. 97, Feldmann et al. 99, Wills et al. 99b], relatively little research has been done on multimedia traffic analysis. Acharya et al. [Acharya et al. 98] analyzed video files stored on Web servers to characterize non-streaming multimedia content on the Internet. Their study showed that these files had a median size of 1.1 MB and most of them contained short videos (< 45 seconds). However, these results were based upon static analysis of the stored content, and thus ignored the issue of how often that content is accessed by Web clients.

Mena et al. [Mena et al. 00] analyzed streaming audio traffic using traces collected from a set of servers at Broadcast.com, a major Internet audio site. The focus of their study was on network-level workload characteristics, such as packet size distributions and other packet flow characteristics. From their analyses, they derived heuristics for identifying and simulating audio flows from Internet servers. Their study showed that most of the streaming-media traffic (60–80%) was transmitted over UDP, and most clients received audio streams at low bit-rates (16–20 Kb/s). The most striking

difference between results obtained from their trace and our analysis is that most of their streaming sessions were long-lived; 75% of the sessions analyzed lasted longer than one hour. We attribute this difference to the fact that they studied server-based audio traces from a single site, while we study a client-based trace of both audio and video streams to a large number of Internet servers.

Van der Merwe et al. [Van Der Merwe et al. 00] extended the functionality of tcpdump (a popular packet monitoring utility) to include support for monitoring multimedia traffic. Although the primary focus of their work was building the multimedia monitoring utility, they also reported preliminary results from traces collected from WorldNet, AT&T's commercial IP network. As in [Mena et al. 00], their study of over 3,000 RTSP flows also focused on network-level characteristics, such as packet length distributions and packet arrival times. In addition, they characterized the popularity of object accesses in the trace and similarly found that they matched a Zipf-like distribution. In contrast to our university client trace, the WorldNet workload peaks later in the evening and has relatively larger weekend workloads. We attribute this difference to the time-of-day usage patterns of the two different client populations; WorldNet users are not active until after work, while the university users are active during their work day.

There has been significant commercial activity recently (by companies such as FastForward Networks, Inktomi, and Akamai) on building caching and multicast infrastructure for the delivery of both on-demand and live multimedia content. However, little has been published about these efforts.

This dissertation builds upon this previous work in a number of significant ways. First, we study a trace with an order of magnitude more sessions. Second, we focus on application-level characteristics of streaming-media workloads – such as session duration and sizes, server and object popularity, sharing patterns, and temporal locality – and compare and contrast those characteristics with those of non-streaming Web workloads. Finally, we explore the potential benefits of performance optimizations such as proxy caching and multicast delivery on streaming-media workloads.

2.2.6 Web Workload Collection

Solutions to learning about Web performance generally fall into three classes: active measurement techniques that inject measurement traffic into the network; logs of behavior generated by Web

clients, servers, and proxies; and passive network monitoring. We limit our discussion of related research here to passive network monitoring, which is the approach we took to workload collection.

We begin with a summary of low-level general-purpose network monitoring tools. These are systems whose main focus is delivering packets from the wire to an application or to stable storage. Next, we look at higher-level general purpose network monitoring tools. Here, the focus is building general purpose infrastructure that makes it easier to write network monitoring applications to solve a specific task. Finally, we examine application-specific network monitoring systems, particularly those focused on Web-related protocols such as HTTP and RTSP.

Low-level Monitoring Tools

A packet filter provides operating system kernel support for delivering network packets to user-level processes. A packet filter typically provides a language that client applications can use to specify which packets should be discarded and which should be delivered. There have been many different packet filter designs developed over the years [Mogul et al. 87, McCanne et al. 93, Yuhara et al. 94, Engler et al. 96].

One of the most widely used tools for passive monitoring of network traffic is tcpdump [Tcpdump 01]. Tcpdump consists of two components: a packet capture library that provides buffer management and a common interface to a variety of different kernel packet filters; and a set of protocol specific presentation routines that decode the headers of a given network protocol and then print that packet header information in a human readable format.

OC3MON [Apisdorf et al. 97] is a combined hardware and software solution for monitoring OC3 Internet backbone links. OC3MON classifies packets into *flows*, where a flow is defined as a set packets that travel from one endpoint to another, and a fixed timeout period is used to determine when a flow terminates. OC3MON generates low-level aggregate flow-based statistics. For instance, OC3MON can be used to record the total number of flows on a given port, and a variety of details about those flows such as packet counts, byte counts, and flow duration.

IPMON [Fraleigh et al. 01] is a distributed passive monitoring infrastructure for measuring IP backbone traffic. IPMON supports collecting time synchronized traces from multiple backbone links. The measurement systems are built using a PC with a SONET interface and a RAID storage

system, each on a separate PCI bus. The traces that are collected include the first 44 bytes of each IP packet, along with a timestamp generated by a GPS reference clock. IPMON can handle backbone links up to OC-48 speed (2.5 Gbps). A one hour trace from an OC-48 link generates 176 GB of trace data. All trace analysis is handled offline with a 16 node cluster of Linux machines.

Higher-level Monitoring Tools

Network Flight Recorder [Ranum et al. 97] is a toolkit for building network traffic analysis applications. NFR consists of a packet capture layer (using the tcpdump packet capture library), a decision engine, and a set of backend modules. The packet capture layer delivers packets to the decision engine, which checks the packet against all the installed filters. A filter is a predicate subroutine used to determine whether the packet is delivered to a backend processing module. The N-code language used to build NFR filters is much higher-level than the filter languages supported by kernel packet filters. For example, NFR supports TCP connection reassembly within the decision engine, so one can easily write a filter that is based on the content of application-level data within a TCP connection. Backend modules are used for statistical analysis and logging of the data.

The high-level goals of the Windmill system [Malan et al. 98] are very similar to those of NFR - they constructed a general purpose toolkit that makes it easier to build specific network analysis applications. However, the implementation focus of Windmill was very different from NFR – they focused much more on performance and much less on statistical and graphical processing of the traffic data. Windmill developed a custom packet filter that is novel in that it can efficiently support delivering a single packet to multiple filters using dynamic code generation. At user-level, Windmill provides a set of abstract protocol modules for commonly used protocols such as IP, UDP, and TCP that applications use to ease the task of creating analysis applications. For each of the provided modules, the programming model is as follows: the protocol module implements a traditional end-host network protocol implementation along with a set of additional interfaces to extract state information from the protocol stack. The monitoring system can then use multiple instances of these modules to emulate multiple end-hosts.

Application-specific Monitoring Tools

Httpdump [Wooster et al. 96] was one of the earliest tools developed specifically for monitoring HTTP. Httpdump is a passive network monitoring tool used to create logs of traffic to a set of Web servers. Httpdump is implemented as a classification layer on top of tcpdump – it identifies HTTP headers and then creates a log of requests and responses. However, it suffered from severe performance problems – there was high packet loss monitoring a group of just 21 client machines that generated 13,000 requests in a 10 day period.

BLT [Feldmann 00] is a passive network monitoring system developed specifically for studying the behavior of HTTP traffic. BLT records both TCP and HTTP events, and this provides significantly more detail than the information that is typically found in Web proxy or server logs. BLT supports continuous online monitoring through the use of a DLT tape loader. The BLT system has been used in a number of locations, running for weeks at a time with less than 0.3% packet loss when monitoring links speeds up to 100 Mb/sec.

In terms of high-level functionality, the passive network monitoring system developed in this dissertation closely resembles the BLT system. However, there are some key differences: our system supports monitoring traffic from multiple network interfaces on the same machine; our system takes a significantly more aggressive approach to protecting the privacy of the users being monitored, which also leads to certain functionality limitations that BLT does not suffer from; our system supports monitoring for the RTSP streaming media protocol in addition to HTTP; and finally, the implementation strategies of the two systems are very different.

As mentioned above, the mmdump [Van Der Merwe et al. 00] tool is a modified version of tcpdump that supports monitoring of streaming media control protocols including RTSP and H.323. The modifications include new modules for parsing each of the control protocols, as well as a mechanism that allows the parsers to dynamically change the installed packet filter based on port numbers parsed from the control protocols. This is needed because the actual streaming media content is usually delivered out of band over a UDP port that is assigned using the control protocol. The uses of mmdump have already been described above.

A number of other Web performance studies (e.g. [Gribble et al. 97, Smith et al. 01]), used network tracing to collect Web workloads, but the task of collecting the workloads was not the

focus of these studies.

2.3 Summary

We began this chapter with an overview of the background information necessary to understand the basic interactions between Web clients, Web proxy caches, and Web servers. We reviewed the architecture of two application-level transport protocols used to deliver Web and streaming-media content, namely HTTP and RTSP. We provided an overview of Web caching, and summarized the explicit mechanisms used to support caching in the HTTP protocol.

We surveyed previous research in the areas of Web workload characterization for caching, and Web workload collection and analysis. This dissertation contributes to further understanding of Web workload characteristics in the areas of document cachability, sharing, cooperative caching, and streaming-media delivery. Furthermore, we develop a passive network monitoring system for Web workload collection, and a set of tools for workload analysis.

We investigate document cachability, and observe a much higher overall rate of requests to uncachable documents than previous studies observed in the 1996 timeframe [Gribble et al. 97, Manley et al. 97]. A key difference in methodology from the [Caceres et al. 98, Feldmann et al. 99] studies is that we base our cachability decision on the Squid proxy cache implementation, which leads us to a more conservative estimate of the percentage of uncachable documents. We provide a detailed breakdown of the possible reasons why requests and responses are uncachable, and we find that 40% of all requests in our UW trace are uncachable. During the same time period, we find that 49% of all requests in our Microsoft campus trace are uncachable.

Earlier studies [Gwertzman et al. 95, Duska et al. 97] investigated geographical locality in Web workloads generated by servers and proxy caches. In this dissertation, we examine organization-based sharing, which is a form of geographic sharing. We develop a novel anonymization strategy that allows us to associate an organization identifier with every client in our trace. This allows us to investigate whether members of the same organization are more likely to share objects than members of different organizations. We answer this question by comparing the amount of sharing within the University organizations to the amount of sharing in randomly constructed organizations of the same size. We find that the amount of sharing in the University organizations is more than in

the random organizations, but that this effect is not a strong one because the percentage increases are quite small.

Along with [Duska et al. 97, Gribble et al. 97, Breslau et al. 99], we observe that proxy cache hit rates grow logarithmically as a function of client population size. Our traces were collected later than the other studies, and were an order of magnitude larger than the traces from other studies. We find that for both the UW and Microsoft populations, hit rates grow logarithmically with the client population size. We find that the popularity distributions for the UW and Microsoft traces are both Zipf-like, with an α parameter of 0.8. Although others have observed the logarithmic growth relationship between proxy cache hit rates and client population size, one contribution of our work is applying that relationship to understand the performance of cooperative caching at different scales.

This dissertation builds upon previous work studying the characteristics of streaming-media content [Acharya et al. 98, Mena et al. 00, Van Der Merwe et al. 00] in a number of significant ways. First, we study a trace with an order of magnitude more sessions. Second, we focus on application-level characteristics of streaming-media workloads – such as session duration and sizes, server and object popularity, sharing patterns, and temporal locality – and compare and contrast those characteristics with those of non-streaming Web workloads. Finally, we explore the potential benefits of performance optimizations such as proxy caching and multicast delivery on streaming-media workloads.

A few tools have been developed to study the network behavior of specific application-level protocols [Wooster et al. 96, Feldmann 00, Van Der Merwe et al. 00]. The passive network monitoring system developed in this dissertation uses a similar approach to the BLT system [Feldmann 00]. We avoid building our system on top of a general purpose high-level monitor, such as NFR [Ranum et al. 97] or Windmill [Malan et al. 98], primary for scalability reasons: the solutions used by these systems would not scale to handle our workload without requiring significantly more in the way of hardware resources. One key difference between our system and BLT is that our system supports monitoring traffic from multiple network interfaces on the same machine. We also take a significantly more conservative approach to protecting the privacy of the users being monitored, which in turn dictates a very different implementation strategy.

Chapter 3

Organization-Based Analysis of Web-Object Sharing and Caching

3.1 Introduction

In this chapter, we investigate the sharing and caching characteristics of Web document access from the perspective of *organizations*. The need to understand Web behavior and performance has led to a large number of studies, aimed in particular at classifying Web document *characteristics* [Cunha et al. 95, Douglis et al. 97b, Duska et al. 97, Gribble et al. 97, Mah 97]. In contrast, the principal goal of this study is to evaluate document *sharing behavior* on the Web, both *within* organizations and *between* organizations. By document sharing, we mean access to the same Web documents by different clients. Sharing behavior has obvious implications for performance, particularly with respect to the effectiveness of proxy caching (e.g., [Chankhuntod et al. 96, Fan et al. 98, Gwertzman et al. 95, Kurcewicz et al. 98, Zhang et al. 97]). Proxy caches are often located at organizational boundaries and improve performance only if many documents are shared by many clients. Therefore, an understanding of sharing gives us added insight into potential performance-enhancing mechanisms and alternative caching structures.

An analysis of document sharing within an organization is straightforward and can help predict the benefits of an organizational proxy cache [Duska et al. 97]. Studying sharing across multiple organizations is much more difficult, however. Tracing of the entire Web is obviously not achievable, but even simultaneous traces of multiple organizations do not currently exist. In addition, the requirement of most organizations for anonymization of URLs and IP addresses, along with the different dates of data capture, makes correlation of separate traces challenging, if not impossible.

In this study, we use the University of Washington (UW) as a basis for modeling intra- and inter-organizational Web-object sharing. The UW is the largest university in the northwest part of the U.S., with a population of over 50,000 people, including 35,000 students, 10,000 full-time staff, and 5,000 faculty. The university has a large communications infrastructure, consisting of

thousands of computers connected via both high-speed networks and modems. As of May 1999, this community generates a peak workload of about 17,400 Web requests per minute that originate within the UW directed to external Web servers (we'll refer to these as UW-external requests).

As with other universities, UW is organized into many colleges, departments, and programs, each with its own disparate administrative, academic, or research focus. For example, the UW includes museums of art and natural history, medical and dental schools, libraries, administrative organizations, and of course academic departments, such as music, Scandinavian languages, and computer science. What do such diverse organizations have in common with respect to their Web access requests? To answer this question, we have traced all UW-external Web requests; we anonymize the data in such a way as to identify requests (and associated responses) with the 170 or so independent organizations from which they were issued. This permits us to study organization-specific document access and sharing behavior. We have collected a number of traces during the period from the Summer of 1998 through the present. In general, all of our traces show the same basic patterns. The results in this chapter are based on a representative one-week trace taken in mid-May 1999.

In this chapter, we expand on previous research efforts studying the cachability of Web documents and document locality. Although our primary focus is on sharing and cachability, we can also compare our current HTTP traffic characteristics to earlier studies, showing how the Web workload has changed. Our work is based on recent data from a large diverse population. More important, we preserve enough information about client location that we can analyze requests with respect to inter-organization and intra-organization document sharing.

The remainder of this chapter is organized as follows. In Section 3.2 we describe our trace-capture methodology. Section 3.3 contains a high-level description of the workload we traced. Section 3.4 focuses on organization-based statistics and also provides inter- and intra-organization sharing analysis. In Section 3.5 we discuss cachability of documents, and reasons why documents are not cachable. Finally, Section 3.6 summarizes our study and its results.

3.2 Measurement Methodology

In this section, we provide a brief overview of the traffic collection and analysis infrastructure developed to perform this study. A significantly more detailed description of the tracing system is

provided in Chapter 6. We use *passive network monitoring* to collect our traces of Web traffic between the University of Washington and the rest of the Internet. We designed and implemented the tracing software that produced the data in this study. Our tracing software performs TCP connection reconstruction on all TCP segments that flow through the UW Internet border. Our tracing software captures all HTTP headers including those from HTTP connections using non-standard ports and those from HTTP persistent connections.

We use an anonymization approach that protects privacy, but preserves some address locality information. For internal addresses, we classify the IP address based on its "organization" membership. An organization is a set of university IP addresses that forms an administrative entity; an organization may include multiple subnets. For instance, all machines in the Computer Science Department are in a single organization, machines in the Department of Dentistry are in another, and machines connected to the campus Museum of Natural History are in yet another. We constructed the mapping from subnets to organization identifiers based on information obtained from the campus network administrators. Once the organization identifiers are assigned, both the IP address and the organization identifier are anonymized. Furthermore, some bits of information in the IP address are destroyed before anonymization to make the anonymization more secure. If the hash function or key is compromised, no transaction can be associated with a given client address with absolute certainty.

For external addresses, we anonymize each octet of the server IP address separately. For our purposes, two servers are near each other if they share most or all of the Internet path between them and the university. We consider two servers to be on the same subnet when the first three octets of their IP addresses match. Given the use of classless routing in the Internet, this scheme will not provide 100% accuracy, but for large organizations we expect that this assumption will be overly conservative rather than overly aggressive.

Although our tracing software records all HTTP requests and responses flowing both in and out of UW, we filter the data presented in this chapter to look at only HTTP requests generated by clients inside UW, and the corresponding HTTP responses generated by servers outside of UW. All of our results are based on the entire trace collected from Friday May 7th through Friday May 14th, 1999, except for the organization-based sharing results in Section 3.4, which are from a single day (Tuesday) of our trace (the limitation is due to the memory requirements of the sharing analysis).

3.3 High-Level Data Characteristics

Table 3.1 shows the basic data characteristics. As the table shows, our trace software saw the transfer of 677 gigabytes of data in response packets, requested from about 23,000 client addresses, and returned from 244,000 servers. It is interesting that, compared to the commonly-used 1996 DEC trace (analyzed, e.g., in [Duska et al. 97]), which had a similar client population, we saw four times as many requests in one week as DEC saw in 3 weeks. These requests and corresponding response and close events follow the typical diurnal cycle, with a minimum of 460 requests per minute (at 5 AM) and a peak of 17,400 requests per minute (at 3 PM).

Figures 3.1a and 3.1b present a histogram of the top content types by object count and bytes transmitted, respectively. By count, the top four are image/gif, text/html, No Content Type, and image/jpeg, with all the rest of the content types at significantly lower numbers. The No Content Type traffic, which accounts for 18% of the responses, consists primarily of short control messages. The largest percentage of bytes transferred is accounted for by text/html with 25%, though the sum of the image/gif (19%) and image/jpeg (21%) types accounts for 40% of the bytes transferred. The remaining content types account for decreasing numbers of bytes with a heavy-tailed distribution.

Another type that accounts for significant traffic, which is not readily apparent from the table, is multimedia content (audio and video). The sum of all 59 different audio and video content types that we observed during the May trace adds up to 14% of all bytes transferred via HTTP. In addition, there is a significant amount of streaming multimedia content that is delivered through an out-of-band channel between the audio/video player and the server.

In a preliminary attempt to view some of this out-of-band multimedia traffic, we extended our tracing software to analyze connections made by the Real Networks audio/video player, examining port 7070 traffic. Newer versions of the Real Networks player use the RTSP protocol, which we do not handle. The Real Networks player sets up a TCP control connection on port 7070, and then transfers the data on UDP port 7070. Our trace software only collects TCP segments, so we analyze the control connection to determine how much data is being transferred. When the control connection is shut down, a “statistics” packet is transmitted that contains the average bandwidth delivered (in bits per second) as measured by the client for the completed connection. We take that bit-rate and multiply it by the connection duration time to estimate the size of the content transferred.

Table 3.1: Overall statistics for the one-week UW HTTP trace.

HTTP Transactions (Requests)	82.8 million
Objects	18.4 million
Clients	22,984
Servers	244,211
Total Bytes	677 GB
Average requests/minute	8,200
Peak requests/minute	17,400

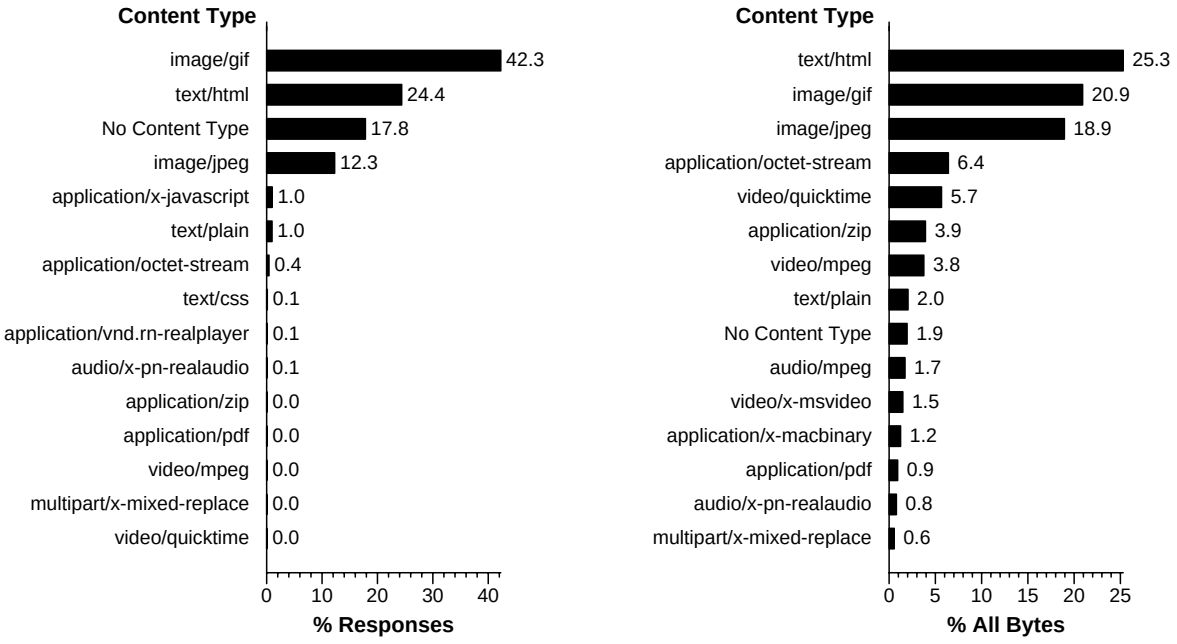


Figure 3.1: Histogram of the top 15 content types by count and size.

Some of the control connections do not transmit the statistics packet, in which case we cannot make an estimate.

During the week of the May trace, we observed 55000 connections, of which approximately 40% had statistics packets. For those 40%, we calculated that 28 GB of Real-Audio and Real-Video data were transferred (which would scale to 10% of the amount of HTTP data transferred if the other 60% of connections have similar characteristics). Furthermore, the Real-Audio and Real-Video objects themselves are quite large, with an average size of just under a megabyte. When we sum up all the different kinds of multimedia content, we see that from 18% to 24% of Web-related traffic coming in to the University is multimedia content, and this is a lower bound since we know that we're missing RTSP traffic at the very least. We believe that the large quantity of audio and video is signaling a new trend; e.g., in the data collected for studies reported in [Douglis et al. 97b] and [Gribble et al. 97], audio traffic does not appear. In Chapter 5, we use a more reliable methodology to perform a detailed study of the characteristics of RTSP traffic usage at UW during a one-week period in April of 2000.

We also examined the distribution of object sizes for HTTP objects transmitted. We observe here once again the usual heavy-tailed phenomenon that has been observed for object size distributions in all previous studies. In our trace, we found a mean object size of 8.3 KB, with a median of just over 1 KB. These numbers are fairly consistent with those measured in earlier traces, e.g., [Mah 97, Gribble et al. 97].

We were also curious about the HTTP protocol versions currently in use. The majority of requests in our trace (53%) are made using HTTP 1.0, but the majority of responses use HTTP 1.1 (69%). In terms of bytes transferred, the majority of bytes (75%) are returned from HTTP 1.1 servers.

These statistics simply serve to provide some background about the general nature of the trace data, in order to set the context for the analysis in the next two sections.

3.4 Analysis of Document Sharing

This section presents and analyzes our trace data, focusing on document sharing. As previously stated, our intention is to use the university organizations as a simple model of independent orga-

nizations connecting to the Internet. Our goal is to answer several key questions with respect to Web-document sharing, for example:

1. How much object sharing occurs between different organizations?

2. What types of objects are shared?

3. How are objects shared in time?

4. Is membership in an organization a predictor of sharing behavior?

5. Are members of organizations more similar to each other than to members of different organizations, or do all clients behave more-or-less identically in their request behavior?

Figure 3.2 plots total Web requests per 5 minute period over the one-week trace period. The shading of the graph divides the curve into three areas: the darkest portion shows the fraction of requests that are initial (first) requests to objects, while the medium and light grey portions show the subset that are duplicate (repeated) requests to documents. A request is considered a duplicate if it is to a document previously requested in the trace by any client. The lightest grey region shows those requests that are both duplicate and cachable, as we will discuss later.

Overall, the data shows that about 75% of requests are to objects previously requested in the trace. This matches fairly closely the results of [Duska et al. 97] on several large organizational traces. The percentage of shared requests rises very slowly over time, as one might expect. From our one-week trace, we cannot yet see the peak; however, this analysis does not consider document timeouts or replacements, therefore the 75% is optimistic if used as a basis for prediction of cache behavior. Furthermore, we cannot tell from this figure how many of the requests to a shared object were duplicate requests from the same client; overall, we found that about 60% of the requests to shared documents were first requests by a given client to those documents; 40% were repeated requests by the same client.

A key component of our data is the encoding of the organization number, which allows us to identify each client as belonging to one of the 170 active university organizations. These organizations include academic and administrative departments and programs, dormitories, and the

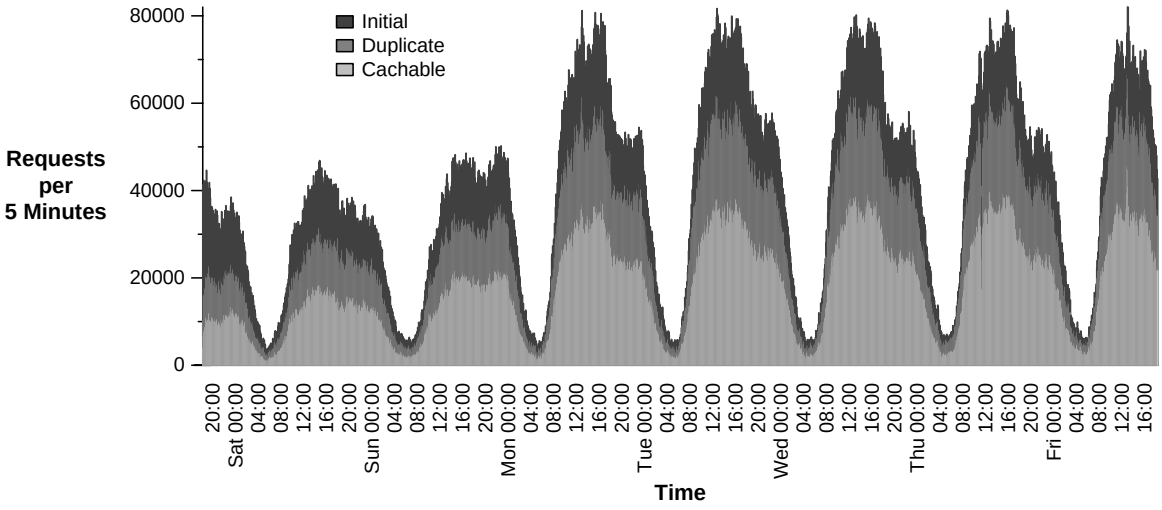


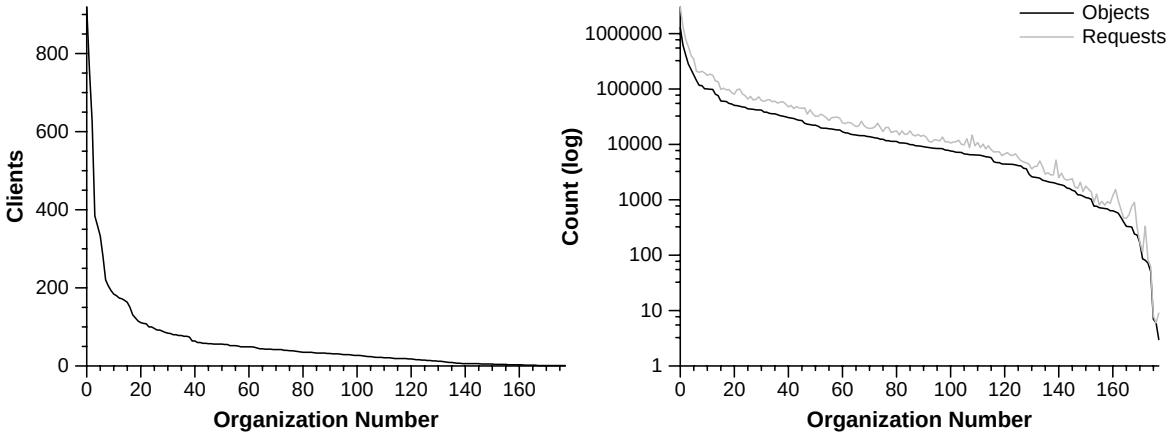
Figure 3.2: Requests broken down into initial, duplicate, and cachable duplicate requests over time.

university-wide modem pool. Figures 3.3a and 3.3b show the organization size, the request rate, and number of objects accessed by each organization. There are several very large organizations, with most somewhat smaller. The largest organization has 919 "anonymized" clients, the second largest organization is the modem pool with 759 clients, and the third largest organization has 626 clients.[1] The top 20 organizations all have more than 100 clients, as shown by the labels in Figure 3.4. Because of the way that client IP addresses are anonymized, we cannot uniquely identify an individual client, i.e., each anonymized client address could correspond to up to 4 separate clients. For this trace the ratio of "real" clients to "anonymized" clients measured by the low levels of our trace software is 1.67; therefore, our 13,701 anonymized clients represent 22,984 true clients.

Using the organization data, we can analyze the amount of object sharing that occurs both within and between organizations.

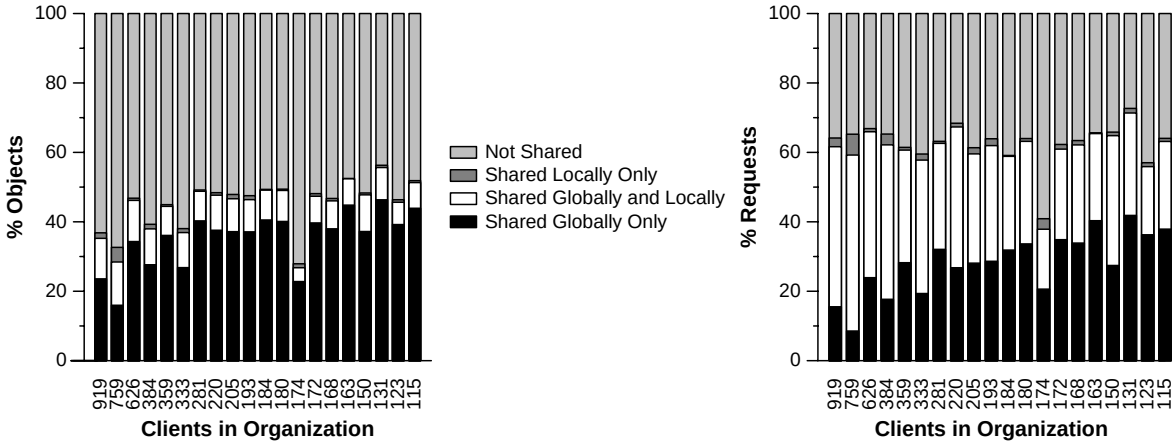
Figure 3.5a shows intra-organization (*local*) sharing from the perspective of both objects and requests. The black line shows the percentage of all objects accessed by each organization that are *locally-shared* objects, i.e., accessed by more than one organization member. The light grey line shows the percentage of all organization *requests* that are to these locally-shared objects. The

[1]The modem pool is somewhat special, because multiple clients can login through a single IP address in the pool.



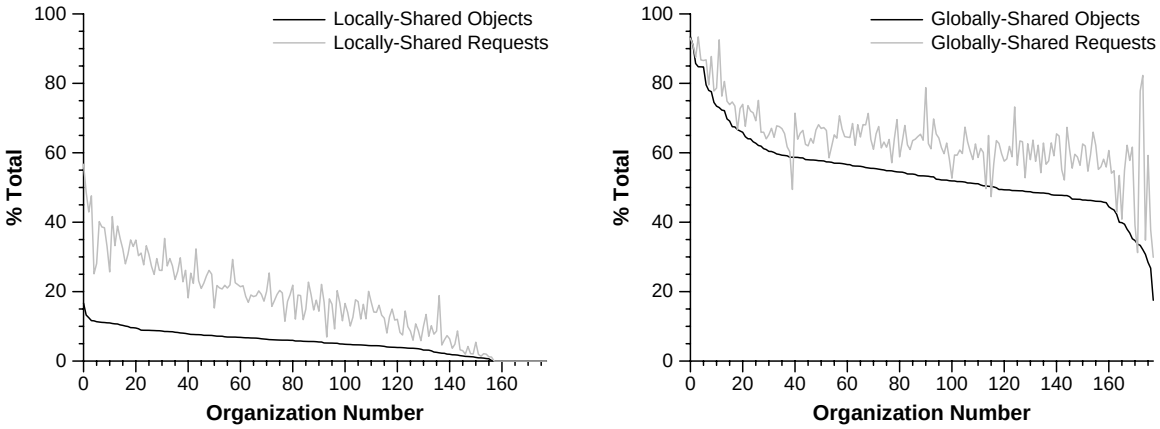
(a) Clients Per Organization (b) Objects and Requests Per Organization

Figure 3.3: Distribution of clients, objects, and requests in organizations. The object and request graph is sorted by the number of objects in an organization. Note that the y-axis of (b) uses a log scale.



(a) Object Sharing (b) Request Sharing

Figure 3.4: Breakdown of objects (a) and requests (b) into the different categories of sharing, for the 20 largest organizations. The labels on the x-axis show the number of clients in each organization.



(a) Intra-Organization Sharing (Local)

(b) Inter-Organization Sharing (Global)

Figure 3.5: The left graph shows the fraction of objects and requests accessed by the organization that are shared by more than one client within the organization. The right graph shows the fraction of objects and requests accessed by the organization that are shared with at least one other organization.

organizations are ordered by decreasing locally-shared object percentage. From our data on intra-organization sharing we can make the following observations:

- Only a small percentage (4.8% on average) of the *objects* accessed within an organization are shared by multiple members of the organization (the smooth black line).
- However, a much larger percentage of *requests* (16.4% on average) are to locally-shared objects (the light grey line).
- The average number of requests per locally-shared object is 4.0 – higher than the minimal 2 requests required for an object to be considered shared.
- Each locally-shared object is requested by two clients on average within each organization.

Figure 3.5b shows the inter-organization (*global*) sharing activity. Here the black line shows the percentage of all objects accessed by each organization that were also accessed by at least one *other* organization; we call such objects *globally-shared* objects. Similarly, the light grey line shows

the percentage of all *requests* by an organization to globally-shared objects. The organizations are ordered by decreasing globally-shared object percentage. From our data on inter-organization sharing we can make the following observations:

- There is more sharing with other organizations than within the organization; the fraction of globally-shared objects and requests in Figure 3.5b is much higher than the locally-shared objects and requests in Figure 3.5a. This is not surprising, because the combined client population of all of the organizations is significantly larger than any one organization alone. As a result, there is a much greater opportunity for the clients in one organization to share with clients from any of the other organizations.

- For 65% of the organizations, more than half of the objects referenced are globally-shared objects (the smooth black line).

- For 94% of the organizations, more than half of the requests are to globally-shared objects, and for 10% of the organizations 75% of the requests are to globally-shared objects (the light grey line).

- However, globally-shared objects are not requested frequently by each organization. On average, each organization makes 1.5 requests to a globally-shared object.

- On average, a globally-shared object is accessed by only one client in each organization.

A key question raised by these figures is whether the objects shared within an organization are the *same* set of objects that are shared across organizations. Figure 3.4a shows, for the 20 largest organizations, a breakdown of organization-accessed objects into various sharing categories: locally-shared only, globally-shared only, both locally and globally-shared, and not shared. Figure 3.4b shows the same breakdown by requests rather than objects. The graphs are ordered in decreasing organization size, with the organization size shown on the x-axis.

From Figure 3.4b, we see that the fraction of requests to shared objects is fairly flat across these organization sizes. As we would expect, the fraction that are shared globally-only rises somewhat with decreased organization size, while the fraction that are locally-shared decreases with decreasing

organization size. That is, in general, the smaller the organization, the less organization-internal sharing, and the more global sharing. Looking at the white section of the bars in both figures, we see that the small percentage of objects that account for both local and global sharing are very hot, and account for a much greater fraction of the requests than the objects they represent. In contrast, the percentage of requests to objects shared locally-only is very small for these organizations.

To aid in the understanding of the degree of object sharing, Figure 3.6 plots the number of objects (on the y-axis) that were shared by exactly x organizations. Most objects are accessed by only one organization, as shown by the steepness of the curve at $x = 1$. We also found that there were more than 1000 objects accessed by 20 organizations and more than 100 objects accessed by 45 organizations.

A key question with respect to our sharing data is whether organization membership is significant. To answer this question, we randomly assigned clients to organizations, and compared the inter- and intra-organization sharing in the random assignments with the sharing seen in our trace analysis presented above. In this experiment, we created the random organizations to have the same number of clients as the actual organizations. Figure 3.7a plots the fraction of requests to locally-shared objects of the trace organizations and three randomly-assigned organizations. From the figure, we see that sharing is higher in the real organizations than in the randomly-assigned organizations. In other words, there is locality of references in organization membership. Figure 3.7b plots the fraction of requests to globally-shared objects for the trace and for the three random organizations. As expected, there is no significant difference in the amount of global sharing between the real trace and the randomized organization assignment.

The organization-oriented data show that membership within an organization does influence the amount of sharing. Members of an organization are more likely to request the same documents than a set of clients of the same size chosen at random. However, the extent of this influence is not large, as the vast majority of the requests made are to objects that are *globally* shared. In addition, objects that are shared both locally within an organization and globally with other organizations are more likely to be requested by an organization member. This suggests that the most requested objects are universally popular.

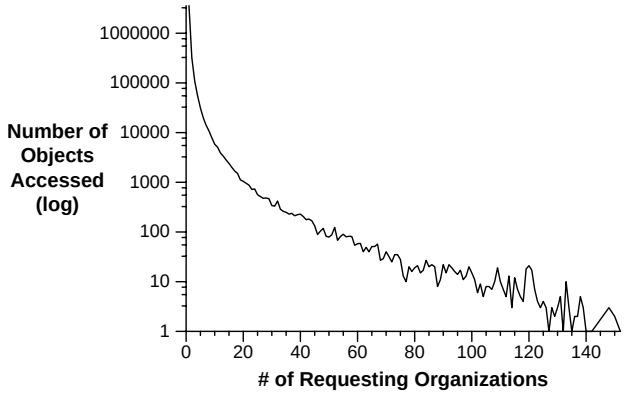
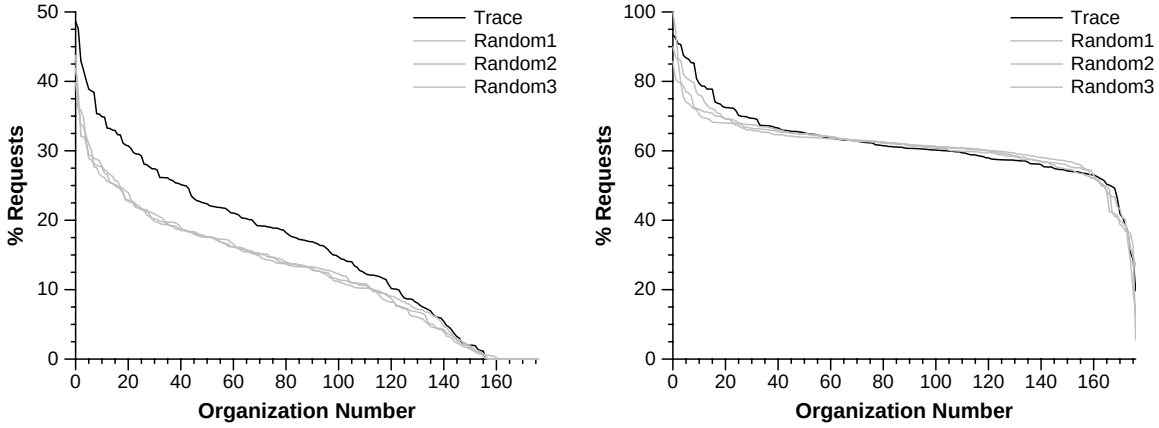


Figure 3.6: The number of objects accessed by a given number of organizations. Note that the y-axis uses a log scale.



(a) Intra-Organization Sharing (Local)

(b) Inter-Organization Sharing (Global)

Figure 3.7: Trace vs. Random Sharing. We show the fraction of requests generated by the organization that are (a) shared within this organization, and (b) shared with at least one other organization, in both cases compared with three random client-to-organization assignments.

3.4.1 Object and Server Popularity

Another aspect of sharing patterns that we examine is the popularity of those servers being accessed. We also investigate popularity using server proximity information (i.e., which servers are close to each other in the network).

Figure 3.8 shows the cumulative distribution functions of both server popularity and server subnet popularity, where popularity is measured by the request-count. We also measured popularity using byte-counts, and the byte-count curves for server popularity and server subnet popularity are effectively identical to the request-count curves shown in the graph. The data indicates that 50% of the objects accessed and bytes transferred come from roughly the top 850 servers (out of a total of 244,211 servers accessed). As described in Section 3.2, we consider a server subnet to be a set of servers that share the same first 24 bits of their IP addresses. Such groups of servers are typically mirrors of each other, or at least sit in a single server farm owned by a single company. We see that 50% of the objects come from about the top 200 server subnets; 18% come from the top 20 subnets. One factor that we cannot account for in the server subnet analysis is Web hosting companies. Web hosting companies provide a set of machines that sit on the same subnet yet act as Web servers for *different* companies or organizations. In fact, a single machine may be acting as a Web server for multiple organizations.

3.5 Document Cachability

This section examines cachability of documents, giving us insight into the potential effectiveness of proxy caches in our environment. Web proxy caches are a key performance component of the Web infrastructure; their objective is to improve performance through caching of documents requested more than once. Proxies typically live at the boundaries of an organization, caching documents for all clients within that organization.

In Figure 3.2 we saw a time-series graph of the percentage of duplicate requests (i.e., requests to a previously-accessed document) and the percentage of cachable duplicate requests in our trace. A request is considered cachable when it is made to a document that would be cached by a standard proxy cache, such as Squid [Squid 01]. We found that, in steady state, approximately 45% of the requests are duplicate and cachable, placing an upper bound on the hit rate. The wide difference

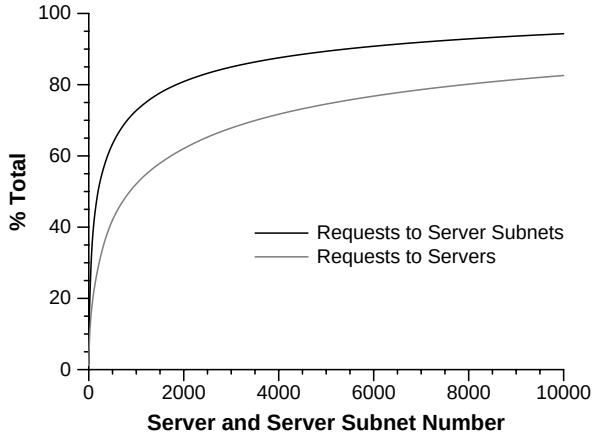


Figure 3.8: The cumulative distributions of server and server subnet popularity.

between the duplicate line and the cachable line indicates that only about half of the duplicate requests (which could benefit from caching) are to objects that are cachable.

Our cachability analysis is based on the implementation of the Squid proxy cache. We examined the policies implemented by both Squid version 1 and Squid version 2. There are several reasons why a Squid proxy may consider a document uncachable.

- *CGI* – The document was created by a CGI script or program and is not cached, because it is produced dynamically.

- *Cookie* – The response contains a set-cookie header. Squid version 1 does not allow these responses to be cached, but Squid version 2 does allow them to be cached. Note that both versions of Squid consider *requests* that contain a cookie header to be cachable.

- *Query* – The request is a query, i.e., the object name includes a question mark (“?”).

- *Pragma* – The request or response is explicitly marked uncachable with a “Pragma: no-cache” header.

- *Cache-Control* – The request or response is explicitly marked uncachable with the HTTP 1.1 Cache-Control header.

- *Method* – The request method is not “GET” or “HEAD”.

- *Response-Status* – The server response code does not allow the proxy to cache the response. For example, response code 302 (Moved Temporarily) cannot be cached when there is no explicit expiration date specified.

- *Push-Content* – The content type “multipart/x-mixed-replace” is used by some servers to specify dynamic content.

- *Auth* – Requests that specify an Authorization header.

- *Vary* – Responses that specify a Vary header.

Figure 3.9 shows a breakdown of all HTTP requests, detailing the percentage that are uncachable for each of the reasons listed above. As the figure shows in the bar labeled “Overall_Uncache”, 40% of the requests are uncachable for one or more of the itemized reasons. Queries and Response Status are the two major reasons for uncachability. Adding up the percentages for each reason sums to an amount greater than the overall uncachability rate, showing that many documents are uncachable for more than one reason. The figure also shows, for each itemized reason, the percentage of HTTP requests that are uncachable only due to that reason. Finally, the figure shows that 16% of Web requests are uncachable for two or more reasons. Figure 3.10 shows the most common content types for the uncachable documents.

Our intent in analyzing the cachability of documents is to show which requests a deployed proxy cache would be allowed to store if it were given the request stream from our trace. However, one should not infer from our analysis that all of the uncachable requests are truly dynamic content. Web content providers may choose to mark documents uncachable for other reasons, such as the desire to track the behavior of individual users. Figure 3.10 shows that more than 12% of all the uncachable documents have the image/gif content type, and we suspect that very few of these images are truly dynamic content.

Figure 3.11a shows, for each organization, the percentage of objects (black line) requested by the organization that are potentially cachable. The light grey line shows, for each organization,

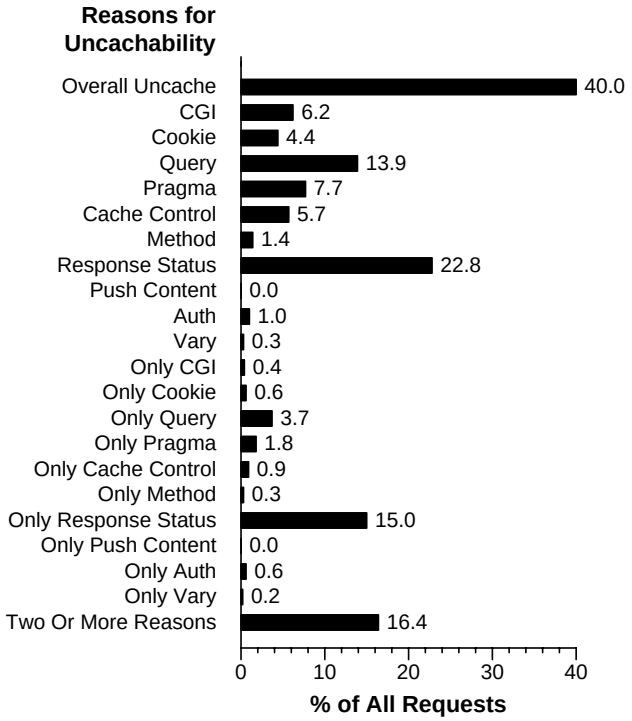


Figure 3.9: Reasons for uncachability of HTTP transactions.

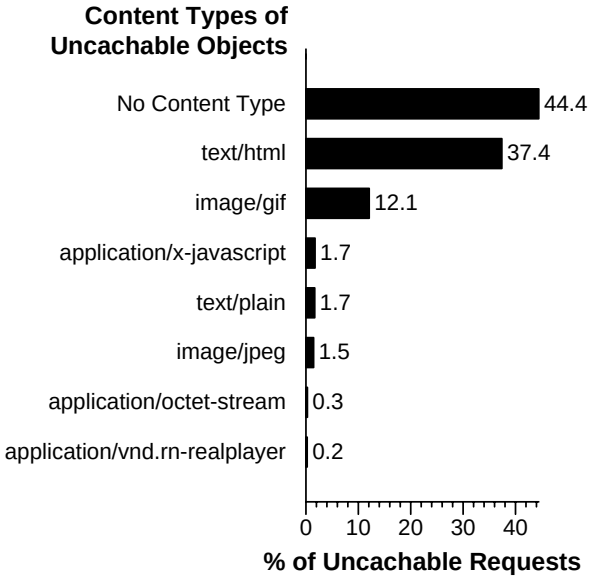
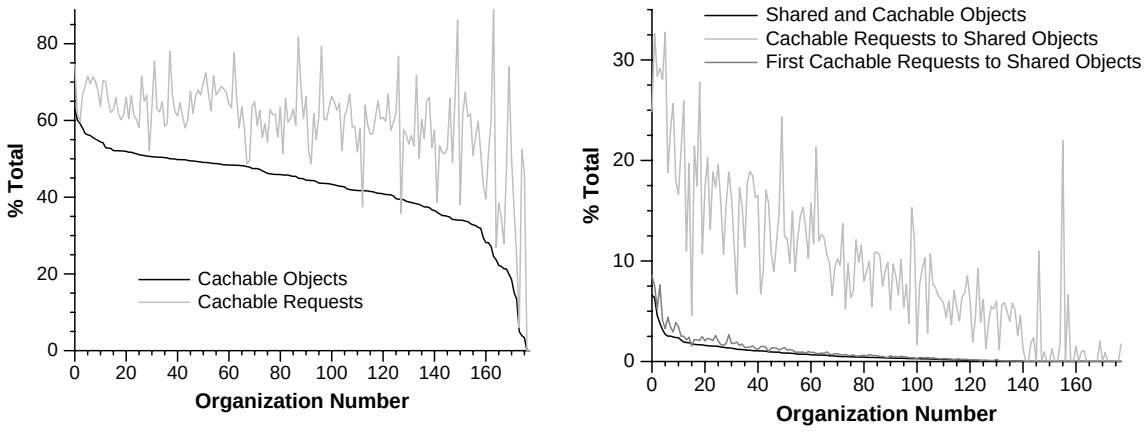


Figure 3.10: Breakdown by content-type of the uncachable HTTP transactions.



(a) Cachable Objects and Requests

(b) Cachable and Shared

Figure 3.11: The left graph shows the fraction of cachable objects and cachable requests accessed by each organization. The right graph shows the fraction of objects and requests that are both cachable and shared by more than one organization.

the percentage of cachable requests. The figure shows that the percentage of cachable objects is somewhat lower than the percentage of cachable requests. The percentage of cachable requests gives an upper bound on the hit rate each organization could see with an organization-local proxy cache.

Figure 3.11b shows, for each organization, the percentage of cachable shared objects (the black line), and the percentage of cachable shared requests in two categories. The medium grey line shows those first requests by an organization to globally shared objects. The light grey line shows the total number of requests by an organization to globally shared objects. The difference between these two lines represents the duplicate requests by an organization to globally shared objects. If each organization has its own cache, then the local cache can handle all duplicate requests whether or not there is a global cache. If there is a global cache in addition to the local caches, then the global cache will miss on the first request by any of the organizations, but will hit on all the first requests by other organizations that follow. One can conclude from this graph that there is significant sharing among organizations (as shown by the light grey line), but that a large fraction of that sharing is captured just with organizational caches (as shown by the difference between light and medium grey lines).

Therefore, a global cache in addition to the local caches will help, but not nearly to the degree indicated by the amount of sharing among organizations. Another interesting question is whether a single global cache would be better than using local caches. We explore this question in Chapter 4.

A last factor that can affect the performance of caching is object expiration time. We found overall that only 9.2% of requests had an expiration specified. Most of these requests are to objects that expire quickly; 47% are to objects that expire in less than 2 hours. Interestingly, of those that did have an expiration specified, 26% had a missing or invalid date and 29% had an expiration time that had already passed.

Finally, we have not presented detailed cache simulations here; our objective is simply to analyze cachability of documents in the most recent data. From our data, it appears that the trends with respect to cachability of documents are getting worse. For example, our measurement that 40% of all document accesses are uncachable is significantly higher than the 7% reported for client traces at Berkeley in 1997 [Gribble et al. 97]. Without widespread deployment of special mechanisms to deal with caching, such as caching systems that handle dynamic content [Cao et al. 98, Challenger et al. 99], the benefits of proxy caching are not likely to improve.

3.6 Conclusions

In this chapter, we have collected and analyzed a large recent trace taken in a university setting. Our study has focused on sharing of Web documents within and among a diverse set of organizations within a large university.

We can reach the following conclusions from our data:

- When clients are members of the same organization, there is a measurable increase in the amount of sharing when compared with clients that are members of different organizations. However, this increase is not large enough to have a significant impact on cache performance. The vast majority of the requests made (and the objects requested) are to objects that are shared among multiple organizations.

- Objects that are simultaneously shared locally by an organization and globally with other organizations are more likely to be requested by an organization member than objects that are

just shared locally or just shared globally. This suggests that the most-requested objects by an organization are globally and universally popular.

- The trace shows mostly minor differences relative to earlier traces in terms of many of the basic characteristics. However, we see two important differences compared to previous traces. The first is that the percentage of requests to uncachable documents is significantly higher. The second is that a significant amount of audio/video content appears in our trace.

When analyzing these conclusions, one must keep in mind that we do not know how similar our university organizations are to typical commercial organizations that connect to the Internet. We do explore this question to some extent in Chapter 4. We have only begun to analyze the data we have collected. Possible future work includes a more detailed statistical analysis of various aspects of the data already collected, as well as a study of the evolution of Web traffic characteristics over time.

Chapter 4

The Scale and Performance of Cooperative Web Proxy Caching

4.1 Introduction

Cooperative caching – the sharing and coordination of cache state among several communicating caches – has been shown to improve the performance of file and virtual-memory systems in a high-speed, local-area network environment [Anderson et al. 96, Feeley et al. 95]. For example, when a file-page miss occurs, the local file cache may transfer the page from the file cache on another node. Cooperative caching works in this environment because network transfer time is much smaller than the disk access time required to service a miss.

Internet proxy caching has become a commonplace approach for improving the performance of Web browsers. Typically, the proxy sits in front of an entire company or organization. By caching requests for a group of users, a proxy can quickly return documents previously accessed by other clients. Ultimately, though, the hit rate of the proxy is a function of the size of the population it manages – a size often dictated by political, organizational, or geographic considerations. An obvious question, then, is whether multiple proxies should cooperate with each other in order to increase total client population, improve hit ratios, and reduce document-access latency. Whether such cooperative proxy caching is a useful architecture for improving performance depends on a number of factors. These include the sharing patterns of documents across organizations, the ratio of inter-proxy communication time to server fetch time, and the scale at which cooperation is undertaken.

Several cooperative-caching protocols have been proposed [Chankhuntod et al. 96, Fan et al. 98, Michel et al. 98, Rabinovich et al. 98, Tewari et al. 99]; however, few studies have examined cooperative Web caching from a systemic viewpoint. As a result, we know neither the environments in which cooperative caching is useful (if any) nor its potential performance benefits. Answering such questions has been difficult in the past, because studying proxy cooperation requires *simultaneous*

traces from multiple proxies.

In this chapter, we explore the potentials and limits of cooperative proxy caching using trace-based analysis. We collect and analyze traces from two environments: the University of Washington and the Microsoft Corporation. As a key component of our university trace, we identify each client in terms of its membership in one of about 200 university departments or programs. This gives us the equivalent of a simultaneous trace of 200 diverse, independent organizations, permitting us to analyze document sharing among those organizations and to measure the potential benefits of cooperation among organization-based proxies. We examine latency and bandwidth benefits of proxy caching for these data, as well. We then use the Microsoft trace of employee traffic to the Internet to explore the potential of cooperation between larger organizations. To do this, we analyze traces from Microsoft and the university that we collected over the same time period and processed with the same anonymization function. This permits a direct computation of the degree of document sharing, and hence the benefit of sharing, between two proxies each handling tens of thousands of clients.

Our results show the benefits of cooperative caching among collections of small organizations. However, we show that cooperative caching is unlikely to have significant benefits for larger organizations or populations. That is, with current sharing patterns, there is little point in designing highly scalable cooperative-caching schemes; all reasonable schemes will have similar performance in the low-end population range where cooperative caching works. Thus, the crucial problem that must be solved to improve Web performance is how to increase document cachability.

The remainder of this chapter is organized as follows. The next section examines in greater detail the questions we aim to answer about the relationship between document sharing and cooperative caching. Sections 4.3 and 4.4 introduces the traces and simulation methods used in this study. Sections 4.5 through 4.8 present the hit-rate, latency, and bandwidth results of our trace-driven simulations. Section 4.9 summarizes and concludes.

4.2 Document Sharing and Cooperative Proxy Caching

This chapter poses and answers a number of questions about the potential of cooperative proxy caching. We focus on exploring the bounds of cooperative-caching performance. Key questions

include:

1. What is the best performance one could achieve with “perfect” cooperative caching?

2. For what range of client populations can cooperative caching work effectively?

3. Does the way in which clients are assigned to caches matter?

4. What cache hit rates are necessary to achieve worthwhile decreases in document access latency?

To answer these questions quantitatively, we have collected and analyzed Web access data from two environments: (1) the University of Washington (UW), consisting of about 50,000 students, faculty, and staff, and (2) the Microsoft Corporation (MS), consisting of about 40,000 employees. Most significantly, the two traces were collected simultaneously and anonymized in the same way, allowing direct comparison of trace records, including URLs and server addresses. We use the UW trace as a way to analyze document sharing by 200 small, independent organizations within a diverse university; we use the UW and Microsoft traces together as a way to analyze document sharing across two large organizations. In both environments, we examine the potential benefits of proxy cache coordination from the perspective of the clients and the network; in this study, we do not investigate the effects of cooperative caching on server load in general or under hot-spot conditions.

The following sections present results derived from these traces. In related work, we developed an analytic model of Web client behavior that goes beyond the limitations of our trace client populations to much larger population sizes [Wolman et al. 99b].

4.3 Trace collection and characteristics

While several client traces exist in the public domain [Cunha et al. 95, Douglis et al. 97b, Duska et al. 97, Gribble et al. 97, Mah 97], most are several years old, and the information they contain is inadequate for our analysis. We therefore designed and implemented custom trace software and installed it at the University of Washington’s Internet border. The details of our trace collection and analysis system are covered in Chapter 6.

Few proxy caches are deployed within our university, so at the border we are able to see almost all of the Web client traffic generated from inside the university. The traces are anonymized, but the anonymization preserves certain key aspects of the data that we require. In particular, we anonymize client IP addresses, but we first classify the client based on its organizational membership. This allows us to identify requests from different academic and administrative entities within the university, and classify each entity as a unique organization. We describe the use of this information in Section 4.7.

The trace used in this chapter was collected from May 7th through May 14th, 1999, and Table 4.1 presents the high-level details of this trace. As the table shows, we saw about 83 million requests by 23,000 clients to 244,000 servers over the seven-day period in the UW trace. Using this data, we can determine upper bounds on the performance of any cooperative caching algorithm. This tells us whether proxy cooperation is worthwhile even in the best case in our environment.

We also processed traces collected by the proxies handling all outgoing traffic from the Microsoft Corporation. These traces were collected on the same days that we collected our UW trace. Our software anonymizes both traces using the same functions, so that URLs and server IP addresses can be directly compared. Table 4.1 shows that in the period May 7th to May 14th, 1999, we saw about 108 million requests by 60,000 clients to 360,000 servers in the Microsoft trace.

4.4 Simulation methodology

The results presented in this chapter are based on Web cache simulations using our traces as input. This section discusses the methods used for the experiments we performed.

We make assumptions in our simulator that a real cache would not make, and therefore do not model reality exactly. However, our goal is to investigate behavior, not to reproduce hit rates exactly, and we believe that our assumptions do not change our conclusions about cache behavior. The caches we simulate are infinite-sized and do not model expirations. As a result, they are somewhat optimistic. Real caches will incur misses owing to capacity limitations that we do not model. However, capacity misses are rarely the bottleneck for Web caches. For example, only three percent of the requests to the Microsoft Web proxies from which we gathered our traces missed due to the finite capacity of the proxies (which have 9 GB of RAM and 180 GB of disk capacity). Real caches

Table 4.1: Overall statistics for the UW and Microsoft HTTP traces.

Parameter	**UW**	**Microsoft**
HTTP Requests	82.8 million	107.7 million
HTTP Objects	18.4 million	15.3 million
Total Request Bytes	677 GB	(N/A)
Average Requests/Min	8,200	11,900
Clients	22,984	60,233
Servers	244,211	360,586
Duration	7 days	6 days 6 hours

will also expire some objects that our simulations keep alive in the cache.

At the same time, our simulation experiments are conservative, because they include compulsory (cold start) misses. We minimize this effect by simulating traces over long periods of time. In a related study [Wolman et al. 99b], we exclude the effect of compulsory misses by using an analytic model to study the steady-state performance of Web caches.

We simulate two kinds of Web caches, a "practical" cache and an "ideal" cache. A practical cache closely models the cachability of documents according to the algorithms in the Squid V2 implementation [Squid 01]. Our cachability predicate accounts for HTTP 1.1 cache control headers, cookies, object names with suffixes naming dynamic objects, no-cache pragmas, uncachable methods and response codes, and headers with Authorization and Vary fields. We reported the detailed breakdown of document cachability in our traces in Chapter 3.

An ideal cache treats all documents as cachable. It is well known that some Web objects, such as images used in advertisements, are marked uncachable even though their contents do not change and could be cached. Future improvements to Web protocols and cache implementations can potentially be more aggressive and cache those objects that practical proxies cannot currently cache. Since we cannot anticipate all future improvements and implement them in our simulator, we instead use an ideal cache to report the upper bound that such improvements can hope to achieve on our workloads.

Many of the experiments examine cache performance as a function of client population. Because

each UW modem is reused by many people, using a modem IP address to represent a client would be inaccurate. As a result, we exclude modem traffic in our analyses and focus on LAN users.

In most of the graphs that follow, the clients for a given population size are randomly selected out of the pool of clients seen in our traces. Unless otherwise specified, each point shown in these graphs is the mean of four independent random trials, and error bars show the standard deviation across these trials.

4.5 The Impact of Population Size

In a cooperative-caching scheme, a proxy forwards a missing request to other proxies to determine if: (1) another proxy holds the requested document, and (2) that document can be returned faster than a request to the server. Whether such cooperation is worthwhile will depend on the number of proxies involved, their distances (inter-proxy communication latencies), their utilizations, the client populations served, and the complexity of the protocols used.

The result of cooperative proxy caching is simply to increase the effective client population. That is, at best, a collection of cooperating caches will achieve the hit rate of a single proxy acting over the *combined* population of all the proxies. In reality, the performance will be less than perfect, because proxies will not have perfect knowledge and will pay the overheads of inter-proxy communication latency. Examining a single, top-level proxy thus gives us an *upper bound* on cooperative-caching performance.

Figure 4.1 graphs hit rate vs. client population size for our traces. The dark black lines in this graph show the behavior of the university population. The dotted black line shows the "cachable hit rate," which corresponds to the hit rate from a practical cache that considers the cachability characteristics of the documents. Both the shapes of the curves and the cachable hit rates are roughly consistent with previous proxy cache studies of client traces: [Gribble et al. 97, Kroeger et al. 97] both report hit rates above 50% (but disregard cookies), [Duska et al. 97] reports hit rates of 40–45%, and [Caceres et al. 98] reports hit rates of 35% (but exaggerates the effects of cookies, which can often be cached in HTTP 1.1).

The solid black line in Figure 4.1 shows the "ideal hit rate" of an ideal cache for the UW trace – one that would be achievable *if* all shared documents were cachable. In the future, improvements

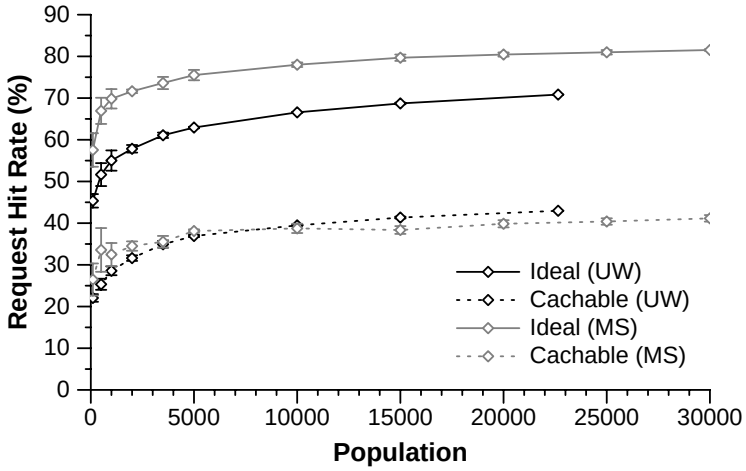


Figure 4.1: Proxy cache request hit rate as a function of client population.

to Web protocols may move the cachable line closer to the ideal line. On the other hand, future changes in document characteristics may move the cachable line in either direction.

The grey lines in Figure 4.1 show the behavior of the Microsoft population. It is interesting to note that the ideal-curve asymptote is higher by about 13% than that of the university environment, an indication that document sharing within Microsoft is much higher than within the university. This suggests, not surprisingly, that the Microsoft population is much more homogeneous in its Web-access behavior than the university population. However, we also see that the Microsoft cachable curve almost overlies the UW cachable curve. This is a direct reflection of the different distributions of requests to cachable documents in the traces; 60% of requests in the UW trace go to cachable documents, but only 51% of requests in the Microsoft trace do so. As a result, even though there is more *sharing* among Microsoft users, document cachability currently prevents a Microsoft proxy cache from achieving a hit rate that is any better than a UW proxy cache.

From this simple figure, we can draw several key conclusions about cooperative caching. The graphs have a sharp knee at about 2500 clients. The steep increase in hit rate below that knee implies that a large potential benefit (hit rate increase) could exist from cooperative caching for multiple proxies with small client populations. For example, given 10 proxies, each handling a population of a few hundred clients, cooperative caching has the potential to improve significantly the hit ratio seen by the clients of those proxies. This improvement occurs because the total population served

by each proxy increases from 200 to 2000 clients.

It is important to note that the total number of clients (below the knee) that can benefit from cooperative caching *could easily be handled by a single proxy cache* for our traces and user populations. Often this will not be possible, however, because decisions on proxy placement are based on political or geographical factors, such as company organization, location, and so on. While one organization may not trust another to proxy all of its requests, it may be willing to cooperate with other proxies for performance reasons.

Figure 4.1 also shows that hit rate increases very slowly with client population once past the knee of the curve. It is therefore not clear whether cooperative caching is beneficial in this region, for proxies whose populations are already above a few thousand clients. We will explore this question in more detail below.

4.6 Latency and Bandwidth

While many caching studies focus on hit rate, in the Web environment it is ultimately latency, not hit rate, that is crucial to clients. From the perspective of Internet service providers, hit rate translates into bandwidth savings over costly Internet links. These bandwidth savings can also reduce wide-area network congestion, potentially improving the performance of the Internet as a whole.

Figure 4.2 shows document latency in our university trace as a function of the number of clients using a proxy cache. The top three lines, from top to bottom, show mean last-byte latency: (1) without a proxy cache (i.e., as extracted from the trace), (2) with a proxy cache respecting cachability, and (3) with an ideal proxy cache. The bottom three lines show median last-byte latency under the same three conditions. Note that, unlike other curves, the error bars on the median lines correspond to the minimum and maximum median values among the trials at each population size. The three mean lines level out quickly, while the medians are essentially flat. This implies that caching will have little impact on mean and median latency beyond very small client populations. The mean trace curve is not constant, because each point represents requests from different client population samples, and mean latency will vary from one sample to another.

In Figure 4.2, the mean latency is much higher than the median owing to many high-latency documents. Can cooperative caching reduce the percentage of these high-latency documents? Fig-

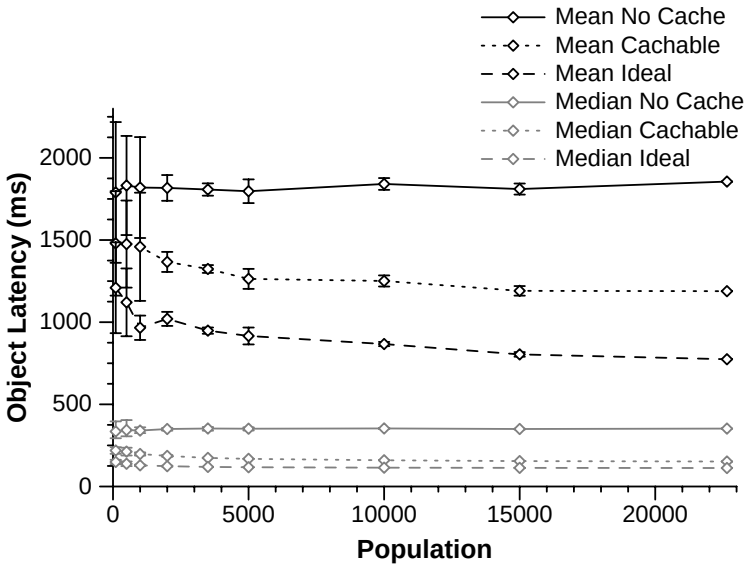


Figure 4.2: Mean and median request latency as a function of client population for the UW trace. The error bars on the median curves are the min and max medians across the trials.

ure 4.3 implies that this is not the case; it shows the percentage of documents with a last-byte latency below two seconds for the three caching policies described above. When graphed as a function of population, this percentage is effectively a horizontal line for all three policies with very little difference among them. The ideal line is the highest (90%), and no cache is the lowest (82%). The insensitivity to population size and closeness of these values demonstrate that neither cachability nor increasing population will significantly reduce the number of high-latency documents. Our trace analysis indicates that these documents are slow due to document size, network latency (e.g., from congestion, low-bandwidth links, long distances), or both. Also, as described below, shared documents tend to be smaller than non-shared ones, biasing misses towards larger documents that consequently take longer to download.

A final dimension is bandwidth. Figure 4.4 shows the byte hit rate as a function of client population for the university trace. Once again, we see a knee in the curve at around 2500 clients. Comparing these results to the hit rates given in Figure 4.1, we can conclude that shared objects are smaller on average than other objects. Figure 4.5 shows the average bandwidth consumed as a function of client population for the three caching situations. We see that while caching reduces

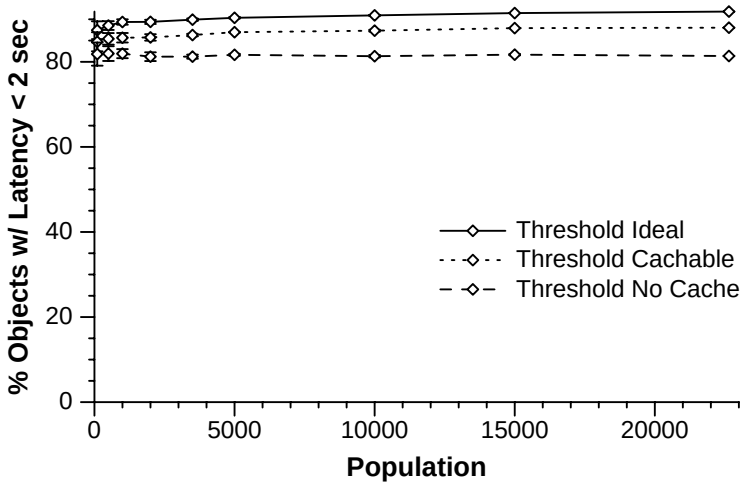


Figure 4.3: Fraction of requests completed in less than two seconds for the UW trace.

bandwidth consumption compared to no caching, there is no benefit to increased client population (i.e., there is no decrease in the slope of the bandwidth line).

4.7 Proxies and Organizations

We produced Figures 4.1 and 4.4 by computing the hit rate of random subsets of clients at each population size; therefore, the figures assume that all clients are essentially identical in their access patterns. A crucial question is whether clients in a single organization, sharing a single proxy, have more in common with each other than with clients in different organizations sharing other proxies. If there is high locality within organizations, then populations smaller than the knee in Figure 4.1 could achieve the maximum hit rate. What benefit would clients in *real* organizations see if their proxies were to cooperate with other *real* organizational proxies?

To answer this question is difficult, because it requires a simultaneous trace of a large number of proxies in the Internet. Such traces have not existed in the past. We have tried to answer the question in our university environment, using UW as a small-scale model of the broader networked community. The University of Washington consists of a large collection of diverse organizations – e.g., museums of art and natural history, schools of nursing and dentistry, and departments such as music, Scandinavian languages, and computer science. Think of each such organization as an

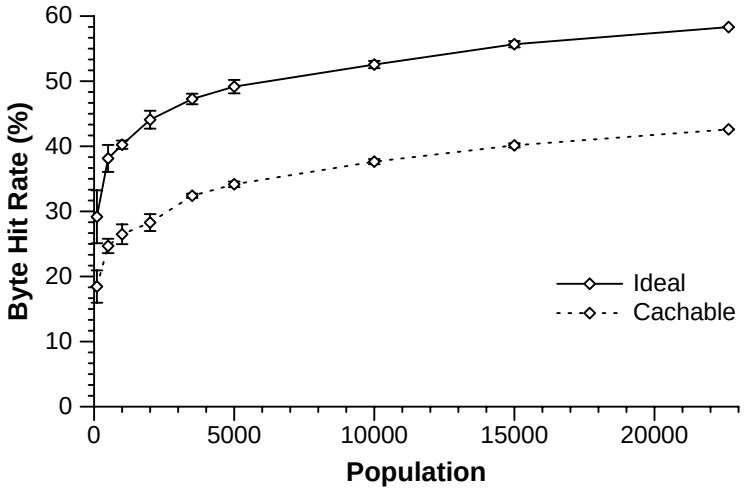


Figure 4.4: Proxy cache byte hit rate as a function of client population for the UW trace.

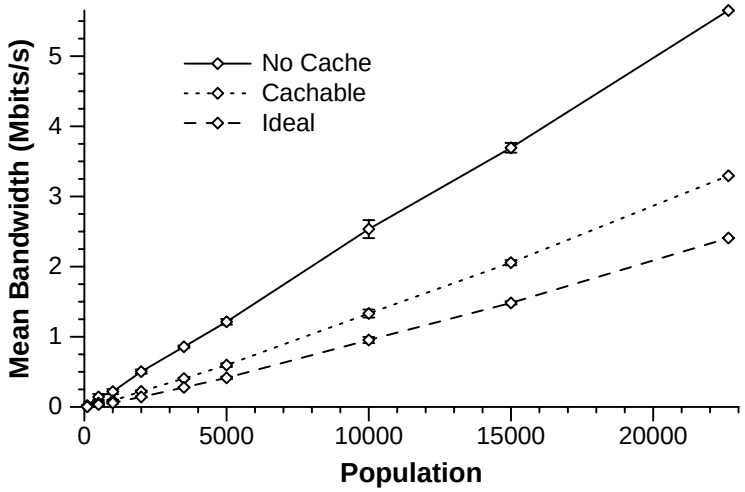


Figure 4.5: Bandwidth consumed as a function of client population for the UW trace.

independent business entity, with its own interests and focus, which would typically have its own proxy sitting on its connection to the Internet.

In fact, somewhat fortuitously, few organizations on our campus currently employ proxies; this permits us to see most outgoing client requests and their responses. In our tracing software, before anonymizing each client's IP address, we first classify that client as belonging to one of about 200 independent university organizations. In this way, we preserve organizational membership information while protecting client identities. In effect, this gives us a simultaneous trace of Web requests from 200 organizations.

Figure 4.6 shows the ideal (left-hand bar) and cachable (right-hand bar) hit rates for the 15 largest UW organizations. The bars are labeled on the x-axis with the number of clients seen in the trace for each of the organizations shown: the smallest organization had 192 clients, while the largest had 978 clients. These bars thus represent 15 medium-sized companies or client communities.

The lower portion of each bar shows the hit rate that would be seen by a local proxy acting on behalf of that organization. The upper portion of the bar shows the *improvement* in hit rate that would be seen by that organization if *all* of the university's organizations used perfectly cooperating proxies. For the ideal bars, the average hit rate per group is 52%. The average cooperative-caching hit rate, i.e., the rate that would be seen by a single proxy over all organizations or by a perfect cooperative-caching scheme, is 69%. For the cachable hit rate bars, the average hit rate per group is 29%, and the average cooperative hit rate is 38%. Therefore, if perfect cooperative caching were possible, it would achieve a noticeable improvement in hit rate for these proxies.

An interesting question raised above is whether clients in our UW population request documents randomly, or whether their access patterns are related in some way to those of other members of the same organization. To answer this question, we grouped clients at random into organizations with the same sizes as the 200 real organizations and compared local hit rates. Figure 4.7 shows a comparison of the hit rates for the 15 largest UW and randomly assigned organizations. The result indicates that there is a small average increase (about 4 percentage points) in hit rates for the real organizations compared to the randomly assigned ones. Therefore, there is some locality in organizational membership, but the impact is not large in this case.

A related question is whether a better grouping exists of clients to proxies – for example, one based on each client's document interests. To examine this question, we conducted a clustering

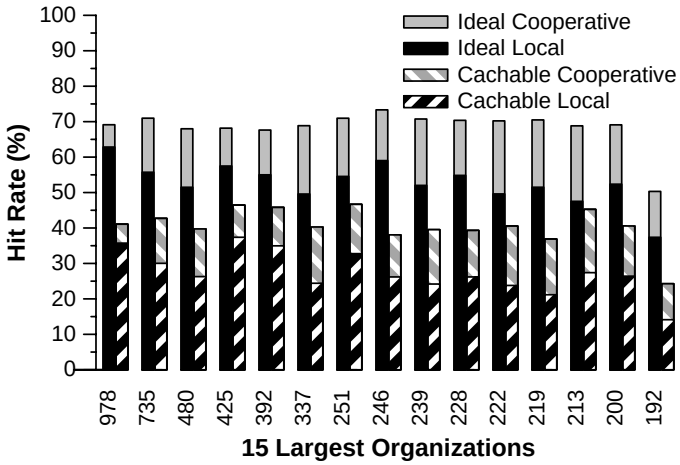


Figure 4.6: Breakdown of local and global proxy hit rates for the 15 largest UW organizations.

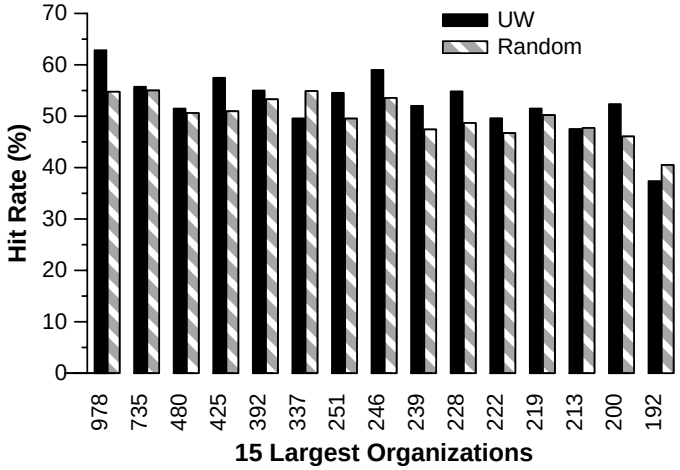


Figure 4.7: Comparison of the proxy hit rates for the 15 largest UW and randomly populated organizations.

study. Using the trace data, we clustered clients based on their document access vectors, using a standard clustering algorithm (K-Means) that attempts to optimize intra-cluster sharing. The existence of such groupings within the university might imply the existence of an improved wide-area caching scheme based on clustering.

As with the university organizations, we compared the cluster-based client assignments to random assignments of clients into groups of the same size. The randomly assigned clusters have a consistently lower hit rate than the optimally clustered organizations. Somewhat surprisingly, the difference between the two assignments is just slightly more than the difference between the UW and random organizations (about 5 percentage points). Again, there is some affinity in client access patterns, but the impact on hit rate is not large.

4.8 Impact of Larger Population Size

We have seen that cooperative caching can increase hit rate, perhaps substantially, below the knee of the curve in Figure 4.1. What happens above the knee, i.e., can cooperative caching be effective in a wide-area network? Figure 4.8 shows the data from Figure 4.1 when it has been plotted as a function of the log of the client population size, fitted linearly using least squares, and then extrapolated past the client population we measured.

This graph suggests a number of interesting conclusions. First, the slopes of the UW lines are greater than of the Microsoft lines, a relation we could only infer from Figure 4.1. Second, we notice that the slopes of the UW ideal and cachable lines are similar. This indicates that there is little correlation between sharing and the cachability of documents for the UW population. However, the slope of the Microsoft cachable line is only 60% of the slope of the ideal line. This indicates that cachable documents were shared to a lesser degree than uncachable documents for the Microsoft population. Third, the cachability curves are limited by the fraction of requests to cachable documents, which is 60% in the UW trace and 51% in the Microsoft trace. In both of these cases, cooperative caching among populations larger than 2.4 million does not increase the hit rate to cachable documents. Fourth, even if all documents were cachable, the ideal hit rate reaches a maximum at a population of 11 million users for the UW trace and 2.9 million for the Microsoft trace.

The Microsoft client population is more homogeneous than the university population and there-

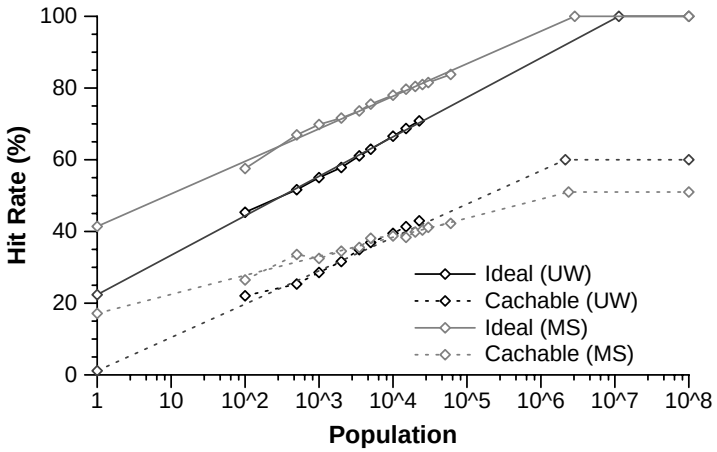


Figure 4.8: Proxy cache hit rate as a function of client population.

fore sees a higher degree of document sharing. Quantitatively, 83.8% of requests in the Microsoft trace are to previously requested documents; in the UW trace, 70.8% of requests are to previously requested documents. These statistics are the "ideal hit rates" that would be seen by a cache, if all documents were cachable. But how much overlap is there to popular documents between the two populations? We looked at the most popular documents requested by each of the two populations, where we defined "most popular" to be those documents accessed more than 500 times. In the UW trace, there were 11,500 such documents, and in the Microsoft trace, there were 17,000 such documents. Looking at the 1000 most popular in each of the two populations, we see a 33% overlap; that is, of the 1000 most-popular documents accessed by Microsoft, 330 of them are also among the 1000 most popular accessed by UW. Therefore, many of the *same* documents are popular in both organizations.

In related work [Wolman et al. 99b], we present an analytical model to look in more detail at the behavior of cooperative caching in large-scale environments. Within the context of our own traces, though, we can perform an interesting experiment to create a larger population. Suppose that we implemented cooperative caching between the University of Washington and the Microsoft proxies. From the perspective of the UW proxy, this increases the size of the population it sees by a factor of 3.6; from the point of view of the Microsoft proxy, population increases by a factor of 1.4. What is the impact of combining the two populations through cooperative caching?

To estimate the benefit of cooperative caching between the two organizations, we did the following analysis. We ran the university trace through a simulated UW proxy; we then fed all misses to a second-level (cooperating) proxy preloaded with all of the objects seen in the Microsoft trace. Similarly, we ran the Microsoft trace simulating its proxy, with a second-level proxy preloaded with all objects seen by the UW trace. This gives us an idea of the maximum *incremental* hit-rate benefit each proxy would see if the two proxies were to cooperate.

Figure 4.9 shows the results of this measurement. From the figure, we see that the UW proxy, whose effective population would increase 3.6-fold (from 23,000 to 83,000 clients), would see its ideal hit rate increase only 5.1 percentage points – from 70.1% to 75.9%; its cachable hit rate would increase only 4.2 percentage points – from 42.7% to 46.9%. The Microsoft proxy, whose effective population would increase by a factor of 1.4 (from 60,000 to 83,000 clients), would see its ideal hit rate increase 2.7 percentage points – from 83.8% to 86.5%; its cachable hit rate would increase only 2.1 percentage points – from 42.3% to 44.4%. To allow a more direct comparison of these results, we ran another experiment where the second-level cooperating proxy was preloaded by only a portion of the Microsoft population, thereby increasing the effective population size for the UW population by the same factor of 1.4. In this experiment, the ideal hit rate for the UW proxy increased by 1.6 percentage points, and the cachable hit rate increased by 1.3 percentage points. When scaled by equal factors, it is interesting to note that Microsoft gains more benefit by cooperating with the UW population than the UW population gains by cooperating with Microsoft.

These results are disappointing, but not surprising. The reason for these very small increases is that the *unpopular* documents are universally unpopular; therefore, it is unlikely that a miss in one of these large populations will find the document in the other population's proxy. For the most-popular documents, cooperation does not help either, because only the first access (of the 500 plus accesses to a popular document) has the potential to benefit from another population's proxy.

4.9 Summary and Conclusions

This chapter studied cooperative proxy caching in local- and wide-area environments. We collected and analyzed traces from two different large organizations to explore the effectiveness of cooperative-caching at a wide range of population sizes.

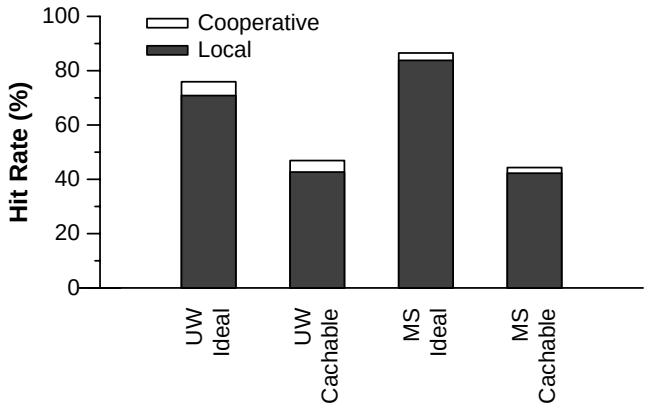


Figure 4.9: Hit rate benefit of cooperative caching between UW and Microsoft proxies.

At a high level, our results show that:

1. The behavior of cooperative caching is characterized by two different regions of the hit rate vs. population curve. For smaller populations, hit rate increases rapidly with population; it is in this region that cooperative caching can be used effectively. However, these population sizes can be handled by a single proxy. Therefore, cooperative caching is only necessary to adapt to proxy assignments made for political or geographical reasons.

2. For larger populations (beyond the knee of the population vs. hit rate curve), cooperative caching is unlikely to provide significant benefit. We demonstrate this using our simultaneous traces of the Microsoft and the University of Washington populations: a four-fold increase to the large university population via cooperative caching netted only an increase of 2.7 percentage points in cachable hit rate.

3. In the absence of significant changes in client behavior, there is little point in continuing to expend effort on the design and evaluation of highly scalable, cooperative-caching schemes. The scale at which cooperative caching makes sense is sufficiently small that any reasonable scheme will achieve most of the benefit.

4. Performance at the population level at which cooperative caching works effectively is basically limited by document *cachability*. Therefore, increasing cachability of documents is the

main challenge for research aimed at improving Web cache behavior.

5. Cluster-based analysis of client access patterns indicates that cooperative-caching organizations based on mutual interest offer no obvious advantages over randomly assigned or organization-based groupings.

Fundamentally, the usefulness of cooperative Web proxy caching depends upon the scale at which it is being applied. From our trace data of users at the University of Washington and Microsoft Corporation, cooperative Web proxy caching is an effective architecture for small individual caches that together comprise user populations in the tens of thousands. At such small scales, any reasonable cooperative caching scheme will serve. But cooperative caching is not required for user populations of this size. If it is administratively and politically feasible, a single proxy cache can provide the same benefits with fewer resources and less overhead.

Whether or not they use cooperative caching locally, large organizations should use proxy caching for their user populations. A key issue is whether these large organizational caches benefit from cooperating. Our experimental results indicate that the benefit of cooperation will be small, and therefore will only make sense if the cost of cooperation is also small. This is most likely to be the case in very high-bandwidth, low-latency environments.

In a related study not included in this dissertation, we developed an analytic model of Web behavior to extend beyond the population limits of our trace results. This model permits us to explore the steady state behavior of caching systems with millions of clients, and it also allows us to investigate how changes in workload characteristics would affect the importance of cooperative Web proxy caching in the future. The results of that study reconfirm our finding that cooperative caching is most effect a limited population sizes.

Finally, we note that our results on cooperative caching are based upon currently observed Web workload behavior. Fundamental shifts in Web workload characteristics might change these results. For example, the workloads we have examined consist primarily of static documents. But we have also observed a growing presence in Web workloads of streaming multimedia traffic [Wolman et al. 99a], and streaming multimedia objects have different characteristics than static Web objects. Their average size is orders of magnitude larger, so cooperative caching for storage efficiency becomes more appealing. Furthermore, last-byte latency is not a critical performance

metric for streaming data. Instead, reducing jitter and making more effective use of the network become more important. Lastly, given the size of streaming objects, and the relatively long period of time over which they are transferred over the network, transport optimizations like multicast might prove more effective.

Chapter 5

Measurement and Analysis of a Streaming-Media Workload

5.1 Introduction

Today's Internet is increasingly used for transfer of continuous-media data, such as video from news, sports, and entertainment Web sites, and audio from Internet broadcast radio and telephony. As evidence, a large 1996 Web study from U.C. Berkeley [Gribble et al. 97] found no appreciable use of streaming media, but our own study three years later at the University of Washington found that RealAudio and RealVideo had become a considerable component of Web-related traffic [Wolman et al. 99a]. In addition, new peer-to-peer networks such as Napster have dramatically increased the use of digital audio over the Internet. A 2000 study of the IP traffic workload seen at the NASA Ames Internet Exchange found that traffic due to Napster rose from 2% to 4% over the course of 10 months [McCreary et al. 00], and a March 2000 study of Internet traffic at the University of Wisconsin-Madison found that 23% of its traffic was due to Napster [Plonka 00].

Streaming-media content presents a number of new challenges to systems designers. First, compared to traditional Internet applications such as email and Web browsing, multimedia streams can require high data rates and consume significant bandwidth. Second, streaming data is often transmitted over UDP [Mena et al. 00], placing responsibility for congestion control on the application or application-layer protocol. Third, the traffic generated tends to be bursty [Mena et al. 00] and is highly sensitive to delay. Fourth, streaming-media objects require significantly more storage than traditional Web objects, potentially increasing the storage requirements of media servers and proxy caches, and motivating more complex cache replacement policies. Fifth, because media streams have long durations compared to the request/response nature of traditional Web traffic, multiple simultaneous requests to shared media objects introduce the opportunity for using multicast delivery techniques to reduce the network utilization for transmitting popular media objects. Unfortunately, despite these new characteristics and the challenges of a rapidly growing traffic component, few

detailed studies of multimedia workloads exit.

This chapter presents and analyzes a client-based streaming-media workload. To capture this workload, we extended our existing HTTP passive network monitor (described in Chapter 6) to trace key events from multimedia sessions initiated inside the University of Washington to servers in the Internet. For this analysis, we use a week-long trace of RTSP sessions from 4,786 university clients to 23,738 distinct streaming-media objects from 866 servers in the Internet, which together consumed 56 GB of bandwidth.

The primary goal of our analysis is to characterize this streaming-media workload and compare it to well-studied HTTP Web workloads in terms of bandwidth utilization, server and object popularity, and sharing patterns. In particular, we wish to examine unique aspects of streaming-media workloads, such as session duration, session bit-rate, and the temporal locality and degree of overlap of multiple requests to the same media object. Finally, we wish to consider the effectiveness of performance optimizations, such as proxy caching and multicast delivery, on streaming-media workloads.

Our analysis shows that, for our trace, most streaming-media objects accessed by clients are encoded at low bit-rates (< 56 Kb/s), are modest in size (< 1 MB), and tend to be short in duration (< 10 mins). However, a small percentage of the requests (3%) are responsible for almost half of the total bytes downloaded. We find that the distributions of client requests to multimedia servers and objects are somewhat less skewed towards the popular servers and objects than with traditional Web requests. We also find that the degree of multimedia object sharing is smaller than for traditional Web objects for the same client population. Shared multimedia objects do exhibit a high degree of temporal locality, with 20–40% of active sessions during peak loads sharing streams concurrently; this suggests that multicast delivery can potentially exploit the multimedia object sharing that exists.

The rest of this chapter is organized as follows. Section 5.2 provides a high-level description of streaming-media protocols for background. Section 5.3 describes the trace collection methodology. Section 5.4 presents the basic workload characteristics, while Section 5.5 focuses on our cache simulation results. Section 5.6 presents our stream merging results. Finally, Section 5.7 concludes.

5.2 Streaming Media Background

This section defines a number of terms and concepts that are used throughout this chapter. We use the term streaming media to refer to the transfer of live or stored multimedia data where the media player begins rendering as soon as the data is received, rather than waiting for the full download to complete before rendering begins. Although streaming techniques are typically used to transfer audio and video streams, they are sometimes used to deliver traditional media (such as streaming text or still images). A wide variety of streaming-media applications are in use today on the Internet. They employ a wide variety of protocols and algorithms that can generally be classified into five categories:

1. **Stream control protocols** enable users to interactively control playback of the media stream, e.g., pausing, rewinding, forwarding or stopping stream playback. Examples of commonly used stream control protocols include RTSP [Schulzrinne et al. 98], PNA [RealNetworks 01a], MMS [Microsoft 01b] and XDMA [Xingtech 00]. These protocols typically rely on TCP as the underlying transport protocol.

2. **Media packet protocols** support real-time data delivery and facilitate the synchronization of multiple streams. These protocols define how a media server encapsulates a media stream into data packets, and how a media player decodes the received data. Most media packet protocols rely on UDP to transport the packets. Examples include RDT [RealNetworks 01b], RTP [Schulzrinne et al. 96], PNA [RealNetworks 01a], MMSU and MMST [Microsoft 01b].

3. **Encoding formats** dictate how a digitized media stream is represented in a compact form suitable for streaming. Examples of encoding schemes commonly used include WMA [Microsoft 01b], MP3 [MPEG-2 Audio 94], MPEG-2 [MPEG-2 Video 94], RealAudio G2 and RealVideo G2 [RealNetworks 01c].

4. **Storage formats** define how encoded media streams are stored in “container” files, which hold one or more streams. Headers in the container files can be used to specify the properties of a stream such as the encoding bit-rate, the object duration, the media content type, and the

object name. ASF [Fleischman 98] and RMFF [Agarwal et al. 98] are examples of container file formats.

5. **Metafile formats** provide primitives that can be used to identify the components (URLs) in a media presentation and define their temporal and spatial attributes. SDP [Handley et al. 98], SMIL [W3C 98] and ASX [Microsoft 01a] are examples of metafile formats.

5.3 Methodology

We collected a continuous trace of RTSP traffic flowing across the border routers serving the University of Washington campus over a one week period between April 18th and April 25th, 2000. In addition to monitoring RTSP streams, the trace tool also maintained connection counts for other popular stream control protocols: PNA [RealNetworks 01a] used by Real Networks' servers, MMS [Microsoft 01b] used by Microsoft Windows Media servers, and XDMA [Xingtech 00], used by Xing's StreamWorks servers. We collected the trace data for this study using our own custom trace collection software. A detailed description of this trace collection and analysis infrastructure is covered in Chapter 6. In the remainder of this section, we provide a high-level overview of the RTSP protocol and the information collected by our tracing system to facilitate interpretation of our results.

5.3.1 RTSP Trace Collection

Capturing streaming-media traffic is challenging because applications may use a variety of protocols. Moreover, while efforts have been made to develop common standardized protocols, many commercial applications continue to use proprietary protocols. Given the diversity of protocols, we decided to focus initially on the standardized and well documented RTSP protocol [Schulzrinne et al. 98]. Widely used media players that support RTSP include Real Networks' RealPlayer and Apple's QuickTime Player. In Section 5.4 we provide data that indicates that RTSP is the most widely used streaming control protocol at the University of Washington during our trace period.

RTSP, a protocol similar to HTTP, is used by media players and streaming servers to control

the transmission of streaming content. In typical operation, the RTSP control traffic is always sent over TCP whereas the media data is often sent over UDP, and less frequently the media data is interleaved with the control traffic on the same TCP connection. Figure 5.1 illustrates a common streaming media usage scenario. First, a user downloads a Web page that contains a link to a media presentation. This link points to a metafile hosted by the media server. The Web browser then downloads the metafile that contains RTSP URLs for all the multimedia objects in the presentation (e.g., a music clip and streaming text associated with the audio). Next, the browser launches the appropriate media player and passes the metafile contents to the player. The media player in turn parses the metafile and initiates an RTSP connection to the media server.

Many of our analysis results will refer to an RTSP "session." An RTSP session is similar to an HTTP "GET" request, in that typically there will be one session for each access to the object. A session begins when the media player first accesses the object, and it ends when the media player sends a TEARDOWN message, though there may be a number of intervening PAUSE and PLAY events. There is not a one-to-one mapping between sessions and RTSP control connections; instead, the protocol relies on session identifiers to distinguish among different streams. In order to make our results easier to understand, when a single RTSP session accesses multiple objects, we consider it to be multiple sessions – one for each object.

We extended our existing HTTP passive network monitor to support monitoring the RTSP streaming-media protocol. The RTSP protocol parsing module extracts pertinent information from RTSP headers such as media stream object names (URLs), transport parameters, and stream play ranges. Data in the RTSP control connections is also used to determine which UDP datagrams to look at when the media data is not interleaved. We record timing and size information about the UDP data transfers, but we do not attempt to process the contents of media stream packets because almost all commonly used encoding formats and packet protocols are proprietary. All sensitive information extracted by the parser, such as IP addresses and URLs, is anonymized to protect user privacy.

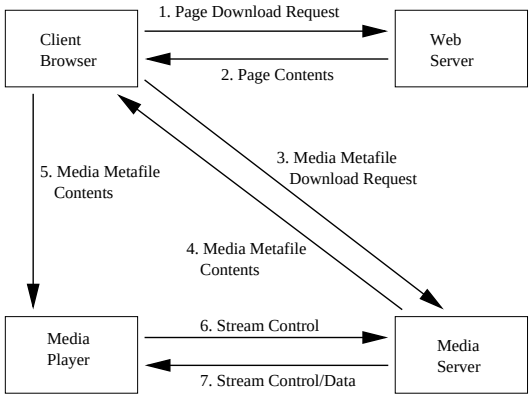


Figure 5.1: A typical initialization sequence for viewing streaming media content.

5.4 Workload Characterization

This section analyzes the basic characteristics of our streaming-media workload; when appropriate we compare these characteristics to those of standard Web object workloads. The analysis ignores non-continuous media data (e.g., streaming text and still images). Since we were interested in the access patterns of the UW client population, we ignored sessions initiated by clients external to UW that accessed servers inside the campus network.

Table 5.1 summarizes the high-level characteristics of the trace. During this one-week period, 4,786 UW clients accessed 23,738 distinct RTSP objects from 866 servers, transferring 56 GB of streaming media data. Using the connection counts from Table 5.2, we see that RTSP is the most widely used stream control protocol at the UW during this period. Furthermore, we can use these connection counts to estimate that RTSP accounts for approximately 40% of all streaming media usage by UW clients.

The detailed analyses in the following sections examine various attributes of the traffic workload, such as popularity distributions, object size distributions, sharing patterns, bandwidth utilization, and temporal locality. Overall, our analysis shows that:

- Most of the streaming data accessed by clients is transmitted at low bit-rates: 81% of the accesses are transmitted at a bit-rate less that 56 Kb/s.

Table 5.1: Overall statistics for the UW RTSP trace.

RTSP Attribute	Value
External Servers	866
Continuous Media Total Bytes	56 GB
UW Clients	4786
Continuous Media Sessions	40070
Continuous Media Objects	23738
Other Objects	3760

- Most of the media streams accessed have a short duration (< 10 minutes) and a modest size (< 1 MB).
- A small percentage of the sessions (3%) are responsible for almost half of the bytes downloaded.
- Streaming media sessions are much larger than HTTP responses. The median session size is 400 times larger than the median HTTP response size, and the mean session size is 175 times larger than the mean HTTP response size.
- The distribution of client requests to objects is Zipf-like, with an α parameter of 0.47.
- While clients do share streaming-media objects, the degree of object sharing is not as high as that observed in Web traffic traces [Duska et al. 97, Wolman et al. 99a].
- There is a high degree of temporal locality in client requests to repeatedly accessed objects.

5.4.1 Bandwidth Utilization

Figure 5.2 shows a time-series plot of the aggregate bandwidth consumed by clients streaming animations, audio, and video data. We see that the offered load on the network follows a diurnal cycle,

Table 5.2: Stream control protocol connection counts.

Protocol	Count
RTSP	58808
MMS	44878
PNA	35230
XDMA	3930

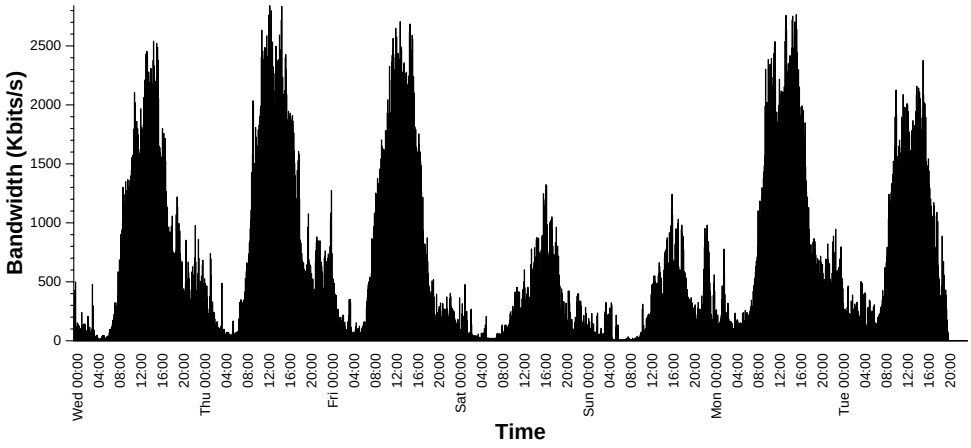


Figure 5.2: Total bandwidth used over time (in Kbits/sec).

with peaks generally between 11 AM and 4 PM. The volume of traffic is significantly lower during weekends; peak bandwidth over a five-minute period was 2.8 Mb/s during weekdays, compared to 1.3 Mb/s on weekends. We found that, on average, clients received streaming content at the rate of 66 Kb/s. This bit-rate is much lower than both the internal UW link capacities (usually 10 Mbps or 100 Mbps) and the current UW ISP link capacity (200 Mbps). We conclude from the prevalence of these low-bit-rate sessions that the sites that clients are accessing encode streaming content primarily at modem bit-rates, the lowest common denominator.

5.4.2 Advertised Stream Length

In this section, we provide a detailed analysis of the advertised duration of continuous media streams referenced during the trace. Note that the advertised duration of a stream is different from the length of time that the client actually downloads the stream (e.g., if the user hits the stop button before the stream is finished). Since media servers generally do not advertise the duration of live streams, we limit our analysis to on-demand (stored) media streams. Sessions accessing these on-demand streams account for 85% of all sessions.

Figure 5.3a presents a histogram of all streams lasting less than seven minutes, and Figure 5.3b plots the cumulative distribution of all stream lengths advertised by media servers. The peaks in the histogram in Figure 5.3a indicate that many streams are extremely short (less than a minute), but the most common stream lengths are between 2.5 and 4.5 minutes. These results suggest that clients have a stronger preference for viewing short multimedia streams. One important observation from Figure 5.3b is that the stream-length distribution has a long tail. Although the vast majority of the streams (93%) have a duration of less than 10 minutes, the remaining 7% of the objects have advertised lengths that range between 10 minutes and 6 weeks.

In Figure 5.4, we separate stream accesses into two categories. The grey curve shows the cumulative distribution of advertised stream lengths for those stream accesses made by clients in the campus modem pool, and the black curve shows the cumulative distribution of advertised stream lengths for streams accessed by clients connected by high-speed department LANs. From this figure, we see that 40% of the streams accessed by modem clients have an advertised length less than 100 seconds, whereas for LAN clients only 23% of the streams have an advertised length less than 100 seconds. This reconfirms our expectation that clients with poor connectivity are somewhat more likely to access shorter streaming-media content.

In Figure 5.5, we examine the relationship between the advertised stream length and the actual download stream length. These two lengths can differ if the client terminates viewing the stream before the stream has finished downloading, or they can differ if the server simply advertises an inaccurate stream length. The largest gap between the two curves in this figure occurs for streams that last less than 3 minutes. This indicates that clients often choose to terminate streams early, and this termination usually occurs within the first few minutes of viewing the stream.

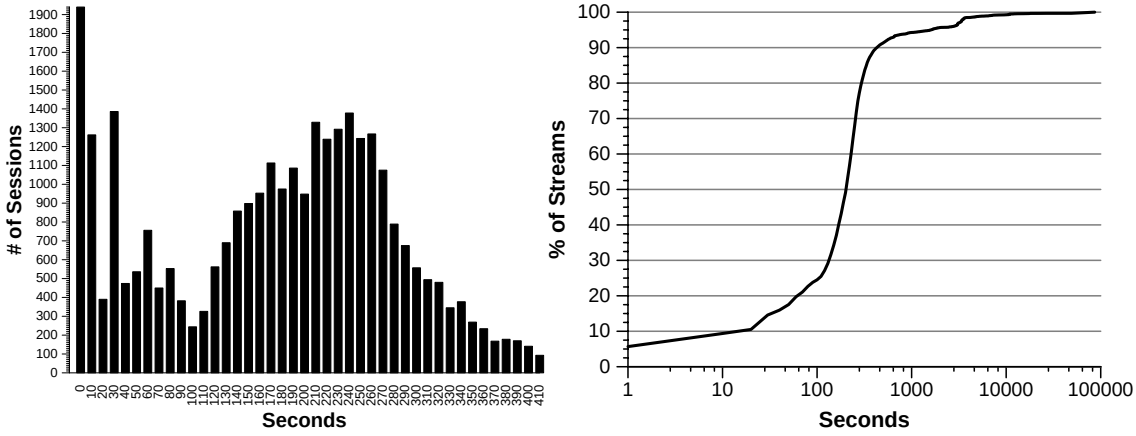


Figure 5.3: The left graph (a) shows a histogram of advertised stream lengths for all streams less than 7 minutes. The right graph (b) shows the cumulative distribution of advertised stream lengths.

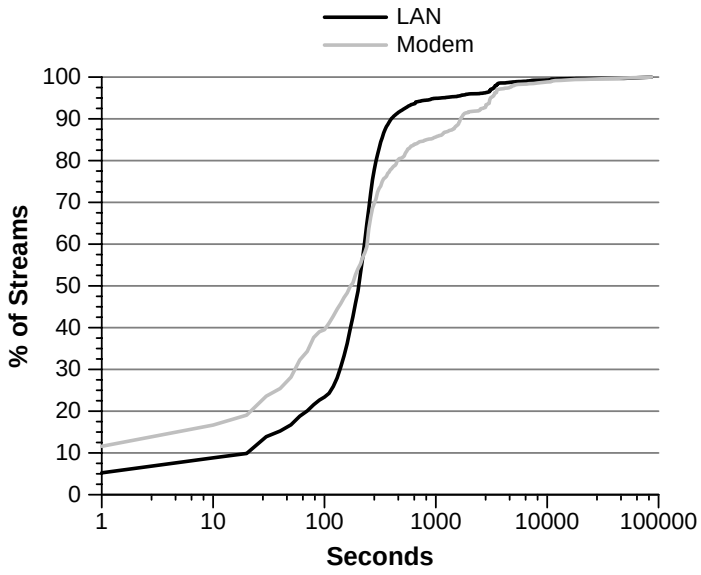


Figure 5.4: Cumulative distributions of advertised stream lengths for modem clients and LAN clients.

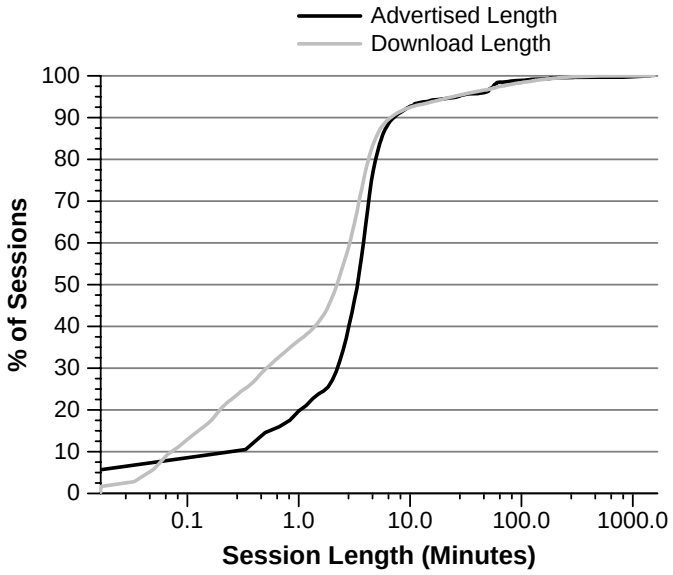


Figure 5.5: Comparison of the stream download length cumulative distribution and the advertised stream length cumulative distribution.

5.4.3 Session Characteristics

In this section, we examine two closely related properties of sessions: the amount of time that a client spends accessing a media stream, and the number of bytes downloaded during a stream access. In Figure 5.6 we present the relationship between the duration of a streaming-media session and the number of bytes transferred. In Figure 5.7 we look at the distinguishing characteristics between sessions accessing shared objects and sessions accessing unshared objects. Finally, in Figure 5.8 we compare the size and duration characteristics of sessions from clients in the campus modem pool to sessions from clients connected by high-speed department LANs.

A number of important trends emerge from these graphs. First, we see that client sessions tend to be short. From Figure 5.7a we see that 85% of all sessions (the solid black line) lasted less than 5 minutes, and the median session duration was 2.2 minutes. The bandwidth consumed by most sessions was also relatively modest. From Figure 5.7b we see that 84% of the sessions transferred less than 1 MB of data and only 5% accessed more than 5 MB. In terms of bytes downloaded, the median session size was 0.5 MB. Both the session duration and the session size distributions have long tails: 20 sessions accessed more than 100 MB of data each, while 57 sessions remained

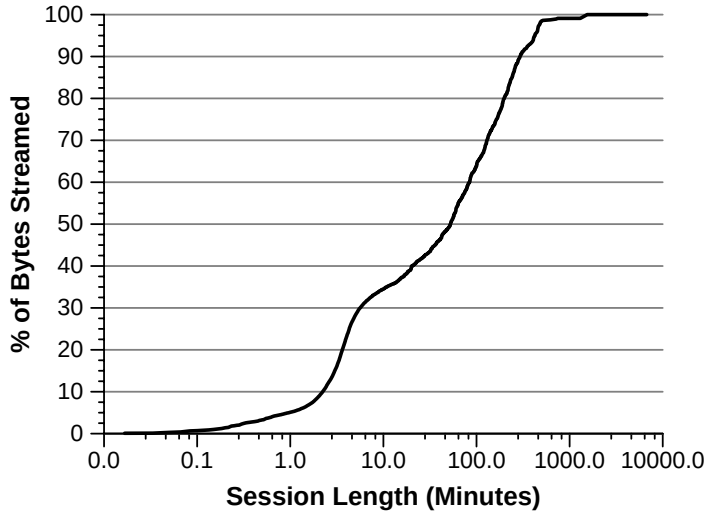


Figure 5.6: Cumulative distribution of bytes transferred by sessions of a given length.

active for at least 6 hours, and one session was active for 4 days. Although the long-lived sessions ($>$ 1 hour) account for only 3% of all client sessions, these sessions account for about half of the bandwidth consumed by the workload. From Figure 5.6 we see that these long sessions account for 47% of all bytes downloaded. The size of a typical streaming media session is considerably larger than the size of a typical Web request. In our RTSP trace, we found that the median session size was 0.5 MB, and the mean session size was 1.4 MB. In our HTTP trace of the same population collected one year earlier, we found that the median response size was just over 1 KB and the mean response size was 8 KB. Therefore, when comparing medians we see that streaming media sessions are 400 times larger than HTTP responses, and when comparing means we see that streaming media sessions are 175 times larger than HTTP responses.

Most of the media objects accessed are downloaded at relatively low bit-rates despite the fact that most of the clients are connected by high-speed links. Using the raw data from Figure 5.6, we calculated that 81% of the streams are downloaded at bit-rates less than 56 Kb/s (the peak bandwidth supported by most modems today). In Figure 5.8, we separate all the trace sessions into those made from clients in the modem pool and those made from LAN clients. Although it does appear that the duration of modem sessions is shorter than the duration of LAN sessions (Figure 5.8a), the difference is not large. On the other hand, the difference in bytes downloaded between modem

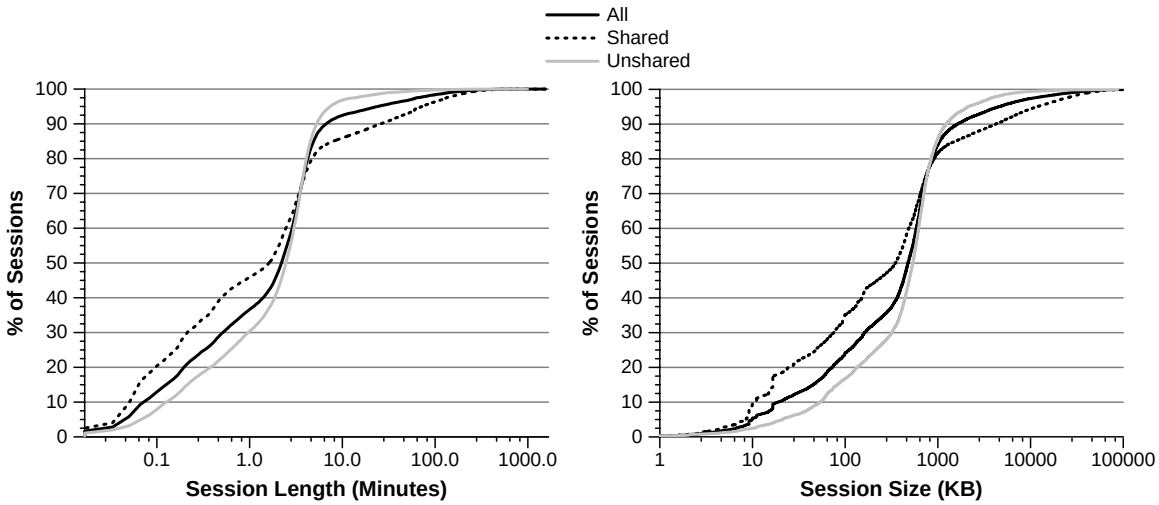


Figure 5.7: Shared and unshared session characteristics. The left graph (a) shows the cumulative distribution of session download lengths for all, shared, and unshared sessions. The right graph (b) shows the cumulative distribution of session sizes for all, shared, and unshared sessions.

sessions and LAN sessions (Figure 5.8b) appears to be much more pronounced. For modem users, the median session size is just 97 KB, whereas for LAN users it is more than 500 KB.

Figures 5.7a and 5.7b also distinguish between accesses to shared objects (the dashed lines) and accesses to unshared objects (the grey lines). A shared object is one that is accessed by more than one client in the trace; an unshared object is accessed by only one client, although it may be accessed multiple times. Overall, sessions that request shared objects tend to be shorter than sessions accessing unshared objects. For example, 46% of shared sessions lasted less than one minute, compared with only 30% of the unshared sessions. Furthermore, we found that most of the sessions accessing shared objects transferred less data than unshared sessions. For example, 44% of shared sessions transferred less than 200 KB of data compared to only 24% of unshared sessions. However, Figure 5.7 shows that the situation changes for sessions on the tails of both curves, where the sessions accessing shared objects are somewhat longer and larger than sessions accessing unshared objects.

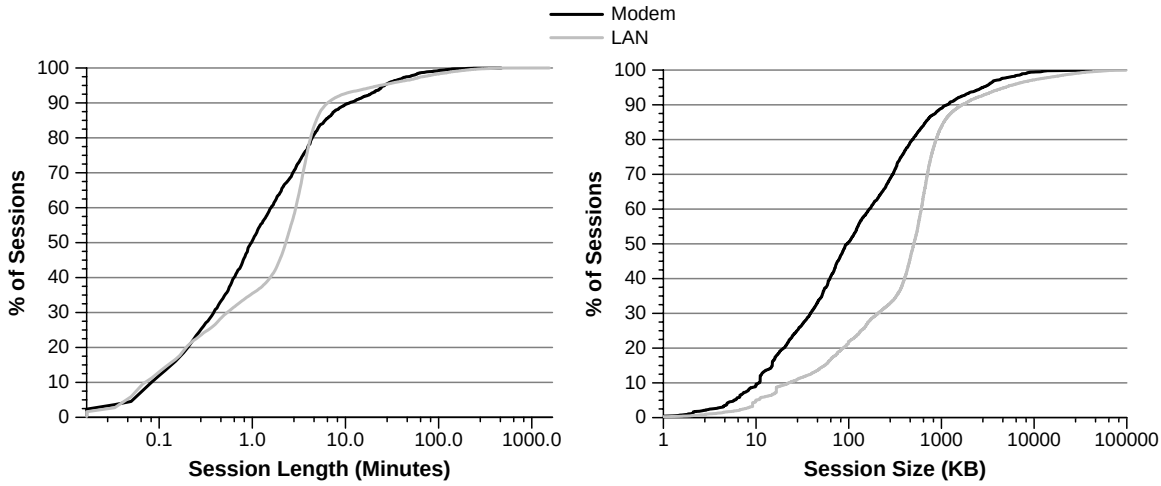


Figure 5.8: Modem and LAN session characteristics. The left graph (a) shows the cumulative distribution of session download lengths for modem and LAN sessions. The right graph (b) shows the cumulative distribution of session sizes for modem and LAN sessions.

5.4.4 Server Popularity

In this section we examine the popularity of media servers and objects. Figure 5.9 plots (a) the cumulative distribution of continuous media objects across the 866 distinct servers referenced, as well as (b) the cumulative distribution of requests to these servers. These graphs show that client load is heavily skewed towards the popular servers. For example, 80% of the streaming-media sessions were served by the top 58 (7%) media servers, and 80% of the streaming-media objects originated from the 33 (4%) most popular servers. This skew to popular servers is *slightly less pronounced* than for requests to non-streaming Web objects. From a May 1999 trace of the same client population, 80% of the requests to non-streaming Web objects were served by the top 3% of Web servers [Wolman et al. 99a].

5.4.5 Object Popularity

One of the goals of our analysis was to understand how client requests were distributed over the set of multimedia streams accessed during the trace period. To determine this, we ranked multimedia objects by popularity (based on the number of accesses to each stream) and plotted the results on

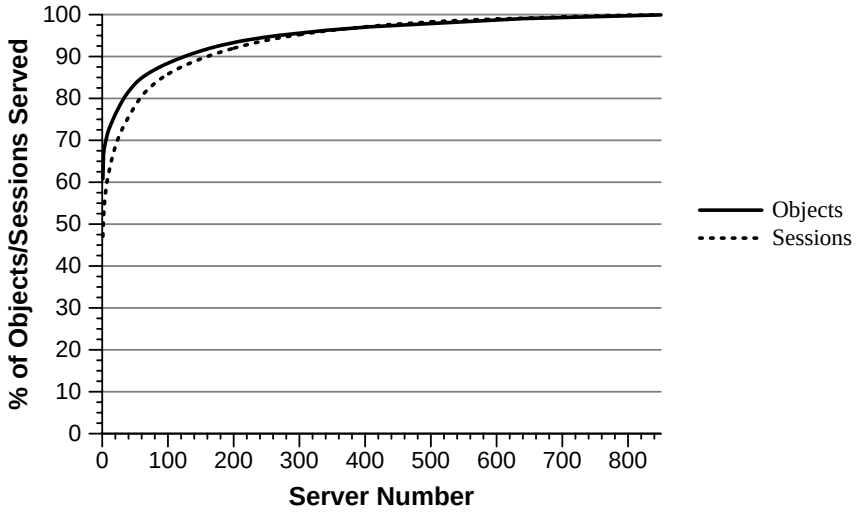


Figure 5.9: Cumulative distribution of server popularity (in terms of both objects and sessions).

the log-scale graph shown in Figure 5.10. Our analysis found that of the 23,738 media objects referenced, 78% were accessed only once. Only 1% of the objects were accessed by ten or more sessions, and the 12 most popular objects were accessed more than 100 times each. From Figure 5.10, one can see that the popularity distribution fits a straight line fairly well, which implies that the distribution is Zipf-like [Breslau et al. 99]. Using the least squares method, we calculated the α parameter to be 0.47. In contrast, the α parameters reported in [Breslau et al. 99] for HTTP proxies ranged from 0.64 to 0.83. The implication is that accesses to streaming-media objects are somewhat less concentrated on the popular objects in comparison with previously reported Web object popularity distributions.

5.4.6 Sharing patterns

In this section we explore the sharing patterns of streaming-media objects among clients. We first examine the most popular objects to determine whether the repeated accesses come from a single client, or whether those popular objects are widely shared. In Figure 5.11, we compute the number of unique clients that access each of the 200 most popular streaming-media objects. This figure shows that the most popular streams are widely shared, and that as the popularity declines, so does the number of unique clients that access the stream.

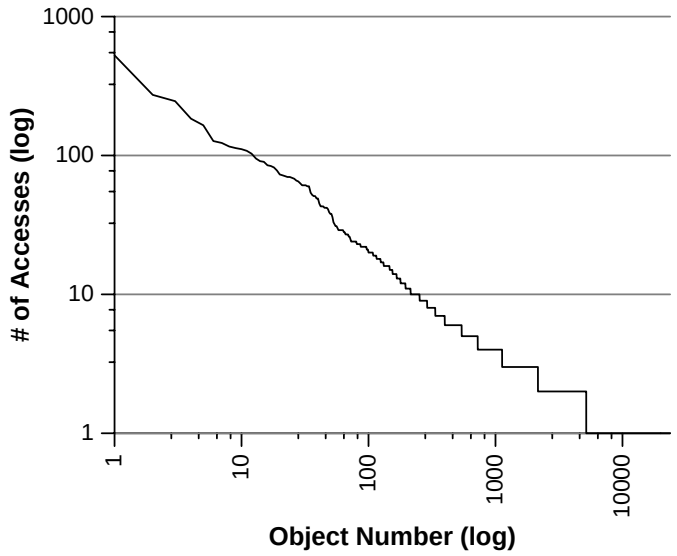


Figure 5.10: Object popularity by number of sessions. Note that both x- and y-axes use a log scale.

Figure 5.12 presents per-object sharing statistics. Of the streaming-media objects requested, only 1.6% were accessed by five or more clients, while 84% were viewed by only one client. Only 16% of the objects were shared (i.e., accessed by two or more clients), yet requests for these shared objects account for 40% of all sessions recorded. From this data, we conclude that the shared objects are also more frequently accessed and can therefore benefit from caching. Note, however, that the degree of object sharing is low compared to the sharing rate for Web documents [Duska et al. 97, Wolman et al. 99a]. Consequently, multimedia caching may not be as effective as Web caching in improving performance.

Figure 5.13 shows the overlap among accesses to the shared media objects by plotting the number of sessions that access unshared objects (black) and the number of sessions that access shared objects (grey) over time for the entire trace. During peak load periods between 11 AM and 4 PM (weekdays), we see that 20%–40% of the active sessions share streams concurrently. This temporal locality suggests that (1) caching will work best when it is needed the most (during peak loads), and that (2) multicast delivery has the opportunity to exploit temporal locality and considerably reduce network utilization.

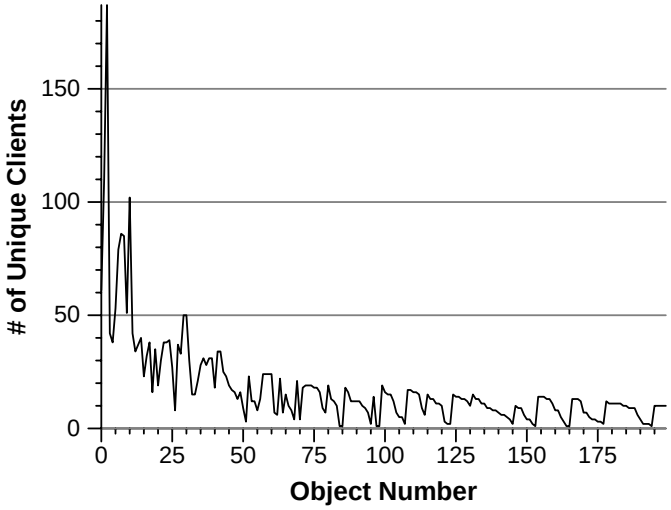


Figure 5.11: Number of unique clients that access the 200 most popular objects in the trace. The x-axis shows objects ordered from left to right by the total number of accesses to each object.

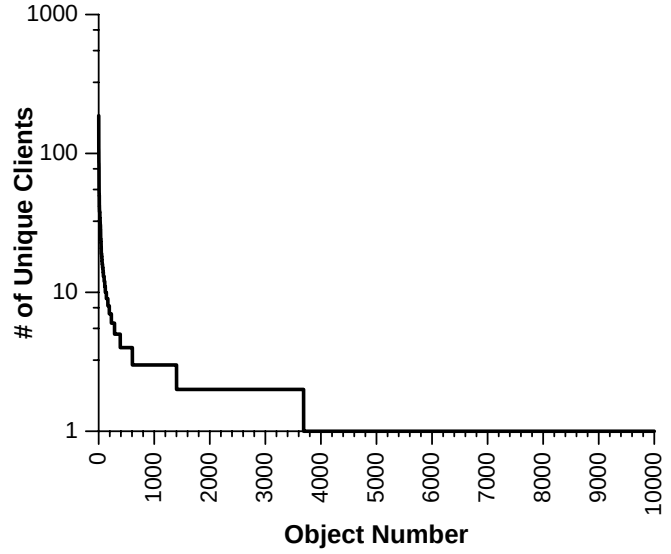


Figure 5.12: Object sharing. The x-axis shows objects ordered from left to right by the number of unique clients that access each object.

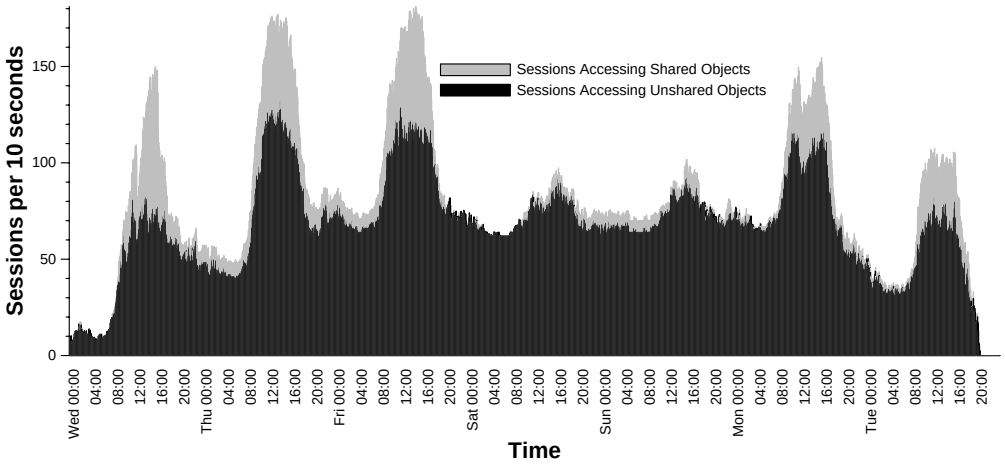


Figure 5.13: Concurrent sharing over time. The trace is divided up into 10 second segments, and a session is classified as shared if there are multiple clients that access the same object during that same segment.

5.5 Caching

Caching is an important performance optimization for standard Web objects. It has been used effectively for HTTP to reduce average download latency, network utilization, and server load. Given the large sizes of streaming-media objects and the significant network bandwidth that they can consume, caching may also be important for improving the delivery of streaming-media objects. Note, however, that the motivation for deploying proxy caches for streaming-media objects is somewhat different than for HTTP objects. Proxy cache hits for streaming objects do not improve the download latency on behalf of individual users. Instead, for a shared access links of an ISP, a proxy cache may reduce network utilization on that link. If caching of streaming objects is widely deployed, it can also reduce server load which effectively increases the peak capacity of a streaming-media server. In this section, we study the potential benefits of proxy caching for streaming-media objects. In particular, we determine cache hit rates and bandwidth savings for our workload, explore the tradeoff of cache storage and hit rate, and examine the sensitivity of hit rate to eviction timeouts.

We use a simulator to model a streaming media caching system for our analyses. The simulator caches the entire portion of any on-demand stream retrieved by a client, making the simplifying as-

sumption that it is allowed to cache all stored media objects. For live streams, the simulator assumes that the cache can effectively merge multiple client accesses to the same live stream by caching a fixed-size sliding interval of bytes from that stream [Eager et al. 00]. The simulator assumes unlimited cache capacity, and it uses a timeout-based cache eviction policy to expire cached objects. For Figures 5.14, 5.15, and 5.16, an object is removed from the cache two hours after the end of the most recent access.

The results of the simulation are presented in the set of graphs below. Figure 5.14 is a time-series plot showing cache size growth over time, while Figure 5.15 shows potential bandwidth savings due to caching. The total height of each bar in the stacked bar graph in Figure 5.16 reflects the total number of client accesses started within a one-hour time window. The lightest area of the graph shows the number of accesses that requested fully-cached objects; the medium-grey section represents the number of accesses that resulted in partial cache hits. Partial cache hits are recorded when a later request retrieves a larger portion of the media stream than was previously accessed. The height of the darkest part of the graph represents the number of accesses that resulted in cache misses.

Since streaming objects are comparatively large in size, the replacement policy for streaming proxy caches may be an important design decision. Many proposed designs for streaming proxy caches assume that multimedia streams are too large to be cached in their entirety [Rejaie et al. 99, Sen et al. 99]. As a result, specialized caches are designed to cache only selected portions of a media stream; uncached portions of the stream have to be retrieved from the server (or neighboring caches) to satisfy client requests.

To determine the need for these complex caching strategies, we explored the sensitivity of hit rate to cache replacement eviction policies by varying the timeout for cached objects. Using the default two hour expiration of our simulator, we found that the simulated cache achieved an aggregate request hit rate of 24% (including partial cache hits) and a byte hit rate of 24% using less than 940 MB of cache storage. Because the required cache size is relatively small (when compared to the total 56 GB of data transferred), it appears that conventional caching techniques and short expiration times might be just as effective as specialized streaming media caching schemes for client populations similar to this.

Figure 5.17 plots request and byte hit rates as object eviction time is increased from 5 minutes

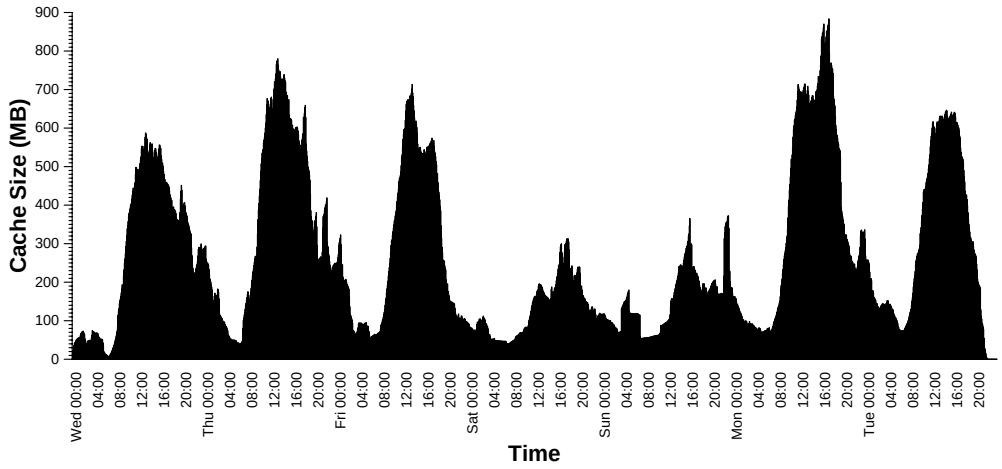


Figure 5.14: Cache size growth over time. This graph shows the total storage requirements of a simulated streaming-media proxy cache over the trace period.

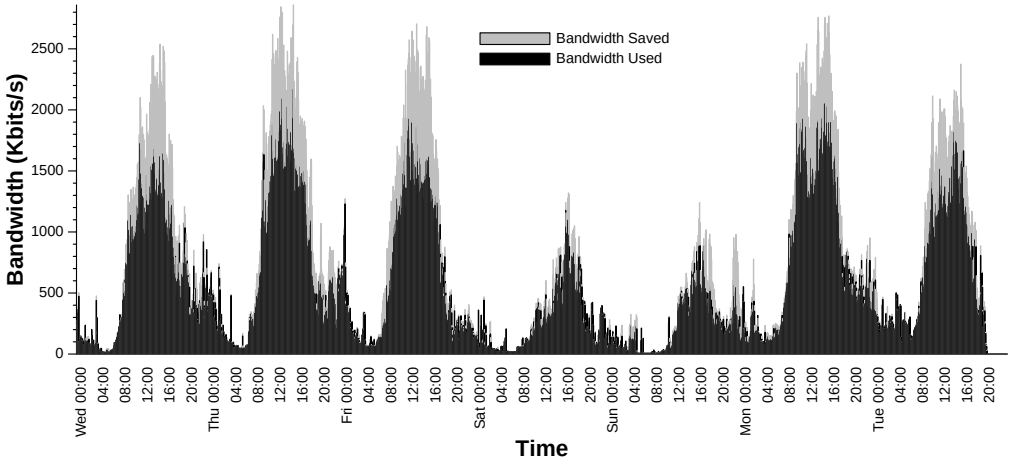


Figure 5.15: Bandwidth saved over time due to proxy caching.

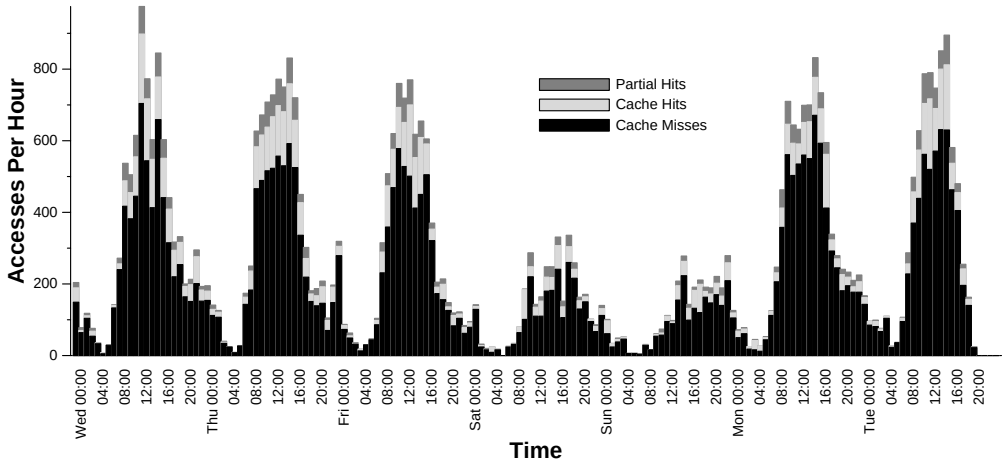


Figure 5.16: Cache accesses: Hits, partial hits, and misses.

up to the entire 7 day trace duration. Notice that reducing the caching window to 5 minutes still yields reasonably high request hit rates (20%). By keeping objects in the cache for only two hours after the last access, we achieve 90% of the maximum possible byte hit rate for this workload while saving significant storage overhead. From this data, we can infer that requests to streaming-media objects that are accessed at least twice have a high degree of temporal locality.

5.6 Stream Merging

Stream merging is a recently developed technique that uses multicast to reduce the bandwidth requirements for delivering on-demand streaming media. The details of stream merging are covered extensively in [Eager et al. 99, Eager et al. 00]. In this section we provide a high level overview of stream merging to motivate our measurements.

Stream merging occurs when a new client requests a stream that is already in transmission. In this case, the server begins sending the client two streams simultaneously: (1) a “patch stream” starting at the beginning of the client’s request, and (2) a multicast stream of the existing transmission in progress. The new client buffers the multicast stream while displaying the patch stream. When the patch stream is exhausted, the client displays the multicast stream from its buffer while it continues to receive and buffer the simultaneous multicast stream ahead of the display. At the merge point,

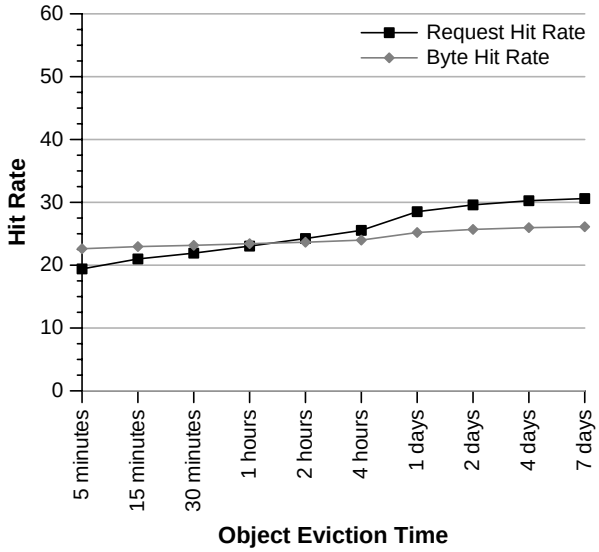


Figure 5.17: Effect of eviction time on cache hit rates.

only one stream, via multicast, is being transmitted to both clients. The cost of stream merging is that clients must be able to receive data faster than the required stream playback rate and must buffer the additional data.

To evaluate the effectiveness of stream merging for our workload, we consider consecutive overlapping accesses to each stream object in our trace and calculate the time it takes to reach the merge point based on [Eager et al. 00]. Given the time of the merge point, we then calculate what percentage of the overlap period occurs after the merge point. This corresponds to the percentage of time that only one stream is being transmitted via multicast to both clients. The results of this analysis are shown as a cumulative distribution in Figure 5.18. Because this stream merging technique is only needed for on-demand streams, live streams are not included in Figure 5.18. This figure shows that stream merging is quite effective for this workload – for more than 50% of the overlapping stream accesses, shared multicast can be used for at least 80% of the overlap period. This result indicates strong temporal locality in our trace, which is consistent with our concurrent sharing and cache simulation results.

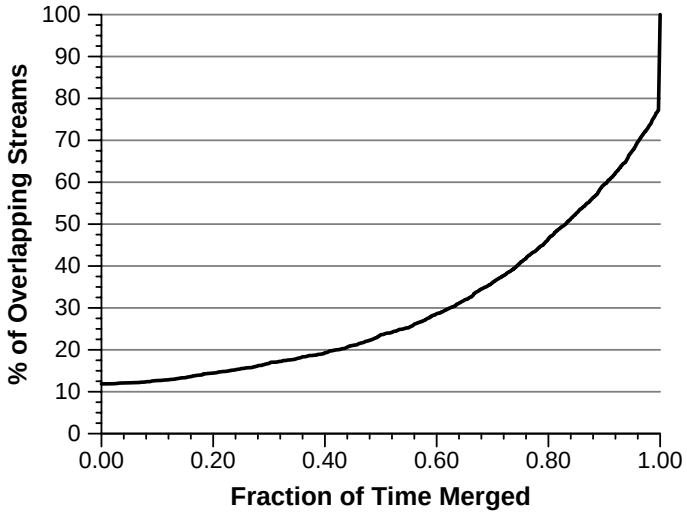


Figure 5.18: Effectiveness of stream merging. This graph shows the cumulative distribution of the time merged for those streams with overlapping accesses.

5.7 Conclusion

We have collected and analyzed a one-week trace of all RTSP client activity originating from a large university. In our analyses, we characterized our streaming multimedia workload and compared it to well-studied HTTP Web workloads in terms of bandwidth utilization, server and object popularity, and sharing patterns. In particular, we examined aspects unique to streaming-media workloads, such as session duration, session bit-rate, temporal locality, and the degree of overlap of multiple requests to the same media object. We also explored the effectiveness of performance optimizations, such as proxy caching and multicast delivery, on streaming-media workloads.

Our results show that:

- Streaming media sessions are much larger than HTTP responses. The median RTSP session size is 400 times larger than the median HTTP response size, and the mean RTSP session size is 175 times larger than the mean HTTP response size.

- *Most* streaming media objects are modest in size ($<$ 1 MB), are encoded at relatively low bit-rates ($<$ 56 Kb/sec), and are short in duration ($<$ 10 mins).

- It appears that current streaming-media workloads benefit less from proxy caching, on average, than traditional HTTP workloads. The α parameter of the Zipf-like popularity distribution for our RTSP workload was 0.47, whereas typical values for HTTP workloads range from 0.6 to 0.8. Furthermore, the byte hit-rate for an ideal RTSP proxy cache with our workload was 26%, compared with a 54% byte hit-rate for an ideal HTTP proxy cache serving the same population measured a year earlier.

- Our streaming-media workload exhibits stronger temporal locality than expected. During peak load periods, between 20% and 40% of all stream accesses are concurrent accesses. This suggests that multicast and stream-merging techniques may prove useful for these workloads.

Our results are fundamentally based upon the workload that we captured and observed. It is clear that usage of streaming media in our environment is still relatively small, and our results could change as the use of streaming media becomes more prevalent. Our one-week trace contains only 40,000 sessions from fewer than 5,000 clients. A one-week trace of Web activity for the same population about a year earlier showed more than 22,000 active clients and more than 80 million Web requests during one week. Shifts in technology use, such as the widespread use of DSL and cable modems, will likely increase the use of streaming media and possibly change underlying session characteristics. As the use of streaming media matures in the Internet environment, further client workload studies will be required to update our understanding of the impact of streaming-media data.

Chapter 6

The Design and Implementation of an Application-Level Internet Tracing System

6.1 Introduction

The ability to measure Web traffic characteristics is important to Web software developers, Web protocol designers, Web content providers, and Internet service providers. Web software developers and protocol designers can learn how their software is used by studying Web traffic characteristics. They can better understand any performance problems that their software exhibits, thereby allowing them to make informed decisions about which components to optimize and how to fine tune any configuration parameters of their software to achieve good performance. Web content providers can learn how to design better Web sites by studying traffic measurements, which provide insight into user access patterns. Internet service providers can learn about the relative importance of network applications in terms of the amount of bandwidth each application consumes.

The task of collecting and analyzing Web workloads is not easy. A number of different techniques are used to collect Web workloads, each of which has strengths and weaknesses. The size and diversity of the Internet makes it impossible to know if the traffic characteristics observed at one location on the Internet will be applicable in other locations. Furthermore, the Internet is a moving target. Rapid growth in Web usage occurs both as new users and new network applications appear. As a consequence, workload characteristics observed at a given point in time may not hold even one year later.

Network tracing is one approach to measuring the characteristics of Web and other network traffic. One of the key advantages of network tracing, as a workload collection technique, is the ability to observe the network behavior of a large number of users without installing a large number of hardware or software components. Network tracing systems offer the ability to observe, analyze, and summarize network traffic. These systems often provide visibility down to the level of individual packets. Tracing systems have proved useful in a wide variety of application domains such as

network security, distributed debugging, network operations, and performance analysis. In this chapter, we present the design and implementation of a tracing system used to collect workloads of two application-level Internet protocols: the HTTP protocol used to transfer Web content and the RTSP protocol used to transfer streaming-media content.

6.1.1 Uses of Network Tracing Systems

For network security, tracing systems are an important component of online intrusion detection systems. They are also used in the response to security incidents; they provide an audit trail of network activity that can be used to understand the attack mechanism for a system that has been recently compromised.

For network operations, tracing systems play an important role in capacity planning. Network managers use them to understand which components of their network are currently overloaded, and also to analyze the rate of traffic growth and upgrade capacity before performance problems arise. Network managers also use tracing systems to discover failures such as routing failures or failed components. Finally, tracing systems provide a method for network operators to implement detailed and accurate billing and accounting systems.

Tracing systems provide a useful debugging tool for distributed application development. The ability to observe application interactions at the network level often leads to the discovery of unexpected communication patterns, unnecessary data transfers, unintended security exposures, and other irregularities.

Performance analysis is perhaps the most compelling use for network tracing tools. Tracing has been used to study low-level network performance issues (such as packet loss, delay and jitter[Paxson 97]), network protocol layer interaction problems[Balakrishnan et al. 98], and aggregate behavior. The key advantage of network tracing for performance analysis is the ability to observe the aggregate traffic from a large number of users.

6.1.2 Contributions

In this chapter, we describe our design, implementation, and deployment experience with a network tracing system explicitly focused on studying application-level Internet protocols. Our system uses

passive network monitoring to observe the Web behavior of a large and diverse group of clients. We installed it at the University of Washington Internet border, and we used it to observe all Internet traffic that flows through the border.

The tracing system was in use at the UW Internet border for approximately three years. We monitored HTTP workloads from the Summer of 1998 through the Spring of 2000, and we monitored RTSP workloads from January of 2000 through the Spring of 2001. During the deployment period, the HTTP traffic load grew by a factor of three, and the tracing system scaled to keep up with the load increases with negligible packet loss.

Our system supports high-speed passive network monitoring of *multiple* network interfaces on the same machine. We also support asymmetric routing paths – a relatively common property of Internet traffic flows. Our system performs on-the-fly reconstruction of TCP connections. This tracing system takes an extremely conservative approach to protecting users privacy. Finally, we use a novel technique to preserve geographic locality for the machines being traced while still maintaining anonymity.

The remainder of this chapter is organized as follows. We begin with an evaluation of alternative Web workload collection techniques in Section 6.2. In Section 6.3 we describe the hardware architecture of our trace collection system. Section 6.4 provides a detailed look at trace collection software architecture. In Section 6.5 we describe the trace analysis tool-set, and Section 6.6 examines the performance of our tracing system.

6.2 Alternative Workload Collection Approaches

Network tracing is one of many techniques used to study the characteristics of Internet workloads. Two other commonly used approaches are application-level logging and synthetic traffic generation.

One advantage of the application logging approach is that many applications, including most Web servers and proxies, provide built-in support for logging a certain amount of detail about the Web traffic they handle. Enabling logging for these applications is usually just a matter of changing an item in a server configuration file.

When the source code for Web components is freely available, it is relatively easy to modify the logging code to collect the desired information. A number of popular Web components fall into

this category, such as the Squid proxy cache, the Apache Web server, and recent versions of the Netscape Web browser.

There is a final advantage of the application logging approach. For certain types of studies, one has access to information that is relevant to the workload, but is not encoded in the transmitted data, and may therefore be unavailable to another approach such as network tracing or synthetic traffic generation. For example, in a trace of HTTP traffic one might want to know which URLs are static HTML content and which URLs are program-generated HTML content. There is no information contained in an HTTP Web response that would allow a network tracing program to determine how to answer this question, yet it would be relatively straightforward to write logging code within a Web server to record this information.

There are also numerous disadvantages to the application logging approach. When using an off-the-shelf product, the source code is often unavailable which makes instrumentation difficult. The default level of detail collected in the logs is often inadequate for performing the necessary analysis. For example, the access logs generated by a Microsoft Web proxy server include a flag to indicate the occurrence of certain HTTP headers (namely the "Last-Modified" and "Expires" headers), but those logs do not include the actual time values for those headers, which prevents one from performing an analysis of Web cache-consistency strategies using these logs. Another disadvantage is that the performance overhead of application logging may be prohibitive, especially when the level of detail required is extensive.

Finally, the most serious problems for the application logging approach relate to deployment. For production services, the deployment problems arise from the risk of modifying those services. In general, system administrators are extremely reluctant to make any changes to production services that do not improve the performance, correctness, or availability of the service. Modifying the source code of an application to collect workload information does not directly help any of those three properties, and furthermore it carries the risk of unintentionally introducing bugs into previously working software.

Synthetic Web-traffic generation has been used for performance characterization by a number of previous research efforts [Worrell 94, Wills et al. 99b, Wills et al. 99a, Krishnamurthy et al. 00]. This approach has also become popular in the commercial arena. Keynote systems [Keynote 01] measures end-to-end Web performance by placing measurement hosts in major metropolitan areas

across the world.

The most significant advantage of generating synthetic Web request traffic is the amount of control that it allows. For instance, it allows a controlled set of sites to be studied, which makes certain types of comparisons easier. In [Krishnamurthy et al. 00], the authors identified 700 popular Web sites that supported both HTTP 1.0 and 1.1. They then generated synthetic Web requests to perform page downloads using four different download policies implemented by their client software, which allowed them to make a direct comparison between the different download strategies.

Another advantage of synthetic Web requests over both application logging and network tracing is the potential to use large amounts of computation or storage to perform an experiment. Consider the case where one wants to study Web document rate of change. To determine when a change occurs or to determine the extent of changes within a document, one wants to save the entire document contents of each document access (or a hash of the document contents). Using the application logging approach, one would need to modify a proxy cache or Web server to save a copy of each service response or to calculate a hash value for each service response. This would require a significant amount of compute and/or storage resources at the machines providing the service, which would in turn affect the users of that service. Using a network tracing system, the additional overhead of saving the document contents or calculating a hash of the document contents could lead to difficulties during a burst of traffic. Using the approach of synthetic traffic generation, requests may be generated at a rate that the CPU and storage resources of that system can handle easily.

The final advantage of synthetic request generation is that it permits the analysis of sites using encryption techniques such as SSL [Freier et al. 96] or IPsec [Kent et al. 98]. Network monitoring does not permit any visibility of SSL or IPsec traffic. Furthermore, the most common application logging approach to collecting Web traces is at a proxy cache, and SSL traffic that goes through a proxy cache cannot be observed and recorded in the proxy logs.

The principal disadvantage of generating synthetic requests is that the results are not weighted by the frequencies with which real users access the documents in question. This can sometimes make results difficult to interpret. For instance, [Wills et al. 99b] studied how often Web documents were uncachable by requesting the top-level home pages from a set of 100 popular Web sites listed at “http://www.100hot.com/”. They calculated the percentage of uncachable documents in their test set, but one cannot directly compare that percentage with the percentage of uncachable documents

in a trace of real Web users. The results from a real trace will be affected by the actual document popularity distribution whereas the results from a synthetic study will not.

Another potential disadvantage of generating synthetic requests is the possibility that the synthetic traffic differs in some unexpected way from the real traffic. For example, [Wills et al. 99b] investigated whether or not document content changes when Web requests to that content contain different cookies. They generated multiple requests, each of which contained a different assigned cookie. This strategy assumed that the document content might change without any specific user action. Many sites that use cookies for customization require additional user actions to actually customize the content, which their methodology will not do. Synthetic Web requests often use a custom HTTP client rather than a full Web browser. This approach is dangerous because differences in the HTTP implementation can also lead to unexpected results.

6.3 Hardware Configuration

Figure 6.1 illustrates the UW network border configuration during the time when the tracing system was operational. Our tracing system used *passive network monitoring* to observe all packets that flowed through the four campus network switches. The two border routers were configured so that during normal operation, one handled all inbound traffic and the other handled outbound traffic. Each of the four campus switches was configured to enable port mirroring, and the mirror port was configured to only support unidirectional traffic. In other words, we avoided any operational risk to the campus network environment by ensuring that any network traffic generated by our tracing machine would be discarded. Traffic from the campus subnet routers was load balanced across the four campus subnet switches. Furthermore, the routing of traffic across these switches was asymmetric. This meant that the inbound and outbound directions of a single TCP connection often traversed different network switches.

For the first two years, our traces were collected by a single DEC Alpha workstation with four 100 Mb/s Ethernet interfaces. The CPU of this machine was a 500MHz Alpha 21164, and it contained 384 MB of RAM. The operating system we ran on this machine was Digital UNIX V4.0D. Our Ethernet network interface cards were built with the DEC PCI 21140 Ethernet chip-set. As the traffic load grew, we moved to a dual host configuration. For the final year, our HTTP traces were

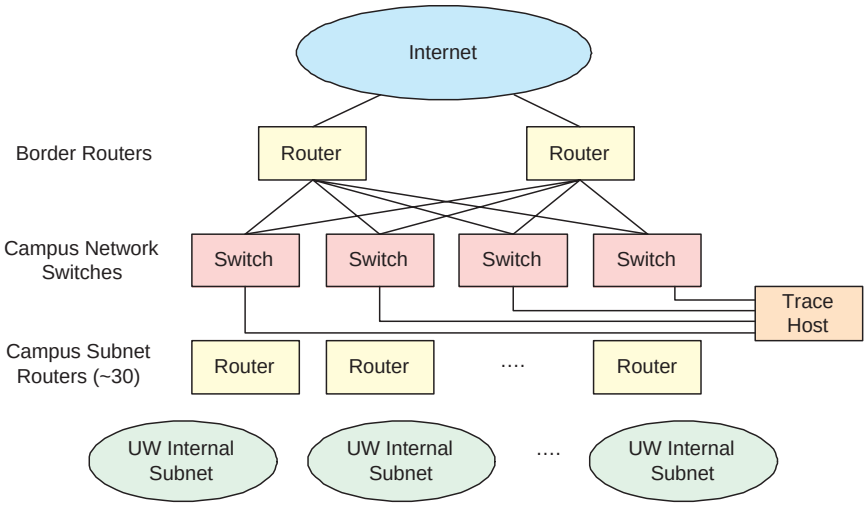


Figure 6.1: University of Washington network border configuration, including the installation of our tracing system.

collected by two Alpha workstations. The second workstation used a 500MHz Alpha 21264 CPU, and it contained 512 MB of RAM. For this configuration, both workstations ran the Compaq Tru64 UNIX V4.0F operating system. Each machine had two Ethernet interfaces connected to the campus switches, and a third Ethernet interface was used for inter-machine communication.

6.4 Trace Collection Software

Figure 6.2 provides a high-level view of the network trace collection software architecture. The left-most box shows an expanded view of the primary kernel components that the network tracer uses, and the right-most box shows an expanded view of the primary modules within the user-level network tracing application. One design principle we use to guide our development strategy is to avoid making kernel modifications as much as possible. By working mostly at user-level, we reduce development time by avoiding the typical debugging problems that arise from working inside the kernel. We pay a performance penalty by implementing most of the tracing functionality at user-level because the filtration process to eliminate uninteresting packets occurs at user-level.

The user-level network tracing application, *httpmon*, uses an event-driven concurrency model. We choose this structure for a number of reasons. First, we avoid the locking overhead and race

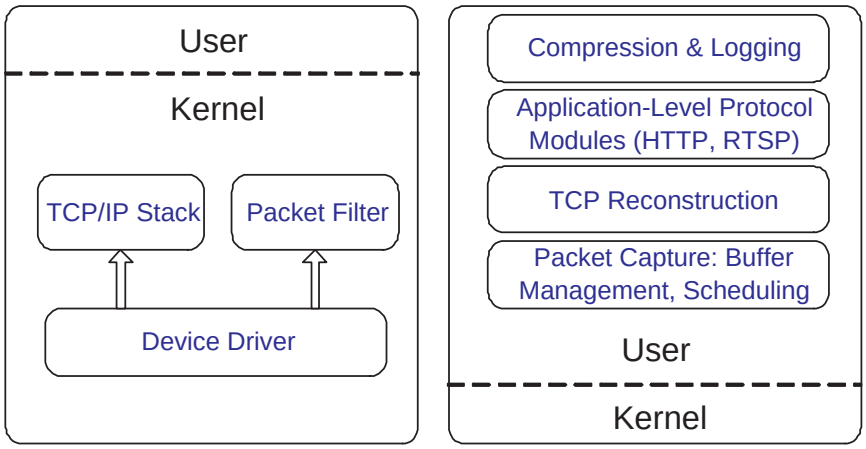


Figure 6.2: Software architecture of the network trace collection system.

conditions that are often a problem when implementing multi-threaded applications. Second, we use the gdb debugger and gcc compiler suite. On the Digital UNIX platform, gdb is unable to do post-mortem analysis when applications are linked with the Digital UNIX POSIX threads library. Also, gdb support for live debugging of multi-threaded applications is unreliable on the Digital UNIX platform. Lastly, we want explicit control over scheduling within the *httpmon* process which would not be available to us using POSIX threads. Due to our event-driven concurrency model, we must ensure that all system I/O operations involve non-blocking system calls.

Because the tracing system was deployed over a long period of time, we went through a number of configurations, each of which placed slightly different demands on the software. Therefore, in the following sections when describing the software implementation we will refer to three specific configurations:

- **Original HTTP configuration:** In the original HTTP configuration, we used a single Alpha 21164 workstation to monitor all four network switches on the same machine. We collected information about all incoming and outgoing HTTP transfers.

- **Dual-host HTTP configuration:** For the dual-host HTTP configuration, we used two Alpha workstations (one with an Alpha 21164 CPU and the other with the Alpha 21264 CPU) to collect our HTTP traces. Each machine had two network interfaces used for packet mon-

itoring and a third network interface used for inter-machine communication. The primary reason for connecting the two machines was to synchronize the clocks so that the timing information in the trace would be useful. We used NTP to synchronize the clocks of the two workstations [Mills 91].

- **RTSP configuration:** For the RTSP configuration, we used the Alpha 21264 workstation to collect our RTSP traces. The software was configured only to monitor RTSP and not HTTP. For reasons discussed below, our software did not support monitoring RTSP with a dual-host configuration.

In the sections that follow, we provide an overview of each major component of the network tracing system. We begin with an overview of our design to protect the privacy of individuals using the campus network because these privacy constraints have a significant impact on the overall design of the tracing system.

6.4.1 Privacy Protection

Our system goes to great lengths to protect the privacy of students, faculty, and staff using the UW campus networks. Our primary goal is to ensure that the traces we collect cannot be used to identify the behavior of individual *internal* users. To accomplish this goal, we enforce three constraints on our system design. First, we ensure that all potentially sensitive information collected in our trace is anonymized. *This means that we do not know which client accessed a Web page, which server was accessed, or even the name of the URL that was accessed.* Second, we erase a few of the low-order bits for all internal client addresses in our trace before the anonymization step occurs. Finally, we ensure that no raw data ever reaches disk storage in an un-anonymized form. We protect the monitoring host by disconnecting it from all networks other than the passive one-way connections to the switches.

We use a keyed version of the MD5 message digest algorithm to anonymize strings such as the name of the URL, the server name, and the referrer header [Rivest 92]. We use a different technique to anonymize all client and server IP addresses in our trace. Previous tracing studies anonymized the entire IP address, so that two clients from the same local network would have their addresses

mapped to two random addresses that have no detectable relation to each other once anonymized. In contrast, our goal is to retain some locality information in the anonymized addresses. We would like to be able to associate addresses from the same local network topology (e.g. campus department) without compromising the anonymization scheme.

IP Address Anonymization

We use slightly different approaches for client and server IP addresses. For clients, we partition the IP address into two parts: a network part and a host part, and we anonymize each part separately. Hosts on the same network will therefore have the same anonymized network part, but because that network part is anonymized it will not be known exactly which network the host is on. Furthermore, to absolutely prevent the possibility that a client address can be matched up against its anonymized version, not only will the host part of the address be anonymized, but the low-order bits of the host part will be erased before the anonymization step. With this anonymization scheme, even in the unlikely event that the key to the hash function is compromised (e.g. discovered, reverse engineered, or involuntarily revealed by a court order), no client address from a machine on campus can be matched one-to-one with absolute certainty to an anonymized client address in the trace. To determine the boundary between the network and host parts of the client address, we precompute a table that stores this information for all internal UW networks.

For server addresses, we know nothing about the breakdown of IP addresses into network and host parts, so instead we anonymize each octet in a server IP address separately. Classless routing for the Internet prevents us from making too many assumptions about addresses that match using some number of prefix bytes, but we can still make some useful assumptions. For our purposes, two servers are near to each other if they share most or all of the Internet path between them and UW. Given this definition, we assume that two servers are close to each other if their first three octets match. Even if more than 24 bits are used for the network address of a server, it is very likely that servers on two different subnets of such a network will share a common path through the Internet from UW.

Key and Data Management

We type the key for the MD5 hash function into the console of the tracing system by hand at the start of each experiment. We generate the key with the PGP key generator which is seeded by sampling keystroke arrival times [PGP 01]. We record this key on paper and not on any non-volatile electronic storage medium. After a tracing experiment is finished, we erase the key from system memory.

To further ensure that our tracing system does not compromise the privacy of the campus network users, we impose the constraint that no raw packet data should ever reach stable storage. This constraint has a significant effect on the overall design of the network tracing application. In particular, it precludes using the approach taken by the BLT system [Feldmann 00], where the high-priority task is simply copying packets from the network interface to a raid storage array with an intermediate step of IP address anonymization. In our system, packet analysis, TCP connection reconstruction, data anonymization, and data compression are done in RAM as an online computation, and the only data written to non-volatile storage is HTTP header information that has been processed and anonymized.

To debug the tracing system, we need to record some raw trace data to reproduce errors generated by the live packet data and to verify that our fixes are correct. Because of the constraint that no raw packet data should be saved on persistent non-volatile storage, we use a memory file system to store the error logs. This has the disadvantage that if the system crashes and reboots the error logs are permanently lost. We also isolate the information in our error logs so that when a sensitive field such as a URL is written to the log, we ensure that no other information (such as an IP address or another header field) is logged about that same connection.

6.4.2 Kernel Modifications

As mentioned earlier, our goal was to make as few kernel modifications as possible. For the original HTTP configuration, we made two minor kernel modifications. Our first modification was to increase the size of the receive descriptor ring in the Ethernet device driver. This driver maintains a ring of memory descriptors used by the Ethernet interface to place incoming packets into system memory. Each descriptor describes a segment of contiguous physical memory, and each memory segment is used as the target of a DMA transfer from the network interface memory to system mem-

ory. We increased the number of receive ring descriptors from 16 for each network interface to 256 per interface. This allows the device to handle bursts of packets when the system is too busy to process a receive interrupt generated by a particular network interface in a timely manner.

Our second modification was to change two buffer management related constants in the kernel packet-filter module. The first constant controls the maximum number of packets that can be queued for a single packet-filter, and the second controls the maximum total number of packets that can be queued across all the packet-filters. The goal of increasing these values is to handle the delay between reads by the monitoring process.

For the RTSP configuration, we used a single machine to collect our traces. We chose to use a single machine for monitoring RTSP because the RTSP protocol module needs information from both incoming and outgoing traffic, and with a dual host configuration it is possible that the incoming traffic is monitored by a different machine than the outgoing traffic. Rather than implementing support for fine-grained sharing of information across the two monitoring hosts, we chose instead to modify the kernel to eliminate all HTTP port 80 traffic at the device driver level. This reduced the load to a point where a single machine could monitor RTSP traffic on all four network interfaces without dropping packets. To support this optional HTTP filtering, we added a device ioctl to the driver so that the monitoring application can enable the filter at startup and disable the filter upon exiting.

6.4.3 Packet Capture: Buffer Management and Scheduling

The packet capture module of our user-level network tracing application is the layer that extracts packets from the kernel, buffers these packets, and schedules the packet processing tasks. The interface to this module is based on the packet capture library, *libpcap*, which is the library component of the popular tcpdump network tracing tool [Tcpdump 01]. The implementation of this module diverges from the original libpcap in a number of ways. Because the original libpcap implementation was designed for monitoring a single network interface, its buffer management and scheduling strategies are inadequate when monitoring multiple network interfaces simultaneously. For instance, the *pcap_read* routine provides a buffer to the kernel packet-filter and then blocks until the packet-filter has filled the buffer. When monitoring multiple network interfaces, this blocking model can

cause packet loss on the other interfaces. Furthermore, the libpcap scheduling model requires that an application process all packets placed into the buffer before it can call the read routine again to extract more packets from the kernel. This can also lead to long delays between read operations which in turn causes packet loss.

In our packet capture module, we allocate a large (4 MB) ring buffer for every network interface being monitored. All read operations from the kernel packet-filter into the ring buffers are non-blocking, and we use the *select* system call to detect when new packets have arrived at a given interface. When the incoming packet rate is high enough that all interfaces have newly arrived packets, we perform read operations on each network interface in a round-robin manner. We structure the packet capture module so that extracting packets from the kernel has a higher priority than processing packets that are already in the ring buffers. During normal operation we process packets from the ring buffers in a round-robin manner. When the amount of data stored in one of the ring buffers exceeds a threshold (currently 1/6th of the total buffer size), we raise the priority of processing packets from that ring buffer. The key to supporting high speed packet monitoring across multiple network interfaces without packet loss is to ensure that the delay between successive read operations to the same kernel packet-filter never gets too large. Section 6.6 describes the amount of packet loss we observed when collecting traces at the University of Washington.

6.4.4 TCP Connection Reconstruction

When the packet capture module schedules a packet to be processed, the first step is performed by the TCP connection reconstruction module. This module identifies all packets that contain TCP segments and then reconstructs the TCP connection that those segments belong to. Next, it classifies each TCP connection using a protocol-specific predicate. For HTTP, this predicate examines the contents of the first data segment in each connection to decide whether or not the connection is an HTTP connection. This technique allows us to see all HTTP traffic and not just the HTTP traffic that happens to use port 80. Once a connection has been classified as HTTP, we monitor further segments along that connection so that we can locate all the HTTP headers when persistent connections are used. For all TCP connections that we determine are not HTTP, the segments that follow are simply discarded.

In order to reconstruct all TCP connections that flow through our tracing host, this module maintains a data structure called the *tcp_state* table where information is stored about the currently active TCP connections. In many respects, the *tcp_state* table resembles the table of TCP protocol control blocks that is maintained by the traditional BSD implementation of the TCP/IP protocol stack. The *tcp_state* table is implemented as a hash table whose key contains the source address, the source port number, the destination addresses, and the destination port number. Due to the asymmetric routing architecture at UW, we actually analyze the inbound and outbound halves of each TCP connection independently. Therefore, entries in the *tcp_state* table correspond to unidirectional flows and each TCP connection will contribute two entries into the table.

One of the challenges of properly reconstructing TCP connections is dealing with out-of-order and retransmitted TCP segments. We want to ensure that retransmissions do not lead to duplicate entries in our trace log of HTTP requests and responses. If a TCP segment contains an HTTP request, and if that segment is retransmitted due to a packet loss in the portion of the network between the tracing machine and the destination, then our TCP layer should detect that retransmission and ensure that the HTTP analysis module only processes that request once. Due to the large number of active TCP connections and the relatively limited amount of memory on our tracing machines, performing TCP segment reassembly in the traditional manner is not possible. A traditional TCP implementation performs reassembly by storing all out-of-order packets while waiting for the preceding packets to arrive. Instead, our tracing software handles this problem by maintaining a sequence-map data structure that records for every connection a map of all contiguous ranges of the TCP sequence space that have been processed by the HTTP analysis module.

Garbage Collection

The final major component of the TCP reconstruction layer is the garbage collector. Entries are added to the *tcp_state* table each time a TCP SYN segment is processed. The garbage collector component is responsible for deleting entries from the *tcp_state* table. Deciding when to delete entries from this table is somewhat more complex than adding entries to the table. When the TCP reconstruction module sees a FIN or a RST segment, we do not immediately delete the corresponding entry from the *tcp_state* table for the following reason. Packets may arrive at the trace machine

out-of-order and the end-host may even retransmit a FIN segment. When a machine terminates a TCP connection using a RST segment, the opposite direction of that TCP connection typically does not see any explicit traffic that indicates the connection is terminated. Because we are monitoring each direction of a TCP connection independently, this means we need another mechanism to garbage collect these entries from the table.

Our garbage collector implements the following policy. Every four seconds, the garbage collector runs and traverses a fixed number of buckets in the hash table, although each bucket will have a variable number of entries depending on how full the table is. This incremental strategy ensures that garbage collector does not run for too long at any one time, which would eventually lead to packet loss. For each connection where we see a FIN or a RST segment and the connection is idle for at least three minutes, we delete that entry from the table. For each connection where we see no FIN or RST segment, we require that the connection be idle for at least fifteen minutes before we delete that entry from the table. One consequence of this policy is that if any persistent connections have very long idle periods, then these connections are garbage collected and any requests or responses that follow the idle period are not included in our trace.

6.4.5 Application-Level Protocol Modules

The application-level protocol modules process data passed in from the TCP reconstruction layer for those TCP connections that are classified as the correct type. In our system, we implement two application-level protocol modules, one for HTTP and another for RTSP. Both the HTTP and RTSP modules perform similar tasks even though the details differ. Both modules perform the task of identifying and then parsing the protocol headers. Because both protocols are based on the MIME message format, the parsing tasks for both protocols are very similar. Once the headers are parsed, we extract the relevant fields to be saved to the trace log, and we then anonymize any of these fields that contain sensitive information. The extracted fields are converted to a compact binary representation, and this data structure is then passed to the logging and compression module. To demonstrate the kind of information stored in these data structures, Figure 6.3 shows a commented version of the data structure used to record information about an HTTP response.

The task of parsing is made significantly more complex by the wide variety of HTTP implemen-

```
struct http_response_entry {
    int64                conn_id;       // unique identifier for each TCP connection
    struct timeval       syn_time;      // time when the initial SYN segment arrived
    struct timeval       hdr_time;      // time when this header segment arrived
    struct timeval       ld_time;       // time when the last data segment arrived
                                        // (for persistent connections)
    unsigned short       hdr_len;       // header length
    seq_t                hdr_seq;       // starting sequence number of this header
    interface_t          interface;     // network interface this header arrived on
    node_addr            src_addr;      // source IP address, port, and organization-id
    node_addr            dest_addr;     // dest IP address, port, and organization-id
    unsigned short       status;        // HTTP response code
    struct http_version  http_version;  // HTTP version number
    int64                size;          // Content-Length header
    time_t               date;          // Date header
    content_type_t       type;          // Content-Type header
    content_encoding_t   encoding;      // Content-Encoding header
    char *               server_agent;  // Server-Agent header
    time_t               last_modified; // Last-Modified header
    time_t               expires;       // Expires header
    hashval_t            etag;          // Etag header
    struct cache_info    cache_control; // Cache-control header
    unsigned short       flags;         // Flags to indicate other conditions
                                        // in the HTTP headers
};
```

Figure 6.3: The data structure used to record information about an HTTP response.

tations that don't quite follow the rules. For example, the protocol specification is very clear as to the exact character sequence that specifies the boundary between HTTP headers and the message body. In practice, our tracing software would fail to detected the boundary approximately 25% of the time if it only used the exact character sequence specified by the RFC. Another example is the misspelling of important header names. Our parser accepts nine different spelling variants of the "Content-Length" header, which we observed often enough to code into our parser. RTSP suffers from this problem to a lesser extent than HTTP because there are not as many diverse implementations.

The RTSP parsing is somewhat more complex than the HTTP code because of a wider variety of message types and because of the fact that the protocol is stateful. Our RTSP parsing module maintains a table that models the state machine for each client session. Furthermore, it deals with control messages that cross TCP segment boundaries, and it deals with extracting the control messages from interleaved connections. Interleaved connections are those where the control traffic and the streaming-media data traffic are sent over the same TCP connection. Finally, there is additional

complexity in the RTSP parser because it has to interact with the lower layers of the tracing software to monitor the out-of-band traffic associated with an RTSP control connection. The parser extracts UDP port numbers from the RTSP control connection, and then the lower layer packet processing code counts the number of bytes transmitted on those UDP channels.

6.4.6 Logging and Compression

The logging and compression layer takes input records passed to it from the application-layer protocol modules. We append these records to a 128 KB buffer, and once that buffer fills we compress that chunk of data and write the compressed chunk to disk. We use the LZO library for data compression because the LZO library is optimized to provide fast compression and decompression [Oberhumer 02]. We use the Digital Unix asynchronous I/O interfaces to ensure that write operations do not block in the kernel. Rather that writing the trace as one very large file, we write the trace in 40 MB segments. This makes the trace easier to manage after it has been collected. The logging layer hides this complexity from the rest of the tracing system as well as from the trace analysis tools.

6.4.7 Denial-of-Service Attacks

Shortly after our initial deployment of the tracing system, we discovered that our system was vulnerable to TCP SYN attacks, a relatively common form of denial-of-service attack on the UW network. Whenever a SYN attack traversed the UW campus network border, it caused the *tcp_state* table maintained by our tracing software to use up a very large amount of memory. After a short period of time, the system ran out of memory at which point the tracing software ceased to function. In order to collect the traces, we had to develop heuristics to detect and eliminate the SYN segments in the attack, while not losing any of the "real" SYN segments.

To detect the occurrence of an attack, on a two-second periodic interval we calculate the ratio of the number of SYN packets that arrive during the most recent interval to the number of connections that receive their first data segment during that interval. When that ratio exceeds four to one, and when the absolute number of SYN segments that arrive during the interval exceeds 2,000, then we initiate the code that attempts to determine the target of the attack. We observe that all of the

SYN attacks that flow through our tracing system have a wide variety of spoofed IP addresses as the source address of the SYN segment, but the target or destination address of the attack is just a single IP address. The code that determines the attack target samples a set of source and destination address pairs from recent SYN segments looking for a common target IP address. When it finds such an address, it records that address and then enables the SYN filtering code.

Our initial approach to filtering was simply to eliminate the bad SYN packets in the TCP reconstruction layer. We discovered that SYN attacks caused an additional problem for our tracing code beyond just wasting lots of memory by filling up the *tcp_state* table: the overall rate of incoming packets during a SYN attack greatly exceeded the rate of incoming packets at any other time during trace collection. The rate was so high that the user-level ring buffers filled up and the system began to drop packets. To fix this problem, we built a fast path inside the packet capture module that eliminated the bogus SYN packets as the first step in packet processing. This fix worked well enough to eliminate the problem, though if it hadn't been effective we could have gone one step further and prevented the packets from ever being delivered to user-level by installing a new kernel packet-filter that eliminated all packets directed to the attacked IP address.

The final aspect of our SYN attack handling involves deciding when the attack is over and when to clean up the filters. We do this by continuing to look at the ratio of SYN segments to real TCP connections, and we wait for a full two minutes after the attack appears to be over before cleaning up the filters. We implement this waiting period because we often observe short periods during a SYN attack where the rate goes to zero and then starts back up again.

6.5 Trace Analysis Software

The trace analysis software is structured as support routines for a set of statistics collection modules (SCM). The SCMs can be run either separately or in parallel to analyze properties of the trace. Examples of SCMs include a document cachability analyzer, a response bandwidth analyzer, and a rate of change analyzer. Overall we implemented more than 90 different SCMs. The entire trace analysis system is approximately 36,000 lines of code. In this section, we describe the support routines that ease the development task of building new SCMs and that improve the performance of SCM processing.

The lowest level support routines implement log decompression and merging of the log segment files. The output of those steps is a sequence of individual records from the log files. We take that sequence of records and then match up requests and responses. The matching process for an HTTP persistent connection creates a data structure that consists of one array storing all the requests records, another array storing all the response records, and two close records that store the termination information for each half of the TCP connection. From this data structure, an additional step matches requests and responses within the persistent connection and creates an *http_xaction* data structure. An *http_xaction* is a single matched request/response pair with the request and response close records.

The matching code consumes a significant amount of CPU and memory resources when processing a large trace. One consequence of this is that the memory used for matching reduces the maximum amount of memory that can be used by one or more SCMs. A number of the SCMs we implemented are very memory intensive (e.g., an ideal cache simulator with no replacement). To compensate for this problem, we implemented a trace sorting pre-processor. The trace sorter uses the matching code described above, and in addition it sorts the resulting *http_xaction* records on any of the time fields in either the request or the response. The output of the trace sorter is a new set of log segments that contain already matched and sorted records. This sorted log can now be used to run the SCMs, and the memory and CPU overhead of the matching code is eliminated. In addition to the trace sorting, we implemented another pre-processing step we call splitting. The motivation for splitting is that some SCMs consume more memory than the 1 GB of RAM that our analysis machine contains. We observe that for all of memory intensive SCMs that we created, the memory overhead scales with the number of objects in the trace. Therefore, the splitting code divides up the object space into N evenly sized chunks. Each SCM can now be run in N passes where each pass runs over a smaller split trace. We run a final result merge step after all the passes are complete that reads in the intermediate results from each pass, merges the data, and writes out the final results. Note that the result merging code is SCM specific.

The primary trace analysis program contains a command-line parser and a set of three very large switch statements. The command-line parser is used to determine the location of the trace files, any constraints that the user places on which portion of the trace to analyze, and the names of one or more SCMs that the user wants to run. The three switch statements are use to invoke routines within

those SCMs that are specified on the command-line. The first switch statement is used to call out to the SCM initialization routines. The second switch statement is a component of the main loop that takes every matched *http_xaction* and calls the processing routines for each selected SCM. The final switch statement calls the result generation routine within every enabled SCM.

Another aspect of the analysis infrastructure is that we provide a set of support routines used by many of the SCMs to collect and analyze information. We provide a set of histogram implementations that support counting occurrences of integers, addresses, arbitrary strings, and time-series data. Our histogram implementations provide result generation routines that allow the output of histograms to be generated in either standard form or in the form of a cumulative distribution function. For the integer histograms, the creator controls the granularity of the histogram by specifying the bucket size as well as the minimum and maximum values. We also provide the standard statistical routines that calculate the mean and standard-deviation for a data set. We include support for reporting the median value of a data set using the integer histograms.

6.6 Performance

In this section, we characterize the performance of our network tracing system. We focus exclusively on the performance of HTTP monitoring because the RTSP traffic rates are not large enough to cause significant performance problems. We begin by characterizing the traffic load on the system during the highest-load one-week HTTP trace that we collected, and then we characterize the performance of the *httpmon* system during trace collection.

The one-week HTTP trace with the highest traffic load was collected from May 16th, 2000 through May 23rd, 2000. We used the dual-host HTTP configuration for trace collection during that time period. Figures 6.4, 6.5, and 6.6 characterize the network load on the tracing system during that time period. Figure 6.4 shows the packet arrival rate as measured by the network tracing software. Figure 6.5 shows network load in terms of bandwidth rather than packets per second. Figure 6.6 shows the number of entries in the TCP state table during the trace collection period. Each open TCP connection contributes two entries to the TCP state table, one for each direction of the TCP connection. The granularity of measurement by the tracing software was every two seconds, and to improve readability of the graphs we performed a post-processing step on the data to show the data

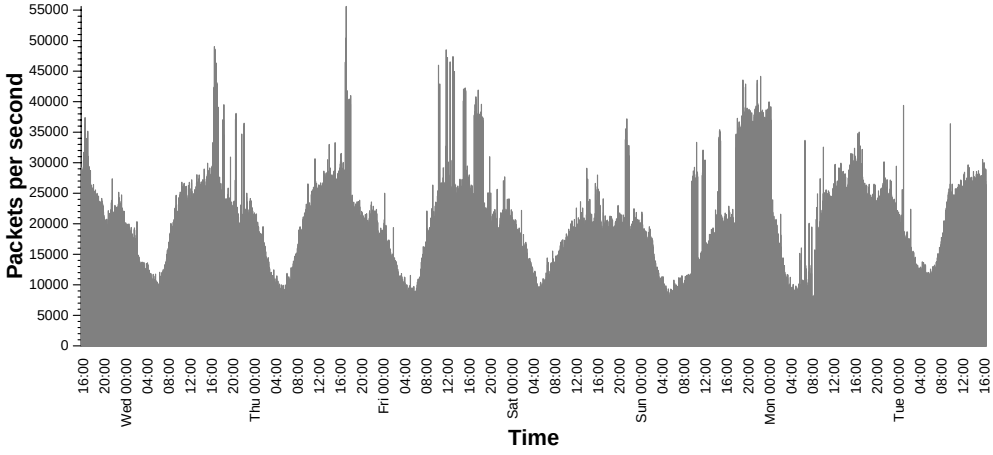


Figure 6.4: Total number of packets per second over time.

in per second granularity. The peak arrival rate during the entire trace period was just over 55,000 packets per second. The peak bandwidth during the trace period was 21.9 MBytes per second (or 175 MBits per second). This figure shows a lengthy burst of packet load lasting more than four hours during the Sunday evening time period, with a sharp start and end to the burst. This burst of load is a denial of service SYN attack and most of the packets are arriving on a single network interface. Because the SYN packets are very small, this burst of load is not easily visible in the bandwidth graph.

Table 6.1 shows the overhead of different components of the system during trace processing, as measured by the DCPI continuous profiling tool [Anderson et al. 97]. DCPI proved to be very useful during the initial development stages of *httpmon* because in many cases our intuition was wrong about which routines needed to be optimized in order to avoid dropping packets. DCPI allowed us to quickly focus on optimizing those routines which were actually consuming a large portion of the CPU. We collected the measurements in this table on Sunday June 11th 2000 during a three hour mid-afternoon period, rather than during the collection of a full one-week trace.

The left-most column of Table 6.1 shows the percentage of time spent in each image while collecting a trace. DCPI defines an image to be an executable, a shared library, or the kernel. Because DCPI groups time spent in the idle_thread along with the rest of the kernel activity, we chose to separate out time spent in the idle_thread for clarity. This portion of the table shows that

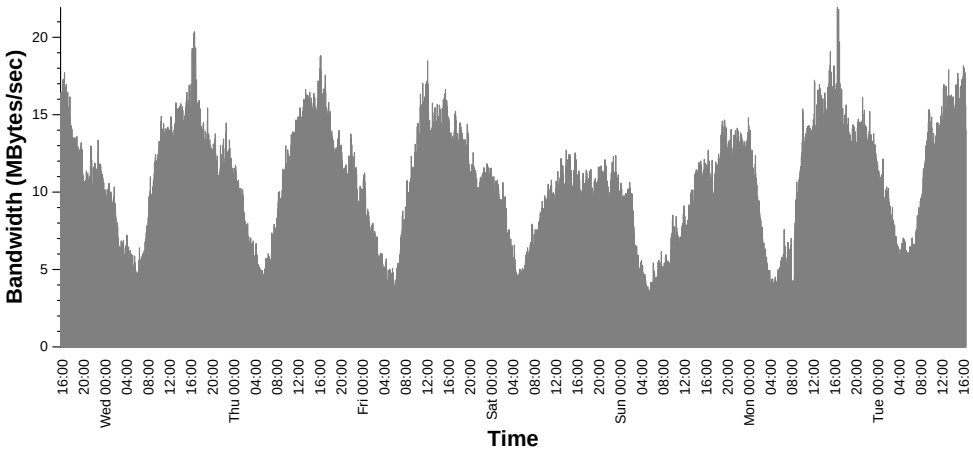


Figure 6.5: Total bandwidth over time (in MBytes/sec).

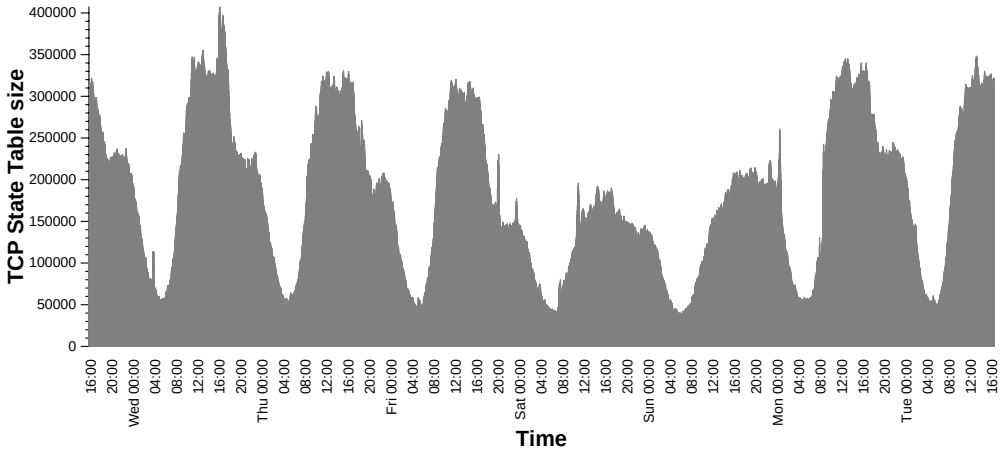


Figure 6.6: Size of the TCP state table over time. Because each direction of a TCP connection is analyzed separately, each open TCP connection contributes two entries to the table.

Table 6.1: DCPI Cycle Count Measurements.

Image	% of Cycles
/vmunix	61.08%
idle_thread	23.69%
httpmon	11.37%
libc.so	3.41%
libm.so	0.33%
dcpid	0.12%

/vmunix Top 10	% of Cycles
bcopy	12.37%
gh_zero_memory	12.16%
vm_page_tester	11.41%
read_io_port	5.35%
Pfilt_read	3.65%
Pfilt_select	2.97%
spec_select	2.95%
free	2.90%
malloc	2.69%
_XentInt	2.53%

httpmon Top 10	% of Cycles
pcap_multi_read_loop	14.42%
0x120015a78	9.51%
process_one_packet	7.64%
0x1200153a0	6.01%
0x120015638	5.41%
handle_ip	5.09%
0x120019d78	4.35%
update_state	4.04%
my_strnstr	3.16%
packet_handler	3.10%

more than 60% of the CPU time during trace collection is spent in the kernel. Approximately 15% of the CPU cycles are spent in the user-level *httpmon* application and the shared libraries that it uses.

The middle column of Table 6.1 shows the percentage of time consumed by the ten most frequently used subroutines in the kernel. Here we see that much of the kernel overhead is due to initialization and copying of memory. One reason for the amount of data copying in the kernel is that the Ethernet device driver we are using performs an extra copy of every received packet so that the TCP/IP headers are aligned on a longword boundary. This is necessary because the DMA engine on these cards can only transfer to a longword-aligned target address. The size of the Ethernet header is 14 bytes so if the beginning of the packet is longword-aligned then the beginning of the TCP/IP header cannot be longword-aligned.

The right-most column of Table 6.1 shows the percentage of time consumed by the ten most frequently used subroutines that are part of the *httpmon* process. From this data we see that most of the overhead within the *httpmon* process comes from the packet capture and TCP reconstruction modules. The subroutine named my_strnstr is the only one of the top ten routines that is a component of the HTTP analysis module. The routine with the largest CPU overhead, pcap_multi_read_loop, is the main loop of the packet capture module that reads data from the kernel packet-filters and schedules the packet processing.

In addition to the DCPI measurements, we also directly instrument the *httpmon* application

using calls to the Alpha CPU process cycle counter. We use this technique to measure the latency of the per-packet processing overhead by excluding the actual parsing and logging of those packets that contain HTTP headers, and then separately measuring the latency of parsing and logging for those packets that contain HTTP headers. For the one-week trace collected in May of 2000, we measured an average per-packet processing overhead of 4,200 cycles-per-packet, and an average HTTP parsing and logging overhead of 57,000 cycles-per-request or -response. Given the 500MHz clock of our tracing machine, these numbers correspond to 8.4 μs and 114 μs respectively.

During trace collection, we keep track of the amount of packet loss that occurs and record that information in our performance log. The Digital UNIX kernel records packet loss at both the device-driver level and at the kernel packet-filter level. We use the *ENTSTAMP* feature of the Digital UNIX packet-filter to extract this information from the kernel, and every two seconds we record an entry in our performance logs that reports packet reception and loss statistics on a per-interface basis. An example line would look like: `interface 0, pkts rcvd 8995, pkts dropped 0, if_overflow 0, qsize 0`. During the 1999 HTTP trace used in Chapters 3 and 4, we recorded a total of 49,115 dropped packets out of 6.4 billion packets received during the one-week trace, and we recorded a total of 88 interface overflows. Thus the overall packet loss rate for this trace was .0007%. Furthermore, these losses were not evenly distributed across the entire trace period. These losses occurred during the afternoon of Tuesday May 11th 1999 during a denial-of-service attack. During the May 2000 HTTP trace, we recorded zero dropped packets and a single interface overflow that occurred during the first two seconds of trace collection due to the structure of the initialization code. In addition to any packet loss that occurs at the tracing machine which we can detect, there is the possibility that the network switches performing the port mirroring may be dropping packets. We have no direct way of measuring any packet loss that occurs at the switches, but we would still like to have some assurance that we are not missing a significant portion of the traffic. To solve this problem, we generate synthetic Web traffic on a periodic basis from an internal UW client machine. We then detect those Web requests generated by our internal UW machine and count what percentage of those requests are seen by the tracing system. In all these experiments, 100% of the synthetically generated traffic was detected by our tracing system. We did not perform any of these experiments during the time period when an official one-week HTTP trace was collected.

We conclude our performance discussion with a brief speculative analysis on the scalability of our tracing system. During the time period from the Summer of 1998 when we began collecting HTTP traces to May of 2000, the load in terms of HTTP requests generated by the University of Washington client population grew by a factor of three. Using a combination of performance tuning the tracing software and upgrading the hardware from a single-machine configuration to a dual-machine configuration, our tracing system scaled to handle the traffic increase. In order to handle further load increases, one could continue to add tracing machines until there is a single tracing host per network switch. If the load scales to a point where a single machine cannot handle the traffic generated by the monitoring port of a single switch, we then need to modify our architecture. Modern Cisco Ethernet switches have the capability of supporting multiple SPAN ports on a single switch [Cisco Systems 02], so this approach could be used to attach more than one tracing host per switch. Another possibility is to use a front-end machine that sits in between the switch and a set of tracing machines. The front-end machine could be used to divide up the incoming traffic based on IP address ranges for the source addresses of incoming packets, and then forward traffic from specific address ranges to the same monitoring machine. This architecture should allow a relatively large number of back-end tracing machines to share the monitoring load from a single network switch.

6.7 Summary

In this chapter we presented a passive network monitoring system used to study the traffic characteristics of application-level Internet protocols. Our system was deployed at the Internet border of the University of Washington for almost three years, and it provided a low-overhead mechanism for observing the Web behavior of a large client population.

We provided a detailed look at the engineering details involved in building a high-speed packet monitoring system with the following properties: our system supports monitoring multiple network interfaces on the same machine simultaneously; our system supports asymmetric routing; it uses a very conservative approach to maintain privacy for network users, including the constraint that no raw packet data should ever be stored directly on disk; and our system performs extensive on-the-fly packet processing including TCP connection reconstruction and application-level protocol parsing as a consequence of our privacy design constraints.

We described a methodology for detecting and filtering bogus SYN packets from TCP SYN denial-of-service attacks that go through our tracing system. We presented the high-level architecture of our trace analysis tools, including our methods for supporting analyses such as cache simulators whose memory requirements when processing a large trace exceed the capacity of the analysis machine.

We characterized the performance of our tracing system including the traffic load, the percentage of time spent in different tasks, and the packet loss characteristics. Our tracing system handled peak data rates of up to 175 Mbits/second and up to 55,000 packets per second. During the deployment period, we used a combination of software performance tuning and hardware upgrades to handle the increase in traffic load.

Chapter 7

Conclusions

In this dissertation, we present the design and implementation of a network trace collection and analysis system for studying Internet application workloads. We demonstrate the effectiveness of this system by performing a set of three Internet traffic studies. The common theme across these studies is their focus on sharing. The amount of sharing in Web workloads plays an important role in many techniques used to improve the performance of Internet content delivery, such as proxy caching and multicast delivery. For example, the amount of sharing that occurs among a group of clients places an upper bound on the hit rate that a proxy cache serving those clients can achieve. The following sections summarize the findings of each of the three workload studies. We then summarize the design of the tracing system and conclude with a discussion of future research directions.

7.1 Organization-Based Sharing and Caching

The first workload study investigates the sharing and caching characteristics of Web documents from the perspective of organizations. An organizational analysis of sharing is important because Web proxy caches are typically deployed on an organizational basis. The principal goal of this study is to evaluate document sharing patterns on the Web, both *within* an organization and *across* multiple organizations. A secondary goal of this study is to understand the rate of access to uncachable documents from within an organization.

Little is known about document sharing patterns across multiple organizations, due to the difficulty of collecting simultaneous Web traces of multiple organizations. To address this problem, we use University of Washington internal organizations to model organizations connecting to the Internet. We identify 170 internal organizations within the University of Washington, and we classify each client in our trace as member of one of these organizations. Our trace collection system preserves this organizational membership information without compromising the privacy and

anonymity of clients using the UW network.

We find that when Web objects are simultaneously shared locally by an organization and shared globally with other organizations, they are more likely to be requested by an organization member than objects that are only shared locally or only shared globally. This suggests that the most-popular objects within an organization are also universally popular. When clients are members of the same organization, we also find there is a measurable increase in the amount of sharing when compared with clients that are members of different organizations. However, this increase is not large enough to have a significant impact on cache performance. When we examine the rate of access to uncachable documents, we find that the responses to 40% of all requests in our workload are uncachable – a rate that is noticeably higher than reported in previous studies.

7.2 Cooperative Caching

Our second study evaluates the performance of cooperative Web proxy caching by focusing on the effectiveness of cooperation over a wide range of client population sizes. One factor that limits the hit rate of a Web proxy cache is the total size of the client population managed by that cache. One technique for increasing the total size of the client population is to take multiple separate proxy caches and have them cooperate with each other. Increasing the client population size offers new opportunities for sharing, and therefore offers the potential to increase cache hit rates. Whether cooperative proxy caching is a useful architecture for improving performance depends on a number of factors. These factors include the sharing patterns of documents across organizations and the scale at which cooperation is undertaken.

We explore the potentials and limits of cooperative proxy caching using trace-based analysis. As with the previous study, we identify each client in our University of Washington trace as a member of one of 170 internal UW organizations. This provides us with the equivalent of 170 simultaneously collected traces of diverse and independent small organizations. We use these traces to measure the potential benefits of cooperation among organization-based proxies. In addition to our University of Washington trace, we collect a trace of proxy logs from the Microsoft corporation campus Web proxy caches during the same time period as we collect our UW network trace. We combine the Microsoft and UW traces to evaluate the benefit of cooperation between large organizations, each

with tens of thousands of clients.

Our results show that cooperative caching works well among collections of small organizations. But cooperative caching is not required for user populations of this size. If it is administratively and politically feasible, a single proxy cache can usually provide the same benefits with fewer resources and less overhead. We also show that cooperative caching is unlikely to provide significant benefits when combining large organizations or populations. Our experimental results indicate that the additional benefit of cooperation will be small, and therefore will only make sense if the cost of cooperation is also small. With current sharing patterns, there is little point in designing highly scalable cooperative-caching schemes; all reasonable schemes perform similarly in the low-end of the population range where cooperative caching works.

7.3 Streaming Media

Our final study is motivated by the observation that people are increasingly using the Internet to download and view multimedia content, such as streaming audio and video. Compared with traditional Web workloads, these multimedia objects may require significantly more storage and transmission bandwidth. As a result, performance optimizations such as streaming media proxy caches and multicast delivery offer the potential to minimize the impact of delivering this content over the Internet. However, few studies of streaming-media workloads exist. Therefore, the extent to which these mechanisms will improve performance is unclear.

This study presents a detailed analysis of the characteristics of a client-based streaming-media workload. As with the previous studies, we collect our workload using the network tracing system deployed at the University of Washington Internet border. The primary goal of our analysis is to characterize RTSP usage by the UW client population and to compare those characteristics to well-studied HTTP Web workloads in terms of bandwidth utilization, server and object popularity, and sharing patterns. We also evaluate the effectiveness of performance optimizations, such as proxy caching and multicast delivery, on our streaming-media workload.

We find that, as expected, streaming media objects are orders of magnitude larger than traditional Web objects. In particular, the median size of streaming media downloads is 400 times larger than the median HTTP request size, and the mean RTSP download size is 175 times larger than the

mean HTTP size. Nevertheless, it is still the case that *most* streaming media objects are modest in size ($<$ 1 MB), are encoded at relatively low bit-rates ($<$ 56 Kb/sec), and are short in duration ($<$ 10 mins). We observe a number of properties that indicate our streaming-media workload would currently benefit less from proxy caching than current HTTP workloads do. Finally, we find that multimedia workloads exhibit stronger temporal locality than we expected. During peak hours, between 20–40% of active sessions share streams concurrently. This suggests that multicast and stream-merging techniques may prove useful for these workloads.

7.4 Trace Collection and Analysis

We present the design and implementation of the network tracing and analysis system used to perform the previous caching studies. This system uses passive network monitoring to observe the Web behavior of a large and diverse group of clients. The tracing system is designed to monitor application-level Internet protocols, with the current implementation supporting HTTP and RTSP. It is installed at the Internet border of the University of Washington (UW) and it observes all Internet traffic that flows through the border. The tracing system was in use at the UW border for approximately three years, from the Summer of 1998 through the Spring of 2001. During that time period, the traffic load grew by a factor of three and the tracing system scaled to keep up with the load increases with negligible packet loss. Our tracing system handled peak data rates of up to 175 Mbits/second and up to 55,000 packets per second.

The tracing system we developed has the following properties: it supports high-speed packet monitoring with multiple network interfaces on the same machine; it supports monitoring connections with asymmetric routing paths; it uses a very conservative approach to maintaining privacy for network users, including the constraint that no raw packet data will ever be stored directly on disk; and it performs on-the-fly processing including TCP connection reconstruction and application-level protocol parsing as a consequence of the privacy design constraints. Finally, as part of the tracing system deployment we were forced to develop a methodology for detecting and filtering bogus SYN packets from TCP SYN denial-of-service attacks that go through our tracing system.

7.5 Future Work

In this section, we discuss future research directions related to the sharing and caching workload studies in this dissertation, and we discuss future research directions enabled by the development of our tracing system. The future directions we discuss fall into three categories: analysis of new application workloads; analysis of new user populations; and questions that remain unanswered in the workload areas studied in this dissertation.

The Internet has experienced tremendous growth in recent years, not only in terms of new users, but also in terms of new applications. Our study of streaming-media workloads examined the characteristics of one such emerging application. We believe that the studying application workloads will be important for the following new classes of applications.

One recent trend in Internet usage is peer-to-peer file sharing applications. This trend began with Napster, an application dedicated to sharing of popular music audio files. In the Spring of 2001, the Napster service was terminated through a legal action in the United States, but a huge number of replacement peer-to-peer applications were created to fill the void. Examples include Gnutella, Kazaa, eDonkey, DirectConnect, Morpheus, and Blubster. These applications are used to share many different forms of content including music, movies, application binaries, and images. Similar to the situation with streaming-media protocols, many of the these peer-to-peer applications use proprietary and undocumented protocols. However, the situation is not entirely bleak for the purposes of application-level protocol monitoring because these proprietary protocols usually implement only the search functionality, and the actual file transfers are performed using a standard protocol such as HTTP or FTP.

Within the last few years, both cell phones and PDAs have begun to acquire Internet access functionality. Because these devices are often resource constrained in terms of CPU, memory, and power, detailed performance analysis of these new applications including their network behavior has the potential to provide significant improvements for both end-user performance and energy consumption. Many hand-held mobile devices currently use WAP 1.0 and are expected to implement WAP 2.0 in the future. WAP is the Wireless Application Protocol, a standardized protocol to enable delivery of information and interactive applications to digital wireless mobile devices. WAP 1.0 requires the use of a protocol gateway to convert from the WAP transport protocols to the standard

Internet protocols, whereas WAP 2.0 makes explicit use of the Internet protocol suite including TCP and HTTP.

XML-based Web services is the final new class of applications where we believe future workload analysis research is needed. Broadly defined, Web services provide the infrastructure for building distributed applications that use the Internet for communication. The Web services standards allow different applications from different vendors to communicate and interoperate. In addition to support for basic inter-machine communication, Web services includes support for application discovery and typed data exchange. The Web services standardization efforts also attempt to ensure that applications are not tied to any single operating system or programming language. The major components of Web services are: SOAP, the Simple Object Access Protocol which is layered on top of HTTP; UDDI, the Universal Discovery, Description, and Integration protocol; and WSDL, the Web Services Description Language.

Although there is not widespread deployment of Web services applications at the present time, major software companies such as IBM, BEA, Microsoft, and Sun are all investing significant resources into the development of Web services software. The expectation is that the Web services protocols will form the basis for most future electronic business applications. With respect to caching, one key difference between Web services workloads and traditional HTTP workloads is that the communication endpoints for Web services are programs rather than people, and programs aren't nearly as good at dealing with inconsistent data as people are. In other words, people can often identify stale data and decide to hit the reload button on a Web browser in those cases.

When considering the user populations that have been studied by previous research and by this dissertation, we find that most tracing efforts have studied the characteristics of University client populations. A few studies have also looked at corporate client populations, but one large segment of the population which remains unstudied is clients that access the Internet through a commercial Internet Service Provider (ISP). We think it would be particularly interesting to study the Web traffic characteristics from a commercial ISP that offers broadband service such as DSL or cable-modem service because broadband users are more likely to be representative of future Internet usage characteristics.

Even in the area of sharing and caching characteristics of traditional HTTP workloads, there are still questions that need to be resolved. For example, if browser caches could cooperate would

that eliminate the need for organizational proxy caches? To what extent does HTTP content exhibit temporal locality over relatively long time-scales (such as weeks or even months)?

Finally, our work and others show that clients currently generate a high rate of requests to uncachable documents. Others also show that the server response content for many of these uncachable documents does not change frequently. Understanding why these documents are marked uncachable would be useful in order to learn how Web software should be changed to prevent unnecessary retransmissions of the same content from servers to clients. This question would best be answered in the context of a raw trace rather than an anonymized trace. A raw trace would allow better attribution of the causes for uncachable Web content. For instance, one could determine which uncachable pages are banner advertisements and which are product databases for an e-commerce site.

Bibliography

[Acharya et al. 98] S. Acharya and B. Smith. An experiment to characterize videos stored on the web. In *Proc. of ACM/SPIE Multimedia Computing and Networking 1998*, Jan 1998.

[Agarwal et al. 98] R. Agarwal, J. Ayars, B. Hefta-Gaub, and D. Stammen. RealMedia File Format. Internet Draft: draft-heftagaub-rmff-00.txt, Mar 1998.

[Almeida et al. 96] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In *Proc. of IEEE Intl. Conference on Parallel and Distributed Information Systems '96*, Dec 1996.

[Anderson et al. 96] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Trans. on Computer Systems*, 14(1):41–79, Feb 1996.

[Anderson et al. 97] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. of the Sixteenth Symposium on Operating Systems Principles*, Oct 1997.

[Apisdorf et al. 97] J. Apisdorf, K. Claffy, K. Thompson, and R. Wilder. OC3MON: Flexible, affordable, high-performance statistics collection. In *Proc. of INET 97*, Jun 1997.

[Balakrishnan et al. 98] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. Katz. TCP behavior of a busy internet server: Analysis and improvements. In *Proc. of IEEE INFOCOM 1998*, Mar 1998.

[Berners-Lee et al. 96] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0. ftp://ftp.isi.edu/in-notes/rfc1945.txt, May 1996.

[Breslau et al. 99] 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 1999*, pages 126–134, Mar 1999.

[Caceres et al. 98] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: The devil is in the details. In *Proc. of the Workshop on Internet Server Performance*, Jun 1998.

[Cao et al. 97] P. Cao and C. Liu. Maintaining strong consistency in the world-wide web. In *Proc. of the Seventeenth Intl. Conference on Distributed Computing Systems*, May 1997.

[Cao et al. 98] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proc. of the IFIP Intl. Conference on Distributed Systems Platforms and Open Distributed Processing*, Sep 1998.

[Cate 92] V. Cate. Alex – a global filesystem. In *Proc. of the 1992 USENIX File Systems Workshop*, May 1992.

[Challenger et al. 99] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proc. of IEEE INFOCOM 1999*, Mar 1999.

[Chankhuntod et al. 96] A. Chankhuntod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proc. of the 1996 USENIX Technical Conference*, Jan 1996.

[Chesire et al. 01] M. Chesire, A. Wolman, G. M. Voelker, and H. M. Levy. Measurement and analysis of a streaming-media workload. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems*, Mar 2001.

[Cisco Systems 02] Cisco Systems. Catalyst 4000 family software configuration guide. http://www.cisco.com/univercd/cc/td/doc/product/lan/cat4000/rel7_1/config/span.htm, [Accessed July 2002].

[Cunha et al. 95] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of www client-based traces. Technical Report TR-95-010, Boston University, Computer Science Department, Apr 1995.

[Deutsch 96] P. Deutsch. RFC 1952: GZIP file format sepcification version 4.3. ftp://ftp.isi.edu/in-notes/rfc1952.txt, May 1996.

[Douglis et al. 97a] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: a live study of the World Wide Web. Technical Report 97.24.2, AT&T Labs, Dec 1997.

[Douglis et al. 97b] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: a live study of the World Wide Web. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems*, pages 147–158, Dec 1997.

[Douglis et al. 97c] F. Douglis, A. Haro, and M. Rabinovich. Hpp: Html macro-preprocessing to support dynamic document caching. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems*, Dec 1997.

[Duska et al. 97] B. M. Duska, D. Marwood, and M. J. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems*, Dec 1997.

[Dykes et al. 02] S. G. Dykes and K. A. Robbins. Limitations and benefits of cooperative proxy caching. *IEEE Journal on Selected Areas in Communications*, 2002.

[Eager et al. 00] D. Eager, M. Vernon, and J. Zahorjan. Bandwidth skimming: A technique for cost-effective video-on-demand. In *Proc. of ACM/SPIE Multimedia Computing and Networking 2000*, Jan 2000.

[Eager et al. 99] D. Eager, M. Vernon, and J. Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. In *Proc. of the 5th Int'l Workshop on Multimedia Information Systems*, Oct 1999.

[Engler et al. 96] D. R. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. of ACM SIGCOMM 1996*, Aug 1996.

[Fan et al. 98] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area cache sharing protocol. In *Proc. of ACM SIGCOMM 1998*, Sep 1998.

[Feeley et al. 95] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proc. of the Fifteenth Symposium on Operating Systems Principles*, Dec 1995.

[Feldmann 00] A. Feldmann. BLT: Bi-Layer Tracing of HTTP and TCP/IP. In *Proc. of the Ninth Int. World Wide Web Conference*, May 2000.

[Feldmann et al. 99] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *Proc. of IEEE INFOCOM 1999*, pages 107–116, Mar 1999.

[Fielding et al. 99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1. ftp://ftp.isi.edu/in-notes/rfc2616.txt, Jun 1999.

[Fleischman 98] E. Fleischman. Advanced Streaming Format (ASF) Specification. Internet-Draft: draft-fleischman-asf-01.txt, Feb 1998.

[Fraleigh et al. 01] C. Fraleigh, C. Diot, B. Lyles, S. B. Moon, P. Owezarski, D. Papagiannaki, and F. A. Tobagi. Design and deployment of a passive monitoring infrastructure. In *Proceedings of PAM 2001: A Workshop on Passive and Active Measurements*, Apr 2001.

[Freier et al. 96] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol Version 3.0. Internet-Draft: http://wp.netscape.com/eng/ssl3/draft-302.txt, Nov 1996.

[Gadde et al. 00] S. Gadde, J. Chase, and M. Rabinovich. Web caching and content distribution: A view from the interior. In *Proc. of the Fifth Int. Web Caching and Content Delivery Workshop*, May 2000.

[Glassman 94] S. Glassman. A caching relay for the World Wide Web. In *Proc. of the First Int. World Wide Web Conference*, May 1994.

[Gribble et al. 97] S. D. Gribble and E. A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems*, pages 207–218, Dec 1997.

[Gwertzman et al. 95] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proceedings of 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[Gwertzman et al. 96] J. Gwertzman and M. Seltzer. World wide web cache consistency. In *Proc. of the 1996 USENIX Technical Conference*, Jan 1996.

[Handley et al. 98] M. Handley and V. Jacobson. RFC 2327: SDP: Session Description Protocol, Apr 1998.

[Huberman et al. 98] B. Huberman, P. Pirolli, J. Pitkow, and R. Lukose. Strong regularities in world wide web surfing. *Science*, 280, Apr 1998.

[Hunt et al. 96] J. Hunt, K.-P. Vo, and W. Tichy. An empirical study of delta algorithms. In *Proc. of the IEEE Software and Configuration Maintenance Workshop 1996*, Mar 1996.

[Karger et al. 99] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proc. of the Eighth Int. World Wide Web Conference*, May 1999.

[Kent et al. 98] S. Kent and R. Atkinson. RFC 2401: Security Architecture for the Internet Protocol, Nov 1998.

[Keynote 01] Keynote perspective services [web page]. http://www.keynote.com/services/, [Accessed July 2001].

[Krishnamurthy et al. 00] B. Krishnamurthy and C. Wills. Analyzing factors that influence end-to-end web performance. In *Proc. of the Ninth Int. World Wide Web Conference*, May 2000.

[Krishnamurthy et al. 97] B. Krishnamurthy and C. E. Wills. Study of piggyback cache validation for proxy caches in the World Wide Web. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems*, Dec 1997.

[Krishnan et al. 98] P. Krishnan and B. Sugla. Utility of co-operating web proxy caches. In *Proc. of the Seventh Int. World Wide Web Conference*, Apr 1998.

[Kristol et al. 97] D. Kristol and L. Montulli. RFC 2109: HTTP State Management Mechanism. ftp://ftp.isi.edu/in-notes/rfc2109.txt, Feb 1997.

[Kroeger et al. 97] T. M. Kroeger, D. D. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems*, pages 13–22, Dec 1997.

[Kurcewicz et al. 98] M. Kurcewicz, W. Sylwestrzak, and A. Wierzbicki. A distributed WWW cache. In *Proc. of the Third Int. Web Caching Workshop*, Jun 1998.

[Li 92] W. Li. Random texts exhibit zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory*, 1992.

[Mah 97] B. A. Mah. An empirical model of HTTP network traffic. In *Proc. of IEEE INFOCOM 1997*, pages 592–600, Apr 1997.

[Malan et al. 98] G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proc. of ACM SIGCOMM 1998*, Sep 1998.

[Manley et al. 97] S. Manley and M. Seltzer. Web facts and fantasy. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems*, Dec 1997.

[McCanne et al. 93] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proc. of the Winter 1993 USENIX Technical Conference*, 1993.

[McCreary et al. 00] S. McCreary and K. Claffy. Trends in Wide Area IP Traffic Patterns: A View from Ames Internet Exchange [web page]. http://www.caida.org/outreach/papers/AIX0005/, May 2000.

[Mena et al. 00] A. Mena and J. Heidemann. An empirical study of real audio traffic. In *Proc. of IEEE INFOCOM 2000*, Mar 2000.

[Michel et al. 98] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching: Towards a New Global Caching Architecture. *Computer Networks and ISDN Systems*, 30(22–23):2169–2177, Nov 1998.

[Microsoft 01a] Microsoft. All About Windows Media Metafiles [web page]. http://msdn.microsoft.com/workshop/imedia/windowsmedia/crcontent/asx.asp, [Accessed July 2001].

[Microsoft 01b] Microsoft. Windows Media Development Center [web page]. http://msdn.microsoft.com/windowsmedia/, [Accessed July 2001].

[Mills 91] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, Oct 1991.

[Mogul et al. 87] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. of the Eleventh Symposium on Operating Systems Principles*, Nov 1987.

[Mogul et al. 97] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *Proc. of ACM SIGCOMM 1997*, Sep 1997.

[MPEG-2 Audio 94] MPEG-2 Standard. ISO/IEC Document 13818-3. Generic Coding of Moving Pictures and Associated Audio Information, Part 3: Audio., 1994.

[MPEG-2 Video 94] MPEG-2 Standard. ISO/IEC Document 13818-2. Generic Coding of Moving Pictures and Associated Audio Information, Part 2: Video., 1994.

[Netscape 95] Netscape. Persistent client state - http cookies [web page]. http://home.netscape.com/newsref/std/cookie_spec.html, 1995.

[Nielsen 01] Nielsen Netratings. http://www.nielsen-netratings.com/ [web page], [Accessed July 2001].

[Oberhumer 02] M. F. Oberhumer. Lzo compression library [web page]. http://www.oberhumer.com/opensource/lzo/, [Accessed July 2002].

[Padmanabhan et al. 00] V. Padmanabhan and L. Qiu. The content and access dynamics of a busy web site: Findings and implications. In *Proc. of ACM SIGCOMM 2000*, Aug 2000.

[Paxson 97] V. Paxson. End-to-end Internet packet dynamics. In *Proc. of ACM SIGCOMM 1997*, Sep 1997.

[PGP 01] Pretty good privacy [web page]. http://www.pgpi.com/, [Accessed July 2001].

[Plonka 00] D. Plonka. UW-Madison Napster Traffic Measurement [web page]. http://net.doit.wisc.edu/data/Napster, Mar 2000.

[Rabinovich et al. 98] M. Rabinovich, J. Chase, and S. Gadde. Not all hits are created equal: Co-operative proxy caching over a wide area network. In *Proc. of the Third Int. Web Caching Workshop*, Jun 1998.

[Ranum et al. 97] M. J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a generalized tool for network monitoring. In *Proc. of the 11th Systems Administration Conference(LISA '97)*, Oct 1997.

[RealNetworks 01a] Firewall PNA Proxy Kit [web page]. http://www.service.real.com/firewall/-pnaproxy.html, [Accessed January 2001].

[RealNetworks 01b] RealNetworks Documentation Library [web page]. http://service.real.com/help/library/, [Accessed January 2001].

[RealNetworks 01c] Realsystem production and authoring guides [web page]. http://service.real.com/help/library/encoders.html, [Accessed January 2001].

[Rejaie et al. 99] R. Rejaie, M. Handley, H. Yu, and D. Estrin. Proxy caching mechanism for multimedia playback streams in the internet. In *Proc. of the Fourth Int. Web Caching Workshop*, Mar 1999.

[Rivest 92] R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm, April 1992.

[Schulzrinne et al. 96] H. Schulzrinne, S. Casner, R. Fredrick, and V. Jacobson. RFC 1889: RTP: A Transport Protocol for Real-Time Applications, April 1996.

[Schulzrinne et al. 98] H. Schulzrinne, A. Rao, and R. Lanphier. RFC 2326: Real Time Streaming Protocol (RTSP), Apr 1998.

[Sen et al. 99] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *Proc. of IEEE INFOCOM 1999*, Mar 1999.

[Shrager et al. 87] J. Shrager, T. Hogg, and B. Huberman. Observation of phase transitions in spreading activation networks. *Science*, 236, 1987.

[Smith et al. 01] F. D. Smith, F. H. Campos, K. Jeffay, and D. Ott. What TCP/IP protocol headers can tell us about the web. In *Proc. of the ACM SIGMETRICS 2001 Conference*, Jun 2001.

[Smith et al. 99] B. Smith, A. Acharya, T. Yang, and H. Zhu. Exploiting result equivalence in caching dynamic web content. In *Proc. of the 2nd USENIX Symposium on Internet Technologies and Systems*, Oct 1999.

[Squid 01] Squid Internet Object Cache [web page]. http://squid.nlanr.net, [Accessed July 2001].

[Tcpdump 01] Tcpdump [web page]. http://www.tcpdump.org/, [Accessed July 2001].

[Tewari et al. 99] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design considerations for distributed caching on the Internet. In *Proc. of the Nineteenth Intl. Conference on Distributed Computing Systems*, May 1999.

[Touch 98] J. Touch. The LSAM proxy cache - a multicast distributed virtual cache. In *Proc. of the Third Int. Web Caching Workshop*, Jun 1998.

[Valloppillil et al. 98] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. ftp://ftp.isi.edu/internet-drafts/draft-vinod-carp-v1-03.txt, Feb 1998.

[Van Der Merwe et al. 00] J. Van Der Merwe, R. Caceres, Y. hua Chu, and C. Sreenan. mmdump - a tool for monitoring multimedia usage on the internet. Technical Report 00.2.1, AT&T Labs, Feb 2000.

[W3C 98] W3C. Synchronized Multimedia Integration Language (SMIL) 1.0 Specification [web page]. http://www.w3.org/TR/1998/REC-smil-19980615/, Jun 1998.

[Warren et al. 99] P. Warren, C. Boldyreff, and M. Munro. Characterizing evolution in web sites: Some case studies. In *Proc. of the First Intl. Workshop on Web Site Evolution*, Oct 1999.

[Wills et al. 99a] C. E. Wills and M. Mikhailov. Examining the cacheability of user-requested web resources. In *Proc. of the Fourth Int. Web Caching Workshop*, Apr 1999.

[Wills et al. 99b] C. E. Wills and M. Mikhailov. Towards a better understanding of web resources and server responses for improved caching. In *Proc. of the Eighth Int. World Wide Web Conference*, pages 153–165, May 1999.

[Wolman et al. 99a] 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 Symposium on Internet Technologies and Systems*, Oct 1999.

[Wolman et al. 99b] 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 Seventeenth Symposium on Operating Systems Principles*, Dec 1999.

[Wooster et al. 96] R. Wooster, S. Williams, and P. Brooks. Httpdump: A network http packet snooper. Technical report, Apr 1996.

[Worrell 94] K. J. Worrell. Invalidation in large scale network object caches. Master's thesis, Department of Computer Science, Univ. of Colorado Boulder, 1994.

[Xingtech 00] Xingtech. Streamworks documentation [web page]. http://www.xingtech.com/support/docs/streamworks/, [Accessed July 2000].

[Yuhara et al. 94] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proc. of the Winter 1994 USENIX Technical Conference*, Jan 1994.

[Zhang et al. 97] L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *Proc. of the Second Int. Web Caching Workshop*, Jun 1997.

[Zipf 49] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

Vita

Alastair Wolman was born on November 2nd, 1966 in Morristown, New Jersey. He grew up in Rumson, New Jersey and attended high school at Middlesex School in Concord, Massachusetts. He attended Harvard University, where he received his B.A. in Computer Science in 1988. For the next four years, he worked for Digital Equipment Corporation at the Cambridge Research Lab in Cambridge, Massachusetts. In 1992, he headed west to Seattle, Washington for graduate school at the University of Washington. He received his Master of Science degree in 1995, and his Ph.D. in 2002. In November 2001, he started work in the Systems and Networking group at Microsoft Research in Redmond, Washington.